
Sarus Documentation

Release 1.6.0

Swiss National Supercomputing Centre

May 08, 2023

CONTENTS:

1	Publications	3
1.1	Quickstart	3
1.2	Overview	5
1.3	Custom installation	10
1.4	Configuration	21
1.5	User guides	48
1.6	Developer documentation	73
1.7	Sarus Cookbook	79
1.8	Glossary	105
2	Indices and tables	109
	Bibliography	111
	Index	113

release: 1.6.0

Sarus is a software to run Linux containers on High Performance Computing environments. Its development has been driven by the specific requirements of HPC systems, while leveraging open standards and technologies to encourage vendor and community involvement.

Key features:

- Spawning of isolated software environments (*containers*), built by users to fit the deployment of a specific application
- Security oriented to HPC systems
- Extensible runtime by means of OCI hooks to allow current and future support of custom hardware while achieving native performance
- Creation of container filesystems tailored for diskless nodes and parallel filesystems
- Compatibility with the presence of a workload manager
- Compatibility with the Open Container Initiative (OCI) standards:
 - Can pull images from registries adopting the OCI Distribution Specification or the Docker Registry HTTP API V2 protocol
 - Can import and convert images adopting the OCI Image Format
 - Sets up a container bundle complying to the OCI Runtime Specification
 - Uses an OCI-compliant runtime to spawn the container process

PUBLICATIONS

- Benedicic, L., Cruz, F.A., Madonna, A. and Mariotti, K., 2019, June. Sarus: Highly Scalable Docker Containers for HPC Systems. In *International Conference on High Performance Computing* (pp. 46-60). Springer, Cham.
https://doi.org/10.1007/978-3-030-34356-9_5

1.1 Quickstart

This page provides instructions to quickly deploy Sarus using a standalone archive containing statically-linked binaries for the engine, the hooks and some dependency software.

The content in this page is intended for those interested in installing and using Sarus without building the software beforehand. If you want to build Sarus from source or through the Spack package manager, please refer to the [Custom installation](#) section.

1.1.1 Install Sarus

You can install Sarus by following the steps below.

1. Download the latest standalone Sarus archive from the official [GitHub Releases](#):

```
mkdir /opt/sarus
cd /opt/sarus
# Adjust url to your preferred version
wget https://github.com/eth-cscs/sarus/releases/download/1.5.0/sarus-Release.tar.gz
```

2. Extract Sarus in the installation directory:

```
cd /opt/sarus
tar xf sarus-Release.tar.gz
```

3. Run the [configuration script](#) to finalize the installation of Sarus:

```
cd /opt/sarus/1.5.0-Release # adapt folder name to actual version of Sarus
sudo ./configure_installation.sh
```

Important: The configuration script needs to run with root privileges in order to set Sarus as a root-owned SUID program.

The configuration script requires the program `mksquashfs` to be installed on the system, which is typically available through the `squashfs-tools` package.

As explained by the output of the script, you need to persistently add Sarus to your PATH; one option is to add a line like `export PATH=/opt/sarus/bin:${PATH}` to your `.bashrc` file.

Also note that the configuration script will create a minimal working configuration. For enabling additional features, please refer to the [Configuration file reference](#).

4. Perform the *Post-installation actions*.

Important: The Skopeo static binary included in the Sarus standalone archive is intended for test and evaluation purposes only!

Static builds of Skopeo are **not supported** by Skopeo maintainers, and are unsuitable for production deployments.

For production uses, it is recommended to [configure Sarus](#) to use a dedicated, dynamically-linked Skopeo binary (either [installed](#) through the package manager or built from source).

1.1.2 Use Sarus

Now Sarus is ready to be used. Below is a list of the available commands:

```
help: Print help message about a command
images: List images
load: Load the contents of a tarball to create a filesystem image
pull: Pull an image from a registry
rmi: Remove an image
run: Run a command in a new container
ssh-keygen: Generate the SSH keys in the local repository
version: Show the Sarus version information
```

Below is an example of some basic usage of Sarus:

```
$ sarus pull alpine
# image           : docker.io/library/alpine:latest
# cache directory : "/home/user/.sarus/cache"
# temp directory  : "/tmp"
# images directory: "/home/user/.sarus/images"
# image digest    : ↪
↪sha256:4ff3ca91275773af45cb4b0834e12b7eb47d1c18f770a0b151381cd227f4c253
Getting image source signatures
Copying blob 2408cc74d12b done
Copying config a366738a18 done
Writing manifest to image destination
Storing signatures
> unpacking OCI image
> making squashfs image: "/home/user/.sarus/images/docker.io/library/alpine/latest.
↪squashfs"

$ sarus images
REPOSITORY TAG          IMAGE ID          CREATED          SIZE          SERVER
alpine     latest          a366738a1861     2022-05-25T09:19:59 2.59MB        docker.io

$ sarus run alpine cat /etc/os-release
```

(continues on next page)

(continued from previous page)

```
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.16.0
PRETTY_NAME="Alpine Linux v3.16"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://gitlab.alpinelinux.org/alpine/aports/-/issues"
```

Note: You can refer to the section *User guides* for more information about how to use Sarus.

1.2 Overview

Sarus is a tool for High-Performance Computing (HPC) systems that provides a user-friendly way to instantiate feature-rich containers from OCI images. It has been designed to address the unique requirements of HPC installations, such as: native performance from dedicated hardware, improved security due to the multi-tenant nature of the systems, support for network parallel filesystems and diskless computing nodes, compatibility with a workload manager or job scheduler.

Keeping flexibility, extensibility and community efforts in high regard, Sarus relies on industry standards and open source software. Consider for instance the use of `runc`, an OCI-compliant container runtime that is also [used internally by Docker](#). Moreover, Sarus leverages the [Open Containers Initiative \(OCI\)](#) specifications to extend the capabilities of `runc` and enable multiple high-performance features. In the same vein, Sarus depends on a widely-used set of libraries, tools, and technologies to reap several benefits: reduce maintenance effort and lower the entry barrier for service providers wishing to install the software, or for developers seeking to contribute code.

1.2.1 Sarus architecture

The workflows supported by Sarus are implemented through the interaction of several software components.

The **CLI** component processes the program arguments entered through the command line and calls other components to perform the actions requested by the user.

The **Image Manager** component is responsible for:

- coordinating the import of container images onto the system (making use of utilities like [Skopeo](#) and [Umoci](#));
- converting imported images to Sarus' own format;
- storing images on local system repositories;
- querying the contents of local repositories.

The **Runtime** component creates and executes containers, first by setting up a bundle according to the OCI Runtime Specification: such bundle is made of a root filesystem directory for the container and a JSON configuration file. After preparing the bundle, the Runtime component will call an external **OCI runtime** (such as `runc`), which will effectively spawn the container process. Sarus can instruct the OCI runtime to use one or more [OCI hooks](#), small programs able to perform actions (usually related to container customization) at selected points during the container lifetime.

In the following sections, we will illustrate how Sarus performs some of its fundamental operations, highlighting the motivations and purposes of the implemented solutions.

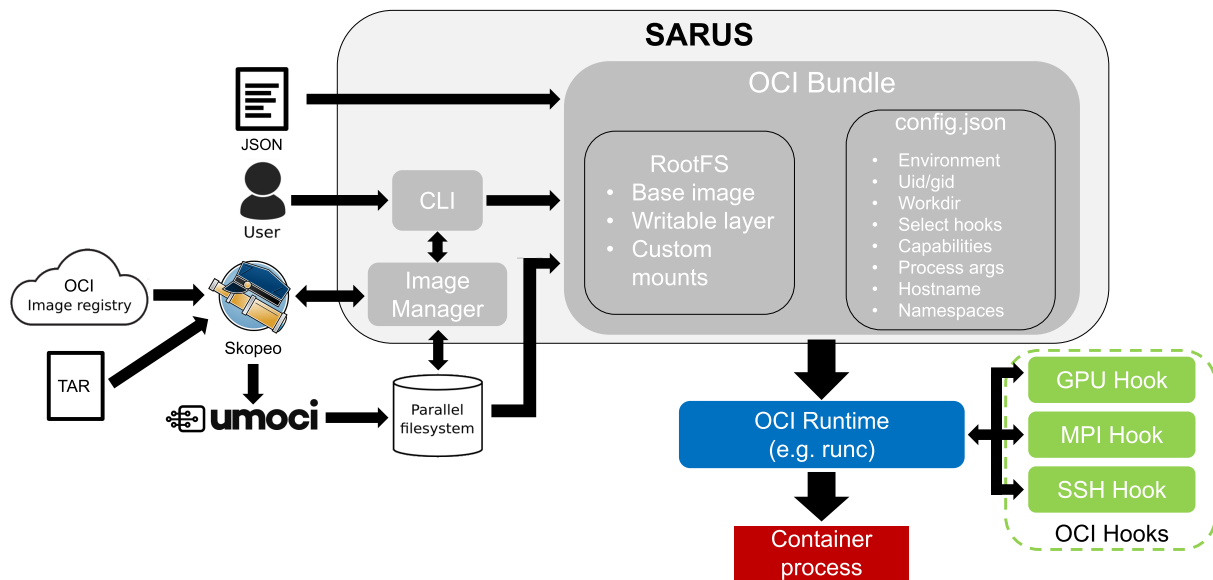


Fig. 1: Sarus architecture diagram

1.2.2 Importing container images

One of the first actions a user will perform with a fresh Sarus installation is getting some container images in a form readily usable by the program. This is accomplished by either:

- retrieving images from a remote registry (e.g. [Docker Hub](#)) using the **sarus pull** command;
- loading images from local **tar** archives using the **sarus load** command.

Sarus imports images through [Skopeo](#), a highly versatile utility able to perform operations on image repositories, transfer container images and convert them between different formats. Skopeo is instructed to retrieve the image data (whether located on a remote registry or in a local file) and transform it into an OCI image directory, that is a directory structured according to the [OCI Image Specification](#).

Once the container image is available in such a standard format, the image filesystem is *unpacked* into a plain *temporary directory* using [Umoci](#). Umoci is a utility to create, modify and interact with OCI images, serving also as a reference implementation of the OCI Image Specification. The unpack operation has the additional effect of reducing the final size of Sarus images on disk, by merging together layers and potentially duplicated data.

The image filesystem contents are then squashed together, obtaining a *flattened* image in the [SquashFS](#) format. A metadata file is also generated from a subset of the OCI image configuration. Squashing the image improves the I/O performance of the container, as detailed below in [Root filesystem](#).

After the SquashFS image and the metadata file are generated, Sarus copies them into the local user repository, described in the next section.

1.2.3 Local system repositories

Sarus stores images available on the system in **local repositories**, which are individual for each user. The application expects to access a configurable *base path*, where directories named after users are located. Sarus will look for a local repository in the `<base path>/<user name>/ .sarus` path. If a repository does not exist, a new, empty one is created.

A Sarus local repository is a directory containing at least:

- the *cache* directory for downloaded image data: this directory is used by Sarus to store downloaded individual image components (like filesystem layers and OCI configuration files), so they can be reused by subsequent pull commands;
- the *images* directory for Sarus images: inside this directory, images are stored in a hierarchy with the format `<registry server>/<repository>/<image name>`, designed to replicate the structure of the strings used to identify images. At the end of a pull or load process, Sarus copies the image SquashFS and metadata files into the last folder of the hierarchy, named after the image, and sets the names of both files to match the image tag;
- the *metadata.json* file indexing the contents of the images folder

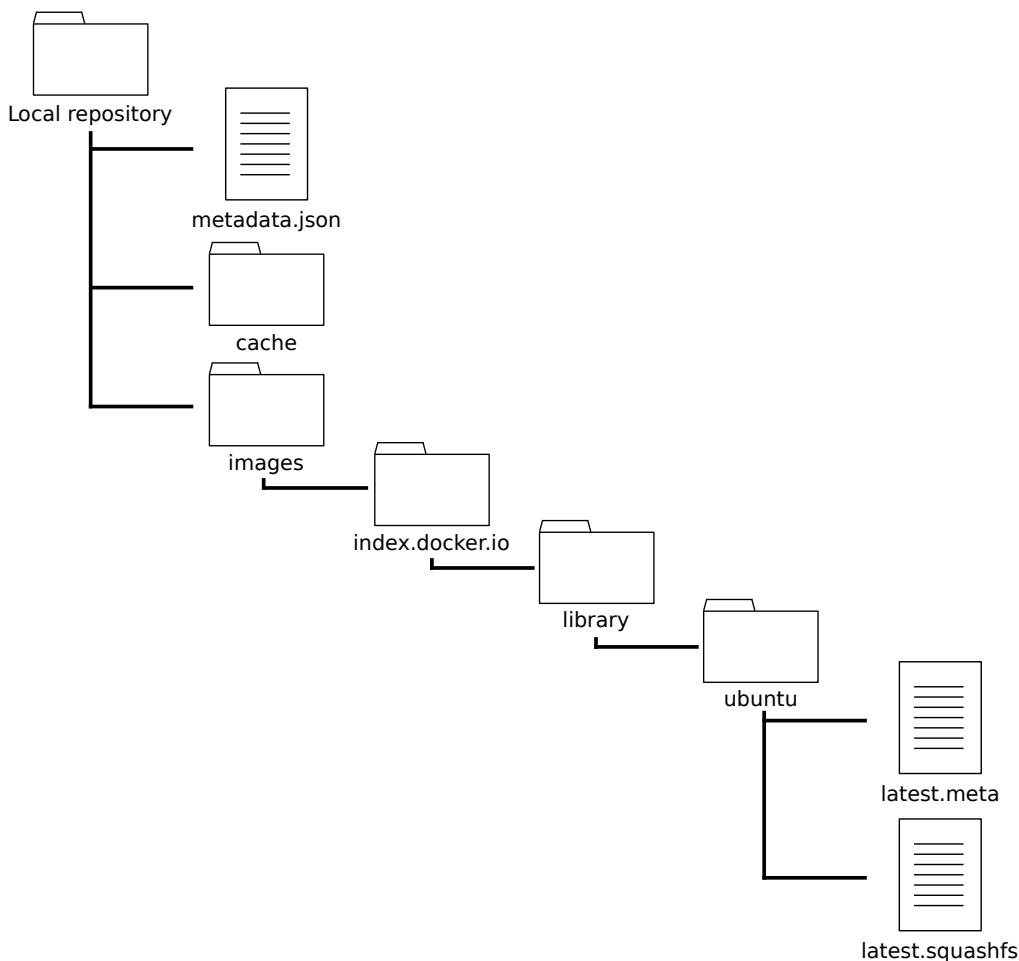


Fig. 2: Structure of a Sarus local repository

Sarus can also be configured to create a system-wide *centralized repository*. Such repository is intended to broadcast images to users, e.g. in cases when said images cannot be freely redistributed. The centralized repository is meant to be read-only for regular users, and its contents should be modifiable only by the system administrators.

Users can query the contents of the individual and centralized repositories using the **sarus images** command.

1.2.4 Container instantiation

The Runtime component of Sarus is responsible for setting up and coordinating the launch of container instances. When the user requests the execution of a container process through the **sarus run** command, an OCI bundle is first created in a *dedicated directory*. As mentioned above, an OCI bundle is defined by the OCI Runtime Specification as the content from which an OCI-compliant low-level runtime, e.g. runc, will spawn a container. The bundle is formed by a *rootfs* directory, containing the root filesystem for the container, and a *config.json* file providing detailed settings to the OCI runtime.

Before actually generating the contents of the bundle, Sarus will create and join a new Linux mount namespace in order to make the mount points of the container inaccessible from the host system. An *in-memory temporary filesystem* is then mounted on the directory designated to host the OCI bundle. This process yields several beneficial effects, e.g.:

- Unsharing the mount namespace prevents other processes of the host system from having visibility on any artifact related to the container instance [[unshare-manpage](#)] [[mount-namespaces-manpage](#)].
- The newly-created mount namespace will be deleted once the container and Sarus processes exit; thus, setting up the bundle in a filesystem that belongs only to the mount namespace of the Sarus process ensures complete cleanup of container resources upon termination.
- Creating the bundle, and consequently the container rootfs, in an in-memory temporary filesystem improves the performance of the container writable layer. This solution also suits diskless computing nodes (e.g. as those found in Cray XC systems), where the host filesystem also resides in RAM.

In the next subsections, we will describe the generation of the bundle contents in more detail.

Root filesystem

The root filesystem for the container is assembled in a *dedicated directory* inside the OCI bundle location through several steps:

1. The SquashFS file corresponding to the image requested by the user is mounted as a *loop device* on the configured rootfs mount point. The loop mount allows access to the image filesystem as if it resided on a real block device (i.e. a storage drive). Since Sarus images are likely to be stored on network parallel filesystems, reading multiple different files from the image¹ causes the thrashing of filesystem metadata, and consequently a significant performance degradation. Loop mounting the image prevents metadata thrashing and improves caching behavior, as all container instances access a single SquashFS file on the parallel filesystem. The effectiveness of this approach has already been demonstrated by Shifter [[ShifterCUG2015](#)].
2. Sarus proceeds to create an *overlay filesystem*. An overlay filesystem, as the name suggests, is formed by two different filesystem layers on top of each other (thus called respectively *upper* and *lower*), but it is presented as a single entity which combines both. The loop-mounted image is re-used as the *read-only* lower layer, while part of the OCI bundle temporary filesystem forms the *writable* upper layer. An overlay filesystem allows the contents of Sarus containers to be transparently modifiable by the users, while preserving the integrity of container images: modifications exist only in the overlay upper filesystem, while corresponding entries in the lower filesystem are hidden. Please refer to the official [OverlayFS](#) documentation for more details.
3. Selected system configuration files (e.g. */etc/hosts*, */etc/passwd*, */etc/group*) are copied into the rootfs of the container. These configurations are required to properly setup core functionality of the container in a multi-tenant cluster system, for example file permissions in shared directories, or networking with other computing nodes.
4. *Custom mounts* are performed. These are bind mounts requested by the *system administrator* or by the *user* to customize the container according to the needs and resources of an HPC system or a specific use case.

¹ A prominent use case is, for example, a Python application.

5. The container's rootfs is completed by finally **remounting** the filesystem to remove potential suid bits from all its files and directories.

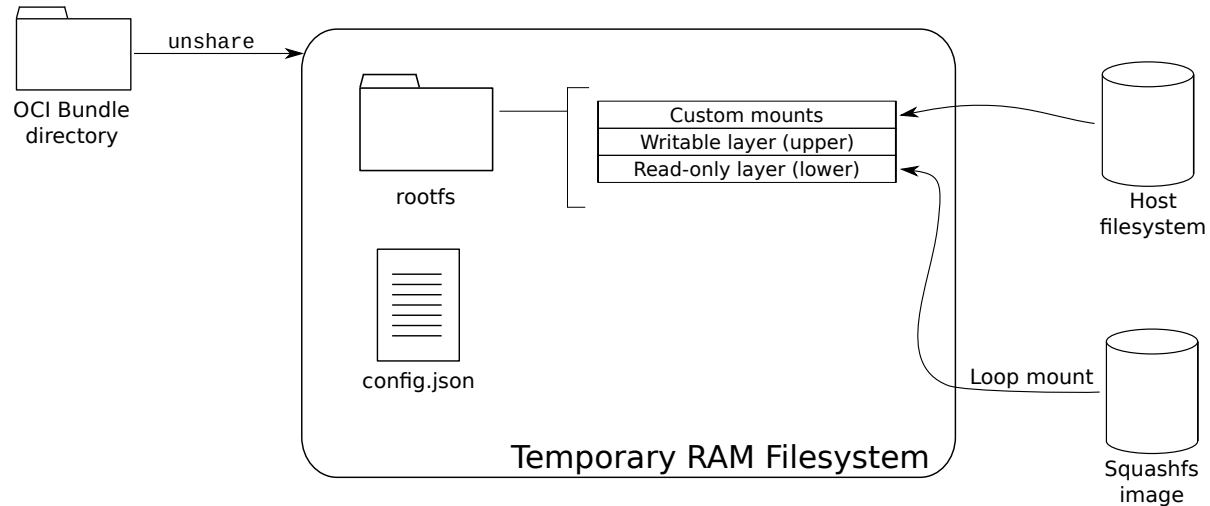


Fig. 3: OCI bundle setup in Sarus

config.json

The JSON configuration file of the OCI bundle is generated by combining data from the runtime execution context, command-line parameters and parameters coming from the image. We hereby highlight the most important details:

- The uid/gid of the user from the host system are assigned to the container process, regardless of the user settings in the original image. This is done to keep a consistent experience with the host system, especially regarding file ownership and access permissions.
- If the image specified an entrypoint or default arguments, these are honored, unless the user specifies an override through Sarus's command line. For more details, please refer to [this section](#) of the User Guide.
- The container environment variables are created by uniting variables from different sources: the host environment, the image, the Sarus configuration file and the command line. Unless explicitly re-defined in the Sarus configuration file or command line, variable values from the image have precedence. This ensures the container behaves as expected by its creators (e.g. in the case of `PATH`). Selected variables are also adapted by Sarus to suit system-specific extensions, like NVIDIA GPU support, native MPI support or container SSH connections.
- If a working directory is specified either in the image or from the Sarus CLI, the container process is started there. Otherwise, the process is started in the container's root directory. In this regard, Sarus shows the same behavior as Docker.
- The container process is configured to run with all Linux capabilities disabled², thus preventing it from acquiring new privileges by any means. This is done in the interest of security.
- A new mount namespace is setup for the container process.

² Linux divides the privileges traditionally associated with superuser into distinct units, known as **capabilities**.

- The container process runs in the PID namespace of the calling host process by default. A separate PID namespace for the container can be created if requested from the command line.
- Settings for OCI hooks are generated from the [OCI hook JSON configuration files](#) which are *configured* by the sysadmin.

Container launch

Once the bundle's rootfs directory and config.json file are in place, Sarus forks a process calling an *OCI-compliant* runtime, which in turn spawns and maintains the container process.

The OCI runtime is also in charge of executing the *OCI hooks* specified by Sarus. Hooks are an effective way of extending the functionality provided by the container runtime without additional development or maintenance effort on the runtime itself. In the context of HPC, hooks have shown the potential to augment containers based on open standards with native support for dedicated custom hardware, like accelerators or interconnect technologies, by letting vendors and third-party developers create ad hoc hook programs.

Once the container and OCI runtime processes terminate, Sarus itself concludes its workflow and exits.

1.3 Custom installation

1.3.1 Requirements

This page documents the requirements and dependencies needed by Sarus. It serves both as a general reference and as preliminary step to performing installations from source.

The Spack package manager is able to build or retrieve software dependencies on its own: as such, when using Spack, the only requirements to satisfy from this page are those related to the operating system.

Operating System

- Linux kernel ≥ 3.0
- [util-linux](#) ≥ 2.20 (these utilities are usually bundled with the Linux distribution itself; v2.20 was released in August 2011)
- The following kernel modules loaded:
 - loop
 - squashfs
 - overlayfs

These requirements are tracked by the [CI/check_host.sh](#) script.

System packages

The required packages for building Sarus are listed below for a selection of popular Linux distributions. Please note that, depending on the distribution, not all dependencies might be available through the system's package manager, or some dependencies might be provided in versions not supported by Sarus. Please follow the [manual installation](#) instructions for such dependencies.

Important: Please note that when building on ARM architecture, Boost version **1.77.x** or later is required.

OpenSUSE Leap 15.4

```
# Install packages
sudo zypper install -y gcc-c++ glibc-static wget which git gzip bzip2 tar \
    make autoconf automake squashfs cmake zlib-devel zlib-devel-static \
    runc tini-static skopeo umoci \
    libboost_filesystem1_75_0-devel \
    libboost_regex1_75_0-devel \
    libboost_program_options1_75_0-devel \
    python3 python3-pip python3-setuptools
sudo zypper clean --all
```

Rocky Linux 8

```
# Install packages
sudo dnf install -y dnf-plugins-core epel-release
sudo dnf config-manager --set-enabled powertools
sudo dnf install -y gcc gcc-c++ glibc-static libstdc++-static git make cmake autoconf \
    diffutils findutils wget which procps squashfs-tools zlib-devel zlib-static \
    boost-devel skopeo runc tini-static \
    python3 python3-pip python3-setuptools platform-python-devel
sudo dnf clean all
sudo rm -rf /var/cache/dnf

# The following dependencies are not provided via the system's package manager
# and should be installed manually:
# - Umoci
```

Rocky Linux 9

```
# Install packages
sudo dnf install -y dnf-plugins-core epel-release
sudo dnf config-manager --set-enabled crb
sudo dnf install -y gcc gcc-c++ glibc-static libstdc++-static git make cmake autoconf \
    diffutils findutils wget which procps squashfs-tools zlib-devel zlib-static \
    boost-devel skopeo runc \
    python3 python3-pip python3-setuptools
sudo dnf clean all
sudo rm -rf /var/cache/dnf

# The following dependencies are not provided via the system's package manager
# and should be installed manually:
# - Umoci
# - tini
```

Fedora 36

```
# Install packages
sudo dnf install -y gcc g++ glibc-static libstdc++-static git make cmake autoconf \
    diffutils findutils wget which procps squashfs-tools zlib-devel zlib-static \
    boost-devel skopeo tini-static \
    python3 python3-pip python3-setuptools
sudo dnf clean all
sudo rm -rf /var/cache/dnf

# The following dependencies are not provided via the system's package manager
# and should be installed manually:
# - Runc version 1.1.6
# - Umoci
```

Debian 11

```
# Install packages
sudo apt-get update --fix-missing
sudo apt-get install -y --no-install-recommends build-essential
sudo DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends kmod curl git \
    autoconf automake gnupg squashfs-tools cmake wget \
    zlib1g-dev runc tini skopeo umoci \
    libboost-dev \
    libboost-program-options-dev \
    libboost-filesystem-dev \
    libboost-regex-dev \
    python3 python3-pip python3-setuptools
sudo apt-get clean
sudo rm -rf /var/lib/apt/lists/*
```

Ubuntu 22.04

```
# Install packages
sudo apt-get update --fix-missing
sudo apt-get install -y --no-install-recommends build-essential
sudo DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends kmod curl git \
    autoconf automake gnupg squashfs-tools cmake wget \
    zlib1g-dev tini skopeo umoci \
    libboost-dev \
    libboost-program-options-dev \
    libboost-filesystem-dev \
    libboost-regex-dev \
    python3 python3-pip python3-setuptools
sudo apt-get clean
sudo rm -rf /var/lib/apt/lists/*

# The following dependencies are not provided via the system's package manager
# and should be installed manually:
# - Runc version 1.1.6
```


Manual installation

This section provides instructions to install some notable Sarus dependencies which might not be available through the system's package manager, or might be provided in a version not supported by Sarus.

Boost libraries

Boost libraries are required to be version **1.60.x** or later. The recommended version, which is used routinely for build integration and testing, is **1.77.x**.

It is especially important to note that when building on ARM architecture, Boost version **1.77.x** or later is required.

Note: The following instructions will default to `/usr/local` as the installation prefix. To install to a specific location, use the `--prefix` option of the `b2` builder/installer tool.

```
pwd_bak=$PWD

# Install boost
cd /tmp && \
  mkdir -p boost/1_77_0 && cd boost/1_77_0 && \
  wget https://downloads.sourceforge.net/project/boost/boost/1.77.0/boost_1_77_0.tar.
↪bz2 && \
  tar xf boost_1_77_0.tar.bz2 && \
  mv boost_1_77_0 src && cd src && \
  ./bootstrap.sh && \
  sudo ./b2 -j$(nproc) \
    --with-filesystem \
    --with-regex \
    --with-program_options \
    install && \
  cd ${pwd_bak} && \
  rm -rf /tmp/boost
```

Skopeo

Skopeo is used to acquire images and their metadata from a variety of sources. Versions **1.7.0 or later** are recommended for improved performance when pulling or loading images.

Up-to-date instructions about building Skopeo from source are available on the project's [GitHub repository](#).

Umoci

Umoci is used to unpack OCI images' filesystem contents before converting them into the SquashFS format:

```
pwd_bak=$PWD

# Install umoci
cd /tmp && \
  wget -O umoci https://github.com/opencontainers/umoci/releases/download/v0.4.7/umoci.
↪amd64 && \
```

(continues on next page)

(continued from previous page)

```
chmod 755 umoci && \  
sudo mv umoci /usr/local/bin/ && \  
sudo chown root:root /usr/local/bin/umoci && \  
cd ${pwd_bak}
```

OCI-compliant runtime

Sarus internally relies on an OCI-compliant runtime to spawn a container.

Here we will provide some indications to install **runc**, the reference implementation from the Open Container Initiative. The recommended version is **v1.1.6**.

Note: runc versions from 1.1.0 to 1.1.2 have a bug which causes an error when Sarus attempts to propagate the PMI2 interface into containers. The bug has been fixed in runc 1.1.3.

The simplest solution is to download a pre-built binary release from the project's GitHub page:

```
pwd_bak=$PWD  
  
# Install runc  
cd /tmp && \  
  wget -O runc.amd64 https://github.com/opencontainers/runc/releases/download/v1.1.6/  
↪runc.amd64 && \  
  chmod 755 runc.amd64 && \  
  sudo mv runc.amd64 /usr/local/bin/ && \  
  sudo chown root:root /usr/local/bin/runc.amd64 && \  
  cd ${pwd_bak}
```

Alternatively, you can follow the instructions to [build from source](#), which allows more fine-grained control over runc's features, including security options.

Init process

Sarus can start an init process within containers in order to reap zombie processes and allow container applications to receive signals.

Here we will provide some indications to install **tini**, a very lightweight init process which is also used by Docker. The recommended version is **v0.19.0**.

The simplest solution is to download a pre-built binary release from the project's GitHub page:

```
pwd_bak=$PWD  
  
# Install tini  
cd /tmp && \  
  wget -O tini-static-amd64 https://github.com/krallin/tini/releases/download/v0.19.0/  
↪tini-static-amd64 && \  
  chmod 755 tini-static-amd64 && \  
  sudo mv tini-static-amd64 /usr/local/bin/ && \  
  sudo chown root:root /usr/local/bin/tini-static-amd64 && \  
  cd ${pwd_bak}
```

Alternatively, you can follow the instructions to [build from source](#).

Python packages for integration tests

The following Python 3 packages are required if you are interested to also run the integration tests:

```
$ pip3 install setuptools
$ pip3 install pytest gcovr pexpect
```

1.3.2 Installation

Sarus can be installed in three alternative ways:

- *Using the standalone Sarus archive*
- *Using the Spack package manager*
- *Installing from source*

This page documents installations using Spack or from source. These installation methods allow more control and customization over how Sarus is built.

When installing from source, make sure that the [required dependencies](#) are available on the system.

Whichever procedure you choose, please also read the page about [Post-installation actions](#) to ensure everything is set to run Sarus.

Installing with Spack

Sarus provides [Spack](#) packages for itself and its dependencies which are not yet covered by Spack's builtin repository. Installing with Spack is convenient because detection and installation of dependencies are handled automatically.

As explained in the [execution requirements](#), Sarus must be installed as a root-owned SUID binary in a directory hierarchy which belongs to the root user at all levels. The straightforward way to achieve this is to use a root-owned Spack and run the installation procedure with super-user privileges. An alternative procedure for test/development installations with a non-root Spack is discussed in [this subsection](#).

Note: A static version of glibc is required to build the Dropbear software for the SSH hook. On Rocky Linux, Fedora and OpenSUSE Leap systems, such library is provided separately from common development tools:

```
# Rocky Linux 8
dnf install -y dnf-plugins-core epel-release
dnf config-manager --set-enabled powertools
dnf install -y gcc gcc-c++ glibc-static

# Rocky Linux 9
dnf install -y dnf-plugins-core epel-release
dnf config-manager --set-enabled crb
dnf install -y gcc gcc-c++ glibc-static

# Fedora
dnf install glibc-static
```

(continues on next page)

(continued from previous page)

```
# OpenSUSE Leap
zypper install glibc-devel-static
```

The installation procedure with Spack follows below. If you are not performing these actions as the root user, prefix each spack command with `sudo`:

```
# Setup Spack bash integration (if you haven't already done so)
. ${SPACK_ROOT}/share/spack/setup-env.sh

# Create a local Spack repository for Sarus-specific dependencies
export SPACK_LOCAL_REPO=${SPACK_ROOT}/var/spack/repos/cscs
spack repo create ${SPACK_LOCAL_REPO}
spack repo add ${SPACK_LOCAL_REPO}

# Import Spack package for Sarus
cp -r <Sarus project root dir>/spack/packages/* ${SPACK_LOCAL_REPO}/packages/

# Install Sarus
spack install --verbose sarus
```

By default, the latest tagged release will be installed. To get the bleeding edge, use the `@develop` version specifier.

The Spack package for Sarus supports the following [variants](#) to customize the installation:

- `ssh`: Build and install the SSH hook and custom SSH software to enable connections inside containers [True].
- `configure_installation`: Run the script to setup a starting Sarus configuration as part of the installation phase. Running the script requires super-user privileges [True].
- `unit_tests`: Build unit test executables in the build directory. Also downloads and builds internally the CppUTest framework [False].

For example, in order to perform a quick installation without SSH we could use:

```
spack install --verbose sarus ssh=False
```

Using Spack as a non-root user (for test and development only)

By default, the Spack package for Sarus will run a [configuration script](#) as part of the installation phase to setup a minimal working configuration. The script needs to run with root privileges, which in turn means the `spack install` command must be issued with root powers.

The `configure_installation` variant of the Spack package allows to control the execution of the script. By negating the variant, Sarus can be installed as a user different than root. It is then necessary to access the installation path and run the configuration script directly. While doing so still requires `sudo`, it can be performed with a user's personal Spack software and the amount of root-owned files in the Sarus installation is kept to a minimum.

Installing as non-root likely means that Sarus is located in a directory hierarchy which is not root-owned all the way up to `/`. This is not allowed by the program's [security checks](#), so they have to be disabled.

Important: The ability to disable security checks is only meant as a convenience feature when rapidly iterating over test and development installations. It is strongly recommended to keep the checks enabled for production deployments.

Sarus can then be loaded and used normally. The procedure is summed up in the following example:

```
# Setup Spack bash integration (if you haven't already done so)
. ${SPACK_ROOT}/share/spack/setup-env.sh

# Create a local Spack repository for Sarus-specific dependencies
export SPACK_LOCAL_REPO=${SPACK_ROOT}/var/spack/repos/cscs
spack repo create ${SPACK_LOCAL_REPO}
spack repo add ${SPACK_LOCAL_REPO}

# Import Spack package for Sarus
cp -r <Sarus project root dir>/spack/packages/* ${SPACK_LOCAL_REPO}/packages/

# Install Sarus skipping the configure_installation script
$ spack install --verbose sarus@develop ~configure_installation

# Use 'spack find' to get the installation directory
$ spack find -p sarus@develop
==> 1 installed package
-- linux-ubuntu18.04-sandybridge / gcc@7.5.0 -----
sarus@develop /home/ubuntu/spack/opt/spack/linux-ubuntu18.04-sandybridge/gcc-7.5.0/
↳sarus-develop-dy4f6tlhx3vwf6zmqityvgdhpnc6clg5

# cd into the installation directory
$ cd /home/ubuntu/spack/opt/spack/linux-ubuntu18.04-sandybridge/gcc-7.5.0/sarus-develop-
↳dy4f6tlhx3vwf6zmqityvgdhpnc6clg5

# Run configure_installation script
$ sudo ./configure_installation.sh

# Disable security checks
$ sudo sed -i -e 's/"securityChecks": true/"securityChecks": false/' etc/sarus.json

# Return to home folder and test Sarus
$ cd
$ spack load sarus@develop
$ sarus pull alpine
[...]
$ sarus run alpine cat /etc/os-release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.11.5
PRETTY_NAME="Alpine Linux v3.11"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```

Installing from source

Build and install

Get Sarus source code:

```
git clone --recursive git@github.com:eth-cscs/sarus.git
cd sarus
```

Create a new folder `${build_dir}` to build Sarus from source. e.g. build-Release:

```
mkdir -p ${build_dir} && cd ${build_dir}
```

Configure and build (in this example `${build_type}` = Release and `${toolchain_file}` = gcc.cmake:

```
cmake -DCMAKE_TOOLCHAIN_FILE=${build_dir}/../cmake/toolchain_files/${toolchain_file} \
↳ \
    -DCMAKE_BUILD_TYPE=${build_type} \
    -DCMAKE_INSTALL_PREFIX=${install_dir} \
    ..
make -j$(nproc)
```

Note: CMake should automatically find the dependencies (include directories, shared objects, and binaries). However, should CMake not find a dependency, its location can be manually specified through the command line. E.g.:

```
cmake -DCMAKE_TOOLCHAIN_FILE=../cmake/toolchain_files/gcc.cmake \
    -DCMAKE_INSTALL_PREFIX=${prefix_dir} \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_PREFIX_PATH="<boost install dir>" \
    ..
```

Note: Old versions of CMake might have problems finding Boost 1.65.0. We recommend to use at least CMake 3.10 in order to avoid compatibility issues.

Below are listed the Sarus-specific options that can be passed to CMake in order to customize your build:

- `CMAKE_INSTALL_PREFIX`: installation directory of Sarus [/usr/local].
- `ENABLE_SSH`: build and install the SSH hook and custom SSH software to enable connections inside containers [TRUE].
- `ENABLE_UNIT_TESTS`: build unit tests. Also downloads and builds internally the CppUTest framework [TRUE].
- `ENABLE_TESTS_WITH_VALGRIND`: run each unit test through valgrind [FALSE].

Copy files to the installation directory:

```
sudo make install
```

Create the directory in which Sarus will create the OCI bundle for containers. The location of this directory is configurable at any time, as described in the next section. As an example, taking default values:

```
sudo mkdir -p ${install_dir}/var/OCIBundleDir
```

Finalize the installation

To complete the installation, run the *configuration script* located in the installation path. This script needs to run with root privileges in order to set Sarus as a root-owned SUID program:

```
sudo cp /usr/local/bin/tini-static-amd64 ${install_dir}/bin || true
sudo cp /usr/local/bin/runc.amd64 ${install_dir}/bin || true

sudo ${install_dir}/configure_installation.sh
export PATH=${install_dir}/bin:${PATH}
```

Note: The configuration script will create a minimal working configuration. For enabling additional features, please refer to the *Configuration file reference*.

As suggested by the output of the script, you should also add Sarus to your PATH. Add a line like `export PATH=/opt/sarus/bin:${PATH}` to your `.bashrc` in order to make the addition persistent.

1.3.3 Post-installation actions

Review permissions required by Sarus

During execution

- Sarus must run as a root-owned SUID executable and be able to achieve full root privileges to perform mounts and create namespaces.
- Write/read permissions to the Sarus's centralized repository. The system administrator can configure the repository's location through the `centralizedRepositoryDir` entry in `sarus.json`.
- Write/read permissions to the users' local image repositories. The system administrator can configure the repositories location through the `localRepositoryBaseDir` entry in `sarus.json`.

Security related

Because of the considerable power granted by the requirements above, as a security measure Sarus will check that critical files and directories opened during privileged execution meet the following restrictions:

- They are owned by root.
- They are writable only by the owner.
- All their parent directories (up to the root path) are owned by root.
- All their parent directories (up to the root path) are writable only by the owner (no write permissions to group users or other users).

The files checked for the security conditions are:

- `sarus.json` in Sarus's configuration directory `<sarus install prefix>/etc`.
- `sarus.schema.json` in Sarus's configuration directory `<sarus install prefix>/etc`.

- The init binary pointed by `initPath` in `sarus.json`.
- The OCI-compliant runtime pointed by `runcPath` in `sarus.json`.
- All the OCI hooks JSON files in the `hooksDir` directory specified in `sarus.json`.
- All the executables specified in the OCI hooks JSON files.

For directories, the conditions apply recursively for all their contents. The checked directories are:

- The directory where Sarus will create the OCI bundle. This location can be configured through the `OCIBundleDir` entry in `sarus.json`.
- If the *SSH Hook* is installed, the directory of the custom SSH software.

Most security checks can be disabled through the *corresponding parameter* in the Sarus configuration file. Checks on `sarus.json` and `sarus.schema.json` will always be performed, regardless of the parameter value.

Important: The ability to disable security checks is only meant as a convenience feature when rapidly iterating over test and development installations. It is strongly recommended to keep the checks enabled for production deployments.

Load required kernel modules

If the kernel modules listed in *Requirements* are not loaded automatically by the system, remember to load them manually:

```
sudo modprobe loop
sudo modprobe squashfs
sudo modprobe overlay
```

Automatic update of Sarus' passwd cache

When executing the *configure_installation script*, the `passwd` and `group` information are copied and cached into `<sarus install prefix>/etc/passwd` and `<sarus install prefix>/etc/group` respectively. The cache allows to bypass the host's `passwd/group` database, e.g. LDAP, which could be tricky to configure and access from the container. However, since the cache is created/updated only once at installation time, it can quickly get out-of-sync with the actual `passwd/group` information of the system. A possible solution is to periodically run a cron job to refresh the cache. E.g. a cron job and a script like the ones below would do:

```
$ crontab -l
5 0 * * * update_sarus_user.sh
```

```
$ cat update_sarus_user.sh

#!/bin/bash

/usr/bin/getent passwd > <sarus install prefix>/etc/passwd
/usr/bin/getent group > <sarus install prefix>/etc/group
```


1.4 Configuration

1.4.1 Basic configuration

At run time, the configuration parameters of Sarus are read from a file called *sarus.json*, which is expected to be a valid *JSON document* located in `<installation path>/etc`. Given the *privileges* that Sarus requires in order to exercise its functions, the configuration file must satisfy specific *security requirements* regarding its location and ownership.

Using the `configure_installation` script

The most straightforward way to generate a basic working configuration for a Sarus installation is by using the `configure_installation.sh` script, which is found in the top installation directory.

This script will set the proper permissions of the Sarus binary, create the initial caches for `passwd` and `group` databases and fill the necessary parameters of *sarus.json*.

To accomplish its tasks, the script has to be run as root and has to be located in the top directory of a valid Sarus installation.

The following is an example script execution (assuming Sarus has been installed in `/opt/sarus`) with the corresponding output:

```
$ sudo /opt/sarus/configure_installation.sh
Setting Sarus as SUID root
Successfully set Sarus as SUID root
Creating cached passwd database
Successfully created cached passwd database
Creating cached group database
Successfully created cached group database
Setting ownership of etc/sarus.schema.json
Successfully set ownership of etc/sarus.schema.json
Configuring etc/sarus.json
Found skopeo: /usr/bin/skopeo
Found umoci: /usr/bin/umoci
Found mksquashfs: /usr/bin/mksquashfs
Found init program: /usr/bin/tini
Found runc: /usr/local/bin/runc
Setting local repository base directory to: /home
Setting centralized repository directory to: /var/sarus/centralized_repository
Successfully configured etc/sarus.json.
Configuring etc/hooks.d/07-ssh-hook.json
Configuring etc/hooks.d/09-slurm-global-sync-hook.json
To execute sarus commands run first:
export PATH=/opt/sarus/default/bin:${PATH}
To persist that for future sessions, consider adding the previous line to your .bashrc
or equivalent file
```

Custom values for the *local repositories base directory* and the *centralized image repository* can be specified by setting the `SARUS_LOCAL_REPO_BASE_DIR` and `SARUS_CENTRALIZED_REPO_DIR` environment variables, respectively. For example:

```
$ sudo SARUS_LOCAL_REPO_BASE_DIR=/users SARUS_CENTRALIZED_REPO_DIR=/sarus_centralized_
repository /opt/sarus/configure_installation.sh
```

The configuration script is the recommended way to finalize any installation of Sarus, regardless of the installation method chosen.

Please note that `configure_installation.sh` will only create a baseline configuration. To enable more advanced features of Sarus, `sarus.json` should be expanded further. Please refer to the [Configuration file reference](#) in order to do so.

The script can also be used on a Sarus installation which has already been configured. In this case, the existing configuration will be backed up in `<installation path>/etc/sarus.json.bak` and a new `sarus.json` file will be generated.

1.4.2 Configuration file reference

Configuration file entries

securityChecks (bool, REQUIRED)

Enable/disable runtime security checks to verify that security critical files are not tamperable by non-root users. Disabling this may be convenient when rapidly iterating over test and development installations. It is strongly recommended to keep these checks enabled for production deployments. Refer to the section about [security requirements](#) for more details about these checks.

Recommended value: `true`

OCIBundleDir (string, REQUIRED)

Absolute path to where Sarus will generate an OCI bundle, from which a container will be created. An OCI bundle is composed by a `config.json` configuration file and a directory which will form the root filesystem of the container. The `OCIBundleDir` directory must satisfy the [security requirements](#) for critical files and directories.

Recommended value: `/var/sarus/OCIBundleDir`

rootfsFolder (string, REQUIRED)

The name Sarus will assign to the directory inside the OCI bundle; this directory will become the root filesystem of the container. The value must be a valid filename and not a path (that is, `/` characters are not allowed).

Recommended value: `rootfs`

prefixDir (string, REQUIRED)

Absolute path to the base directory where Sarus has been installed. This path is used to find all needed Sarus-specific utilities.

Example value: `/opt/sarus/<version>`

hooksDir (string, OPTIONAL)

Absolute path to the directory containing the [OCI hook JSON configuration files](#). See [Configuring Sarus for OCI hooks](#). Sarus will process the JSON files in *hooksDir* and generate the configuration file of the OCI bundle accordingly. The hooks will effectively be called by the OCI-compliant runtime specified by *runcPath*.

Example value: `/opt/sarus/<version>/etc/hooks.d`

tempDir (string, REQUIRED)

Absolute path to the directory where Sarus will create temporary files and folders for various purposes, like unpacking filesystem layers when pulling and loading images.

Recommended value: `/tmp`

localRepositoryBaseDir (string, REQUIRED)

Absolute base path to individual user directories, where Sarus will create (if necessary) and access local image repositories. The repositories will be located in `<localRepositoryBaseDir>/<user name>/sarus`.

Example value: `/home`

centralizedRepositoryDir (string, OPTIONAL)

Absolute path to where Sarus will create (if necessary) and access the system-wide centralized image repository. This repository is intended to satisfy use cases where an image can only be broadcasted to the users, without allowing them to pull the image directly (e.g. images that cannot be redistributed due to licensing agreements).

The centralized repository is meant to be read-only for regular users, and its contents should be modifiable only by the system administrators.

Example value: `/var/sarus/centralized_repository`

skopeoPath (string, REQUIRED)

Absolute path to a trusted `skopeo` binary, which will be used to pull images from container registries or load them from local files.

umociPath (string, REQUIRED)

Absolute path to a trusted `umoci` binary, which will be used to unpack image contents before converting them to SquashFS format.

mksquashfsPath (string, REQUIRED)

Absolute path to trusted `mksquashfs` binary. This executable must satisfy the *security requirements* for critical files and directories.

mksquashfsOptions (string, OPTIONAL)

String with whitespace-separated command line options which will be passed to the binary at *mksquashfsPath* when creating SquashFS-based images. This parameter can be used, for example, to control the compression algorithm, the compression level, or the number of processors used by `mksquashfs`. Please refer to the help message or manpage of your `mksquashfs` installation for the exact list of options available. The following online manpage can serve as a general reference: [mksquashfs\(1\)](#).

initPath (string, REQUIRED)

Absolute path to trusted init process static binary which will launch the user-specified applications within the container when the `--init` option to **sarus run** is used. This executable must satisfy the *security requirements* for critical files and directories.

By default, within the container Sarus only executes the user-specified application, which is assigned PID 1. The PID 1 process has unique features in Linux: most notably, the process will ignore signals by default and zombie processes will not be reaped inside the container (see [1] , [2] for further reference).

Running the container application through an init system provides a solution for signaling container applications or reaping processes of long-running containers.

The standalone package of Sarus uses `tini` as its default init process.

Warning: Some HPC applications may be subject to performance losses when run with an init process. Our internal benchmarking tests with `tini` showed overheads of up to 2%.

runcPath (string, REQUIRED)

Absolute path to trusted OCI-compliant runtime binary, which will be used by Sarus to spawn the actual low-level container process. This executable must satisfy the *security requirements* for critical files and directories.

ramFilesystemType (string, REQUIRED)

The type of temporary filesystem Sarus will use for setting up the base VFS layer for the container. Must be either `tmpfs` or `ramfs`.

A filesystem of this type is created inside a dedicated mount namespace unshared by Sarus for each container. The temporary filesystem thus generated will be used as the location of the OCI bundle, including the subsequent mounts (loop, overlay and, if requested, bind) that will form the container's rootfs. The in-memory and temporary nature of this filesystem helps with performance and complete cleanup of all container resources once the Sarus process exits.

Warning: When running on Cray Compute Nodes (CLE 5.2 and 6.0), `tmpfs` will not work and `ramfs` has to be used instead.

Recommended value: `tmpfs`

siteMounts (array, OPTIONAL)

List of JSON objects defining filesystem mounts that will be automatically performed from the host system into the container bundle. This is typically meant to make network filesystems accessible within the container but could be used to allow certain other facilities.

Each object in the list must define the following fields:

- **type** (string): The type of the mount. Currently, only `bind` (for bind-mounts) is supported.
- **source** (string): Absolute path to the host file/directory that will be mounted into the container.
- **destination** (string): Absolute path to where the filesystem will be made available inside the container. If the directory does not exist, it will be created.

Bind mounts

In addition to **type**, **source** and **destination**, bind mounts can optionally add the following field:

- **flags** (object, OPTIONAL): Object defining the flags for the bind mount. Can have the following fields:
 - *readonly* (string, empty value expected): Mount will be performed as read-only.

By default, bind mounts will always be of `recursive private` flavor. Refer to the [Linux docs](#) for more details.

General remarks

siteMounts are not subject to the limitations of user mounts requested through the CLI. More specifically, these mounts:

- Can specify any path in the host system as source
- Can specify any path in the container as destination

It is not recommended to bind things under `/usr` or other common critical paths within containers.

It is OK to perform this under `/var` or `/opt` or a novel path that your site maintains (e.g. `/scratch`).

siteDevices (array, OPTIONAL)

List of JSON object defining device files which will be automatically mounted from the host filesystem into the container bundle. The devices will also be whitelisted in the container's device cgroup (Sarus disables access to custom device files by default).

Each object in the list supports the following fields:

- **source** (string, REQUIRED): Absolute path to the device file on the host.
- **destination** (string, OPTIONAL): Absolute path to the desired path for the device in the container. In the absence of this field, the device will be bind mounted at the same path within the container.
- **access** (string, OPTIONAL): Flags defining the the type of access the device will be whitelisted for. Must be a combination of the characters `rwm`, standing for *read*, *write* and *mknod* access respectively; the characters may come in any order, but must not be repeated. Permissions default to `rwm` if this field is not present.

As highlighted in the related [section of the User Guide](#), Sarus cannot grant more access permissions than those allowed in the host device cgroup.

environment (object, OPTIONAL)

JSON object defining operations to be performed on the environment of the container process. Can have four optional fields:

- **set** (object): JSON object with fields having string values. The fields represent the key-value pairs of environment variables. The variables defined here will be set in the container environment, possibly replacing any previously existing variables with the same names. This can be useful to inform users applications and scripts that they are running inside a Sarus container.
- **prepend** (object): JSON object with fields having string values. The fields represent the key-value pairs of environment variables. The values will be prepended to the corresponding variables in the container, using a colon as separator. This can be used, for example, to prepend site-specific locations to PATH.
- **append** (object): JSON object with fields having string values. The fields represent the key-value pairs of environment variables. The values will be appended to the corresponding variables in the container, using a colon as separator. This can be used, for example, to append site-specific locations to PATH.
- **unset** (array): List of strings representing environment variable names. Variables with the corresponding names will be unset in the container.

userMounts (object, OPTIONAL)

Normal users have to possibility of requesting custom paths available to them in the host environment to be mapped to another path inside the container. This is achieved through the `--mount` option of `sarus run`. The `userMounts` object offers the means to set limitations for this feature through two arrays:

- **notAllowedPrefixesOfPath**: list of strings representing starting paths. The user will not be able to enter these paths or any path under them as a mount destination. Default set to `["/etc", "/var", "/opt/sarus"]`.
- **notAllowedPaths**: list of strings representing exact paths. The user will not be able to enter these paths as a mount destination. Default set to `["/opt"]`.

Both these fields and `userMounts` itself are optional: remove them to lift any restriction.

These limitations apply only to mounts requested through the command line; Mounts entered through `siteMounts` are not affected by them.

seccompProfile (string, OPTIONAL)

Absolute path to a file defining a seccomp profile in accordance with the [JSON format specified by the OCI Runtime Specification](#). This profile will be applied to the container process by the OCI runtime.

Seccomp (short for "SECure COMputing mode") is a Linux kernel feature allowing to filter the system calls which are performed by a given process. It is intended to minimize the kernel surface exposed to an application.

For reference, you may refer to the default seccomp profiles used by [Docker](#), [Singularity CE](#) or [Podman](#).

apparmorProfile (string, OPTIONAL)

Name of the [AppArmor](#) profile which will be applied to the container process by the OCI runtime. The profile must already be loaded in the kernel and listed under `/sys/kernel/security/apparmor/profiles`.

selinuxLabel (string, OPTIONAL)

[SELinux](#) label which will be applied to the container process by the OCI runtime.

selinuxMountLabel (string, OPTIONAL)

[SELinux](#) label which will be applied to the mounts performed by the OCI runtime into the container.

containersPolicy (object, OPTIONAL)

The `containers-policy.json(5)` file (formally called the "signature verification policy file") is used by Skopeo and other container tools to define a set of policy requirements (for example trusted keys) which have to be satisfied in order to qualify a container image, or individual signatures of that image, as valid and secure to download.

By default, a user-specific policy is read from `${HOME}/.config/containers/policy.json`; if such file does not exist, the system-wide `/etc/containers/policy.json` is used instead. This system-wide file is usually provided by the [containers-common](#) package.

The `containersPolicy` object defines fallback and enforcement options for the policy file and supports the following fields:

- **path** (string, REQUIRED): Absolute path to a fallback [containers-policy.json\(5\)](#) file, which will be passed by Sarus to Skopeo in case neither the user-specific nor the system-wide default policy files exist. This allows to use a policy also on systems which don't have the default files present on all nodes. If no default file exists and the `containersPolicy` parameter is not defined, Sarus throws an error.
- **enforce** (bool, OPTIONAL): If true, always use the policy file at `containersPolicy/path`, even if any default file exists. This allows to have a Sarus-specific policy different from the one(s) used by other tools on the system.

Important: Sarus installations come with a policy file at `<prefixDir>/etc/policy.json`, which is set as the starting value of `containersPolicy/path`. This policy file is very permissive and is in line with the defaults provided by package managers for the most popular Linux distributions. It is intended only as a starting point in case a system does not feature default policy files.

containersRegistries.dPath (string, OPTIONAL)

Absolute path to a [containers-registries.d\(5\)](#) directory for registries configurations. If defined, this directory will be used by Skopeo instead of the default `${HOME}/.config/containers/registries.d` or `/etc/containers/registries.d` directories.

defaultMPIType (string, OPTIONAL)

Default value which will be set in the `com.hooks.mpi.type=<value>` bundle annotation when the `--mpi` option of **sarus run** is used without an accompanying `--mpi-type` option providing an explicit MPI type.

In practice, this parameter identifies the default MPI-related hook when *multiple hooks are configured* with a `com.hooks.mpi.type=<value>` condition, and enables users to just use `--mpi` to access the default MPI support mechanism configured for the system. This parameter also provides compatibility with workflows which do not use the `--mpi-type` option for reasons of portability or legacy.

If this parameter is not defined and the command line does not feature the `--mpi-type` option, the `com.hooks.mpi.type=<value>` annotation is not set. This could fit the case of a system where only a single MPI hook is configured.

Refer to [this section](#) for more details about the interaction between command line options, annotations, and hook selection.

repositoryMetadataLockTimings (object, OPTIONAL)

To prevent race conditions when accessing metadata files of local image repositories, Sarus implements a mechanism based on a lock file.

The `repositoryMetadataLockTimings` JSON object allows to control the behavior of Sarus while it is waiting to acquire the lock on the metadata file. This object can have the following optional fields:

- `timeoutMs` (integer): Maximum amount of time (in milliseconds) for which Sarus attempts to acquire the lock. If Sarus is still waiting after this time has elapsed, the program times out and exits with an error. The value must be a positive integer. If the parameter is not defined, defaults to 60000 ms (60 seconds).
- `warningMs`: While waiting to acquire the lock, Sarus prints regular warning messages to inform of its status (i.e. access to the metadata file is taking longer but the program is not stalled for another reason) until the timeout is reached or acquisition of the lock is successful. This parameter determines the time interval (in milliseconds) between such warning messages. The value must be a positive integer. If the parameter is not defined, defaults to 10000 ms (10 seconds).

enablePMIxv3Support (bool, OPTIONAL) (experimental)

Enable experimental support for propagating a PMIx v3 context into the container. If support is enabled and use of PMIx is detected during container creation, Sarus will perform additional bind mounts and ensure specific environment variables used by the PMIx v3 interface are set.

Default value: False

Example configuration file

```
{
  "securityChecks": true,
  "OCIBundleDir": "/var/sarus/OCIBundleDir",
  "rootfsFolder": "rootfs",
  "prefixDir": "/opt/sarus/1.6.0",
  "hooksDir": "/opt/sarus/1.6.0/etc/hooks.d",
  "tempDir": "/tmp",
  "localRepositoryBaseDir": "/home",
  "centralizedRepositoryDir": "/var/sarus/centralized_repository",
```

(continues on next page)

(continued from previous page)

```

"skopeoPath": "/usr/bin/skopeo",
"umociPath": "/usr/bin/umoci",
"mksquashfsPath": "/usr/sbin/mksquashfs",
"mksquashfsOptions": "-comp gzip -processors 4 -Xcompression-level 6",
"runcPath": "/usr/local/sbin/runc.amd64",
"ramFilesystemType": "tmpfs",
"siteMounts": [
  {
    "type": "bind",
    "source": "/home",
    "destination": "/home",
    "flags": {}
  }
],
"siteDevices": [
  {
    "source": "/dev/fuse",
    "access": "rw"
  }
],
"environment": {
  "set": {
    "VAR_TO_SET_IN_CONTAINER": "value"
  },
  "prepend": {
    "VAR_WITH_LIST_OF_PATHS_IN_CONTAINER": "/path/to/prepend"
  },
  "append": {
    "VAR_WITH_LIST_OF_PATHS_IN_CONTAINER": "/path/to/append"
  },
  "unset": [
    "VAR_TO_UNSET_IN_CONTAINER_0",
    "VAR_TO_UNSET_IN_CONTAINER_1"
  ]
},
"userMounts": {
  "notAllowedPrefixesOfPath": [
    "/etc",
    "/var",
    "/opt/sarus"
  ],
  "notAllowedPaths": [
    "/opt"
  ]
},
"seccompProfile": "/opt/sarus/1.6.0/etc/seccomp/default.json",
"apparmorProfile": "sarus-default",
"selinuxLabel": "system_u:system_r:svirt_sarus_t:s0:c124,c675",
"selinuxMountLabel": "system_u:object_r:svirt_sarus_file_t:s0:c715,c811"
"containersPolicy": {
  "path": "/opt/sarus/1.6.0/etc/policy.json",
  "enforce": false
}

```

(continues on next page)

(continued from previous page)

```
},
"containersRegistries.dPath": "/opt/sarus/1.6.0/etc/registries.d"
"defaultMPIType": "mpich",
"repositoryMetadataLockTimings": {
  "timeoutMs": 120000,
  "warningMs": 15000
}
}
```

1.4.3 Configuring Sarus for OCI hooks

In order to extend the core functionality provided by a container runtime, the OCI Runtime Specification allows for components to be hooked into the container's lifecycle, performing custom actions. These [OCI hooks](#) are especially amenable to HPC use cases, where the dedicated hardware and highly-tuned software adopted by high-performance systems are in contrast with the platform-agnostic nature of software containers. Effectively, OCI hooks provide solutions for the container runtime to allow access to system-specific features and specialized resources within container instances.

To enable the use of hooks in Sarus, the *hooksDir* directory specified in *sarus.json* must be populated with [OCI hook JSON configuration files](#). Sarus currently supports the OCI hook schema version 1.0.0.

Sarus parses the JSON files in *hooksDir* (subdirectories excluded) and, if the *when* conditions specified in a JSON are all true, the corresponding OCI hook is inserted into the OCI bundle's configuration file that is generated by Sarus.

The hooks will effectively be called by the OCI-compliant runtime specified with the *runcPath* parameter in *sarus.json*.

The hooks are inserted into the OCI bundle's configuration file and executed by the OCI-compliant runtime in the order obtained by lexicographically sorting the JSON file names.

All the hooks configured in the *hooksDir* directory must satisfy the [security requirements](#) for critical files and directories.

Hooks use cases

In the following pages, we will provide guidance on how to enable hooks for specific use cases:

Native MPI hook (MPICH-based)

Sarus's source code includes a hook able to import native MPICH-based MPI implementations inside the container. This is useful in case the host system features a vendor-specific or high-performance MPI stack based on MPICH (e.g. Intel MPI, Cray MPI, MVAPICH) which is required to fully leverage a high-speed interconnect.

When activated, the hook will enter the mount namespace of the container, search for dynamically-linkable MPI libraries and replace them with functional equivalents bind-mounted from the host system.

In order for the replacements to work seamlessly, the hook will check that the host and container MPI implementations are ABI-compatible according to the standards defined by the [MPICH ABI Compatibility Initiative](#). The Initiative is supported by several MPICH-based implementations, among which MVAPICH, Intel MPI, and Cray MPT. ABI compatibility and its implications are further discussed [here](#).

Hook installation

The hook is written in C++ and it will be compiled when building Sarus without the need of additional dependencies. Sarus' installation scripts will also automatically install the hook in the `$CMAKE_INSTALL_PREFIX/bin` directory. In short, no specific action is required to install the MPI hook.

Hook configuration

The program is meant to be run as a **prestart** hook and does not accept arguments, but its actions are controlled through a few environment variables:

- **LDCONFIG_PATH**: Absolute path to a trusted `ldconfig` program **on the host**.
- **MPI_LIBS**: Colon separated list of full paths to the host's libraries that will substitute the container's libraries. The ABI compatibility check is performed by comparing the version numbers specified in the libraries' file names as follows:
 - The major numbers (first from the left) must be equal.
 - The host's minor number (second from the left) must be greater or equal to the container's minor number. In case the minor number from the container is greater than the host's minor number, the hook will print a warning but will proceed in the attempt to let the container application run.
 - If the host's library name does not contain the version numbers or contains only the major version number, the missing numbers are assumed to be zero.

This compatibility check is in agreement with the MPICH ABI version number schema.

- **MPI_DEPENDENCY_LIBS**: Colon separated list of absolute paths to libraries that are dependencies of the **MPI_LIBS**. These libraries are always bind mounted in the container under `/usr/lib`.
- **BIND_MOUNTS**: Colon separated list of absolute paths to generic files or directories that are required for the correct functionality of the host MPI implementation (e.g. specific device files). These resources will be bind mounted inside the container with the same path they have on the host. If a path corresponds to a device file, that file will be whitelisted for read/write access in the container's devices cgroup.

The following is an example of **OCI hook JSON configuration file** enabling the MPI hook:

```
{
  "version": "1.0.0",
  "hook": {
    "path": "/opt/sarus/bin/mpi_hook",
    "env": [
      "LDCONFIG_PATH=/sbin/ldconfig",
      "MPI_LIBS=/usr/lib64/mvapich2-2.2/lib/libmpi.so.12.0.5:/usr/lib64/mvapich2-2.2/lib/libmpicxx.so.12.0.5:/usr/lib64/mvapich2-2.2/lib/libmpifort.so.12.0.5",
      "MPI_DEPENDENCY_LIBS=",
      "BIND_MOUNTS="
    ]
  },
  "when": {
    "annotations": {
      "^com.hooks.mpi.enabled$": "^true$"
    }
  },
  "stages": ["prestart"]
}
```

Configuring to leverage Sarus annotations and CLI options

Sarus automatically generates the `com.hooks.mpi.enabled=true` OCI annotation if the `--mpi` command line option is passed to **sarus run**. Such annotation can be entered in the hook configuration's `when` conditions to tie the activation of the hook to the presence of the `--mpi` option:

```
"when": {
  "annotations": {
    "^com.hooks.mpi.enabled$": "^true$"
  }
}
```

Additionally, the `--mpi-type` option of **sarus run** automatically generates both the `com.hooks.mpi.enabled=true` and `com.hooks.mpi.type=<value>` annotations, where the value of `com.hooks.mpi.type` is the value passed to the CLI option. This allows to configure multiple MPI hooks for different native MPI stacks (for example differing in implementation, compiler, or underlying communication framework) and choose one through the **sarus run** command line. For example:

```
# With these conditions, a hook will be enabled by '--mpi-type=mpich3'
"when": {
  "annotations": {
    "^com.hooks.mpi.enabled$": "^true$"
    "^com.hooks.mpi.type$": "^mpich3$"
  }
}

# With these conditions, a hook will be enabled by '--mpi-type=mpich4-libfabric'
"when": {
  "annotations": {
    "^com.hooks.mpi.enabled$": "^true$"
    "^com.hooks.mpi.type$": "^mpich4-libfabric$"
  }
}
```

When multiple hooks are configured, the `defaultMPIType` parameter in `sarus.json` can be used to define the default MPI hook for the system and allow users to enable it just by using the `--mpi` option.

Note: The rules for the OCI hook configuration file state that a hook is enabled only if *all* the `when` conditions match.

This means that if the administrator did not define `defaultMPIType` in the Sarus configuration and the user did not provide `--mpi-type=<value>` in the CLI arguments, the hooks configured with the `mpi.type` condition will **NOT** be enabled.

Note: Multiple hooks can be enabled at the same time by configuring them with identical conditions.

NVIDIA Container Runtime hook

NVIDIA provides access to GPU devices and their driver stacks inside OCI containers through an OCI hook called NVIDIA Container Runtime hook (part of the [NVIDIA Container Toolkit](#)), which acts as a driver for the `nvidia-container-cli` utility.

Dependencies

At least one NVIDIA GPU device and the NVIDIA CUDA Driver must be correctly installed and working in the host system.

The hook depends on the library and utility provided by the `libnvidia-container` project to carry out the low-level actions of importing the GPU device and drivers inside the container. At the time of writing, the latest release of `libnvidia-container` is version 1.13.0. The most straightforward way to obtain the library and its CLI utility binary is to configure the related [package repository](#) for your Linux distribution and install the pre-built packages. For example, on a RHEL-based distribution:

```
# Configure the NVIDIA repository
$ DIST=$(. /etc/os-release; echo $ID$VERSION_ID)
$ curl -s -L https://nvidia.github.io/libnvidia-container/$DIST/libnvidia-container.repo \
  | sudo tee /etc/yum.repos.d/libnvidia-container.repo

# Install the packages
$ sudo yum install libnvidia-container1 libnvidia-container-tools
```

Installation

At the time of writing, the latest revision of the NVIDIA Container Runtime hook is version 1.13.0.

System packages

NVIDIA provides distribution-specific packages for their Container Toolkit (which includes the hook) through the `libnvidia-container` repository.

For example, to install the hook on RHEL 8:

```
# Configure the NVIDIA repository
$distribution=$(. /etc/os-release; echo $ID$VERSION_ID) \
$ curl -s -L https://nvidia.github.io/libnvidia-container/$distribution/libnvidia-
  container.repo | \
  sudo tee /etc/yum.repos.d/nvidia-container-toolkit.repo

# Install the package
sudo dnf clean expire-cache \
&& sudo dnf install -y nvidia-container-toolkit
```

More information is available on the [official NVIDIA documentation](#).

Build from source

To build the hook from source, an installation of the [Go programming language](#) is needed. System packages are available for major Linux distributions, or you can follow the official documentation for [manual installation](#).

For a regular installation, the default Go working directory is `$HOME/go`. Should you prefer a different location, remember to set it as the `$GOPATH` in your environment.

You can now proceed to build the hook:

```
$ git clone https://github.com/NVIDIA/nvidia-container-toolkit.git
$ cd nvidia-container-toolkit
$ git checkout v1.13.0
$ make cmd-nvidia-container-runtime-hook

# Copy the hook binary to an installation directory
$ sudo cp ./nvidia-container-runtime-hook /opt/sarus/bin/nvidia-container-runtime-hook-1.
↪ 13.0
```

To ensure correct functionality, the hook also needs a TOML configuration file to be present on the system, and will look for it in the default path `/etc/nvidia-container-runtime/config.toml`, unless instructed otherwise. The configuration file is platform specific (for example, it tells the hook where to find the system's `ldconfig`). NVIDIA provides basic flavors for Ubuntu, Debian, OpenSUSE Leap, and distributions based on the YUM/DNF package manager (e.g. CentOS, Fedora, Amazon Linux 2):

```
# Install hook config.toml (e.g. for OpenSUSE Leap)
$ sudo mkdir /etc/nvidia-container-runtime/
$ sudo cp <NVIDIA Container Toolkit git repo>/config/config.toml.opensuse-leap /etc/
↪ nvidia-container-runtime/config.toml
```

Configuration

The NVIDIA Container Runtime hook is meant to run as a **prestart** hook. It also expects to receive its own name/location as the first program argument, and the string **prestart** as positional argument. Any other positional argument will cause the hook to return immediately without performing any action.

The hook environment also needs to grant visibility to the `libnvidia-container` library (e.g. `libnvidia-container.so`) and CLI utility (`nvidia-container-cli`).

Below is an example of [OCI hook JSON configuration file](#) enabling the NVIDIA Container Runtime hook. Notice that in this example the `-config=/path/to/config.toml` flag is entered before the **prestart** positional argument to point the hook to a configuration file installed in a custom location. Such flag is not required if you installed the TOML configuration file in the custom location as described in the previous section.

```
{
  "version": "1.0.0",
  "hook": {
    "path": "/opt/sarus/bin/nvidia-container-runtime-hook",
    "args": ["nvidia-container-runtime-hook", "-config=/opt/sarus/bin/config.toml",
↪ "prestart"],
    "env": [
      "PATH=/usr/local/libnvidia-container_1.13.0/bin",
      "LD_LIBRARY_PATH=/usr/local/libnvidia-container_1.13.0/lib"
    ]
  }
}
```

(continues on next page)

(continued from previous page)

```

    },
    "when": {
        "always": true
    },
    "stages": ["prestart"]
}

```

Sarus support at runtime

The actions performed by the NVIDIA Container Runtime hook are controlled via a set of specific [environment variables](#). Most of these can (and should) come from container images or from the host environment (see [here](#) for more about environments in Sarus containers). Notably, the `NVIDIA_VISIBLE_DEVICES` variable defines which GPUs will be made accessible inside the container by the hook.

However, in an HPC scenario, the hardware resources should be assigned from a supervisory entity, such as a workload manager. For example, the SLURM workload manager Generic Resource Scheduling (GRES) plugin selects which GPU devices are assigned to a job by setting the `CUDA_VISIBLE_DEVICES` environment variable inside the job process.

For this reason, when preparing a container Sarus will look for `CUDA_VISIBLE_DEVICES` in the *host* environment, and modify accordingly both `NVIDIA_VISIBLE_DEVICES` and `CUDA_VISIBLE_DEVICES` in the *container*. These modifications ensure that the host resource allocations are respected, while guaranteeing the correct operation of CUDA applications inside the container, even in the case of partial or shuffled devices selection on multi-GPU systems.

Mount hook

The source code of Sarus includes a hook able to perform an arbitrary sequence of bind mounts and device mounts (including device whitelisting in the related cgroup) into a container.

When activated, the hook enters the mount namespace of the container and performs the mounts it received as CLI arguments. The formats for such arguments are the same as those for the `--mount` and `--device` options of `sarus run`. This design choice has several advantages:

- Reuses established formats adopted by popular container tools.
- Explicitly states mount details and provides increased clarity compared to lists separated by colons or semicolons (for example, path lists used by other hooks).
- Reduces the effort to go from experimentation to consolidation: the mounts for a feature can be explored and prototyped on the Sarus command line, then transferred directly into a Mount hook configuration file.

In effect, the hook produces the same outcome as entering its `--mount` and `--device` option arguments in the command line of Sarus (or other engines with a similar CLI, like Docker and Podman).

However, the hook provides a way to group together the definition and execution of mounts related to a specific feature. By doing so, the feature complexity is abstracted from the user and feature activation becomes either comfortable (e.g. via a single CLI option) or completely transparent (e.g. if the hook either is always active or if it relies on an OCI annotation from the container image). Some example use cases are described in [this section](#).

Note: Compared to the [MPI](#) or [Glibc hooks](#), the Mount hook does not check ABI or version compatibility of mounted resources, and it does not deduce on its own the mount destination paths within the container, since its purpose is not strictly tied to replacing library stacks.

Hook installation

The hook is written in C++ and it will be compiled when building Sarus without the need of additional dependencies. Sarus' installation scripts will also automatically install the hook in the `$CMAKE_INSTALL_PREFIX/bin` directory. In short, no specific action is required to install the MPI hook.

Hook configuration

The program is meant to be run as a **prestart** hook and accepts option arguments with the same formats as the `--mount` or `--device` options of **sarus run**.

The hook also supports the following environment variables:

- `LDCONFIG_PATH` (optional): Absolute path to a trusted `ldconfig` program **on the host**. If set, the program at the path is used to update the container's dynamic linker cache after performing the mounts.

The following is an example of **OCI hook JSON configuration file** enabling the MPI hook:

```
{
  "version": "1.0.0",
  "hook": {
    "path": "/opt/sarus/bin/mount_hook",
    "args": ["mount_hook",
      "--mount=type=bind,src=/usr/lib/libexample.so.1,dst=/usr/local/lib/
↪ libexample.so.1",
      "--mount=type=bind,src=/etc/configs,dst=/etc/host_configs,readonly",
      "--device=/dev/example:rw"
    ],
    "env": [
      "LDCONFIG_PATH=/sbin/ldconfig"
    ]
  },
  "when": {
    "always": true
  },
  "stages": ["prestart"]
}
```

Example use cases

Libfabric provider injection

Libfabric is a communication framework which can be used as a middleware to abstract network hardware from an MPI implementation. Access to different fabrics is enabled through dedicated software components, which are called *libfabric providers*.

Fabric provider injection¹ consists in bind mounting a dynamically-linked provider and its dependencies into a container, so that containerized applications can access a high-performance fabric which is not supported in the original container image. For a formal introduction, evaluation, and discussion about the advantages of this approach, please refer to the reference publication.

¹ A. Madonna and T. Aliaga, "Libfabric-based Injection Solutions for Portable Containerized MPI Applications", 2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), Dallas, TX, USA, 2022, pp. 45-56, <https://doi.org/10.1109/CANOPIE-HPC56864.2022.00010>.

To facilitate the implementation of fabric provider injection, the Mount hook supports the `<FI_PROVIDER_PATH>` wildcard (angle brackets included) in `--mount` arguments. `FI_PROVIDER_PATH` is an environment variable recognized by libfabric itself, which can be used to control the path where libfabric searches for external, dynamically-linked providers. The wildcard is recognized by the hook during the acquisition of CLI arguments, and is substituted with a path obtained through the following conditions:

- If the `FI_PROVIDER_PATH` environment variable exists within the container, its value is taken.
- If `FI_PROVIDER_PATH` is unset or empty in the container's environment, and the `LDCONFIG_PATH` variable is configured for the hook, then the hook searches for a libfabric library in the container's dynamic linker cache, and obtains its installation path. The wildcard value is then set to `"libfabric library install path"/libfabric`, which is the default search path used by libfabric. For example, if libfabric is located at `/usr/lib64/libfabric.so.1`, the wildcard value will be `/usr/lib64/libfabric`.
- If it's not possible to determine a value with the previous methods, the wildcard value is set to `/usr/lib`.

The following is an example of hook configuration file using the wildcard to perform the injection of the GNI provider, enabling access to the Cray Aries high-speed interconnect on a Cray XC50 supercomputer:

```
{
  "version": "1.0.0",
  "hook": {
    "path": "/opt/sarus/default/bin/mount_hook",
    "args": ["mount_hook",
      "--mount=type=bind,src=/usr/local/libfabric/1.18.0/lib/libfabric/libgnix-fi.
↪so,dst=<FI_PROVIDER_PATH>/libgnix-fi.so",
      "--mount=type=bind,src=/opt/cray/xpmem/default/lib64/libxpmem.so.0,dst=/usr/
↪lib/libxpmem.so.0",
      "--mount=type=bind,src=/opt/cray/ugni/default/lib64/libugni.so.0,dst=/usr/
↪lib64/libugni.so.0",
      "--mount=type=bind,src=/opt/cray/udreg/default/lib64/libudreg.so.0,dst=/usr/
↪lib64/libudreg.so.0",
      "--mount=type=bind,src=/opt/cray/alps/default/lib64/libalpsutil.so.0,dst=/
↪usr/lib64/libalpsutil.so.0",
      "--mount=type=bind,src=/opt/cray/alps/default/lib64/libalpslli.so.0,dst=/usr/
↪lib64/libalpslli.so.0",
      "--mount=type=bind,src=/opt/cray/wlm_detect/default/lib64/libwlm_detect.so.0,
↪dst=/usr/lib64/libwlm_detect.so.0",
      "--mount=type=bind,src=/var/opt/cray/alps,dst=/var/opt/cray/alps",
      "--mount=type=bind,src=/etc/opt/cray/wlm_detect,dst=/etc/opt/cray/wlm_detect
↪",
      "--mount=type=bind,src=/opt/gcc/10.3.0/snos/lib64/libatomic.so.1,dst=/usr/
↪lib/libatomic.so.1",
      "--device=/dev/kgni0",
      "--device=/dev/kdreg",
      "--device=/dev/xpmem"
    ],
    "env": [
      "LDCONFIG_PATH=/sbin/ldconfig"
    ]
  },
  "when": {
    "annotations": {
      "^com.hooks.mpi.enabled$": "^true$",
      "^com.hooks.mpi.type$": "^libfabric$"
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    }
  },
  "stages": ["prestart"]
}

```

Accessing a host Slurm WLM from inside a container

The Slurm workload manager from the host system can be exposed within containers through a set of bind mounts. Doing so enables containers to submit new allocations and jobs to the cluster, opening up the possibility for more articulated workflows.

The key components to bind mount are the binaries for Slurm commands, the host Slurm configuration, the MUNGE socket, and any related dependencies. Below you may find an example of hook configuration file enabling access to the host Slurm WLM on a Cray XC50 system at CSCS:

```

{
  "version": "1.0.0",
  "hook": {
    "path": "/opt/sarus/default/bin/mount_hook",
    "args": ["mount_hook",
      "--mount=type=bind,src=/usr/bin/salloc,dst=/usr/bin/salloc",
      "--mount=type=bind,src=/usr/bin/sbatch,dst=/usr/bin/sbatch",
      "--mount=type=bind,src=/usr/bin/sinfo,dst=/usr/bin/sinfo",
      "--mount=type=bind,src=/usr/bin/squeue,dst=/usr/bin/squeue",
      "--mount=type=bind,src=/usr/bin/srun,dst=/usr/bin/srun",
      "--mount=type=bind,src=/etc/slurm,dst=/etc/slurm",
      "--mount=type=bind,src=/usr/lib64/slurm,dst=/usr/lib64/slurm",
      "--mount=type=bind,src=/var/run/munge,dst=/run/munge",
      "--mount=type=bind,src=/usr/lib64/libmunge.so.2,dst=/usr/lib64/libmunge.so.2",
    ],
    "mounts": [
      "--mount=type=bind,src=/opt/cray/alpscomm/default/lib64/libalpscomm_sn.so.0,dst=/usr/lib64/libalpscomm_sn.so.0",
      "--mount=type=bind,src=/opt/cray/alpscomm/default/lib64/libalpscomm_cn.so.0,dst=/usr/lib64/libalpscomm_cn.so.0",
      "--mount=type=bind,src=/opt/cray/swrap/default/lib64/libswrap.so.0,dst=/usr/lib64/libswrap.so.0",
      "--mount=type=bind,src=/opt/cray/socketauth/default/lib64/libsocketauth.so.0,dst=/usr/lib64/libsocketauth.so.0",
      "--mount=type=bind,src=/opt/cray/comm_msg/default/lib64/libcomm_msg.so.0,dst=/usr/lib64/libcomm_msg.so.0",
      "--mount=type=bind,src=/opt/cray/sysadm/default/lib64/libsysadm.so.0,dst=/usr/lib64/libsysadm.so.0",
      "--mount=type=bind,src=/opt/cray/codbc/default/lib64/libcodbc.so.0,dst=/usr/lib64/libcodbc.so.0",
      "--mount=type=bind,src=/opt/cray/nodeservices/default/lib64/libnodeservices.so.0,dst=/usr/lib64/libnodeservices.so.0",
      "--mount=type=bind,src=/opt/cray/sysutils/default/lib64/libsysutils.so.0,dst=/usr/lib64/libsysutils.so.0",
      "--mount=type=bind,src=/opt/cray/pe/atp/libAtpDispatch.so,dst=/opt/cray/pe/atp/libAtpDispatch.so",
      "--mount=type=bind,src=/opt/cray/pe/atp/3.14.5/slurm/libAtpSLaunch.so,dst=/opt/cray/pe/atp/3.14.5/slurm/libAtpSLaunch.so",
    ]
  }
}

```

(continues on next page)

(continued from previous page)

```

        "--mount=type=bind,src=/usr/lib64/libxmlrpc-epi.so.0,dst=/usr/lib64/
↪libxmlrpc-epi.so.0",
        "--mount=type=bind,src=/usr/lib64/libodbc.so.2,dst=/usr/lib64/libodbc.so.2",
        "--mount=type=bind,src=/usr/lib64/libexpat.so.1,dst=/usr/lib64/libexpat.so.1
↪",
        "--mount=type=bind,src=/usr/lib64/libltdl.so.7,dst=/usr/lib64/libltdl.so.7",
        "--mount=type=bind,src=/opt/cray/job/default/lib64/libjob.so.0,dst=/usr/
↪lib64/libjob.so.0",
        "--mount=type=bind,src=/opt/cray/job/default/lib64/libjobctl.so.0,dst=/usr/
↪lib64/libjobctl.so.0",
        "--mount=type=bind,src=/opt/cray/ugni/default/lib64/libugni.so.0,dst=/usr/
↪lib64/libugni.so.0",
        "--mount=type=bind,src=/usr/lib64/libjansson.so.4,dst=/usr/lib64/libjansson.
↪so.4",
        "--mount=type=bind,src=/opt/cscs/jobreport/jobreport.so,dst=/opt/cscs/
↪jobreport/jobreport.so",
        "--mount=type=bind,src=/opt/cscs/nohome/nohome.so,dst=/opt/cscs/nohome/
↪nohome.so",
        "--mount=type=bind,src=/usr/lib64/libslurm.so.36,dst=/usr/lib64/libslurm.so.
↪36",
        "--mount=type=bind,src=/usr/lib64/libcurl.so.4,dst=/usr/lib64/libcurl.so.4",
        "--mount=type=bind,src=/usr/lib64/libnghttp2.so.14,dst=/usr/lib64/libnghttp2.
↪so.14",
        "--mount=type=bind,src=/usr/lib64/libssh.so.4,dst=/usr/lib64/libssh.so.4",
        "--mount=type=bind,src=/usr/lib64/libpsl.so.5,dst=/usr/lib64/libpsl.so.5",
        "--mount=type=bind,src=/usr/lib64/libssl.so.1.1,dst=/usr/lib64/libssl.so.1.1
↪",
        "--mount=type=bind,src=/usr/lib64/libcrypto.so.1.1,dst=/usr/lib64/libcrypto.
↪so.1.1",
        "--mount=type=bind,src=/usr/lib64/libldap_r-2.4.so.2,dst=/usr/lib64/libldap_
↪r-2.4.so.2",
        "--mount=type=bind,src=/usr/lib64/liblber-2.4.so.2,dst=/usr/lib64/liblber-2.
↪4.so.2",
        "--mount=type=bind,src=/usr/lib64/libsas12.so.3,dst=/usr/lib64/libsas12.so.3
↪",
        "--mount=type=bind,src=/usr/lib64/libyaml-0.so.2,dst=/usr/lib64/libyaml-0.so.
↪2"
    ],
    "env": [
        "LDCONFIG_PATH=/sbin/ldconfig"
    ]
},
    "when": {
        "annotations": {
            "^com.hooks.slurm.activate$": "^true$"
        }
    },
    "stages": ["prestart"]
}

```

The following is an example usage of the hook as configured above:

```
$ srun --pty sarus run --annotation=com.hooks.slurm.activate=true -t debian:11 bash

nid00040:/$ cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
NAME="Debian GNU/Linux"
VERSION_ID="11"
VERSION="11 (bullseye)"
VERSION_CODENAME=bullseye
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"

nid00040:/$ srun --version
slurm 20.11.8

nid00040:/$ squeue -u <username>
  JOBID      USER      ACCOUNT   NAME  ST REASON   START_TIME   TIME   TIME_LEFT  NODES  _
↪CPUS
  714067  <username>  <account>  sarus   R None    12:48:41    0:40    59:20      1  _
↪24

nid00040:/$ salloc -N4 /bin/bash
salloc: Waiting for resource configuration
salloc: Nodes nid0000[0-3] are ready for job

nid00040:/$ srun -n4 hostname
nid00002
nid00003
nid00000
nid00001

# exit from inner Slurm allocation
nid00040:/$ exit
# exit from the container
nid00040:/$ exit
```

References

Native glibc hook

Sarus's source code includes a hook able to inject glibc libraries from the host inside the container, overriding the glibc of the container.

This is useful in case that we need to upgrade the container's glibc to a newer version. For example, when we want to inject a host's library inside the container (e.g. MPI), but the host's library relies on a newer glibc than the container's one.

When activated, the hook will enter the mount namespace of the container, search for dynamically-linkable glibc libraries and replace them with functional equivalents bind-mounted from the host system.

In order for the replacements to work seamlessly, the hook does the following:

- Compare glibc versions of host and container. The container's libraries are only replaced when they are older than the host's libraries.
- Check ABI compatibility between host and container glibc. Host and container glibc libraries must have the same soname for the replacement to take place.

If an ABI-compatible counterpart for a host library cannot be found in the container, the host library will be added to the container's filesystem.

Hook installation

The hook is written in C++ and it will be compiled when building Sarus without the need of additional dependencies. Sarus's installation scripts will also automatically install the hook in the `$CMAKE_INSTALL_PREFIX/bin` directory. In short, no specific action is required to install the glibc hook.

Hook configuration

The program is meant to be run as a **prestart** hook and does not accept arguments, but its actions are controlled through a few environment variables:

- `LDD_PATH`: Absolute path to a trusted `ldd` program **on the host**.
- `LDCONFIG_PATH`: Absolute path to a trusted `ldconfig` program **on the host**.
- `READELF_PATH`: Absolute path to a trusted `readelf` program **on the host**.
- `GLIBC_LIBS`: Colon separated list of full paths to the host's glibc libraries that will substitute the container's libraries.

The following is an example of [OCI hook JSON configuration file](#) enabling the glibc hook:

```
{
  "version": "1.0.0",
  "hook": {
    "path": "/opt/sarus/bin/glibc_hook",
    "env": [
      "LDD_PATH=/usr/bin/ldd",
      "LDCONFIG_PATH=/sbin/ldconfig",
      "READELF_PATH=/usr/bin/readelf",
      "GLIBC_LIBS=/lib64/libSegFault.so:/lib64/librt.so.1:/lib64/libnss_dns.so.2:/
↪ lib64/libanl.so.1:/lib64/libresolv.so.2:/lib64/libnsl.so.1:/lib64/libBrokenLocale.so.
↪ 1:/lib64/ld-linux-x86-64.so.2:/lib64/libnss_hesiod.so.2:/lib64/libutil.so.1:/lib64/
↪ libnss_files.so.2:/lib64/libnss_compat.so.2:/lib64/libnss_db.so.2:/lib64/libm.so.6:/
↪ lib64/libcrypt.so.1:/lib64/libc.so.6:/lib64/libpthread.so.0:/lib64/libdl.so.2:/lib64/
↪ libmvec.so.1:/lib64/libc.so.6:/lib64/libthread_db.so.1"
    ]
  },
  "when": {
    "annotations": {
      "^com.hooks.glibc.enabled$": "^true$"
    }
  },
  "stages": ["prestart"]
}
```

Sarus support at runtime

The `com.hooks.glibc.enabled=true` annotation that enables the hook is automatically generated by Sarus if the `--glibc` or `--mpi` command line options are passed to **sarus run**.

SSH hook

Sarus also includes the source code for a hook capable of enabling SSH connections inside containers. The SSH hook is an executable binary that performs different ssh-related operations depending on the argument it receives from the runtime. For the full details about the implementation and inner workings, please refer to the related [developer documentation](#).

Hook installation

The hook is written in C++ and it will be compiled along with Sarus if the `ENABLE_SSH=TRUE` CMake option has been used when configuring the build (the option is enabled by default). The Sarus installation scripts will also automatically install the hook in the `<CMAKE_INSTALL_PREFIX>/bin` directory.

A custom SSH software (statically linked Dropbear) will also be built and installed in the `<CMAKE_INSTALL_PREFIX>/dropbear` directory. This directory must satisfy the [security requirements](#) for critical files and directories.

Hook configuration

The SSH hook must be configured to run as a **prestart** hook. It expects to receive its own name/location as the first argument, and the string `start-ssh-daemon` as positional argument. In addition, the following environment variables must be defined:

- `HOOK_BASE_DIR`: Absolute base path to the directory where the hook will create and access the SSH keys. The keys directory will be located in `<HOOK_BASE_DIR>/<username>/.oci-hooks/ssh/keys`.
- `PASSWD_FILE`: Absolute path to a password file (`PASSWD(5)`). The file is used by the hook to retrieve the username of the user.
- `DROPBEAR_DIR`: Absolute path to the location of the custom SSH software.
- `SERVER_PORT`: TCP port on which the SSH daemon will listen. This must be an unused port and is typically set to a value different than 22 in order to avoid clashes with an SSH daemon that could be running on the host.

The following is an example of [OCI hook JSON configuration file](#) enabling the SSH hook:

```
{
  "version": "1.0.0",
  "hook": {
    "path": "/opt/sarus/bin/ssh_hook",
    "env": [
      "HOOK_BASE_DIR=/home",
      "PASSWD_FILE=/opt/sarus/etc/passwd",
      "DROPBEAR_DIR=/opt/sarus/dropbear",
      "SERVER_PORT=15263"
    ],
    "args": [
      "ssh_hook",
      "start-ssh-daemon"
    ]
  }
}
```

(continues on next page)

(continued from previous page)

```

    ]
  },
  "when": {
    "annotations": {
      "^com.hooks.ssh.enabled$": "^true$"
    }
  },
  "stages": ["prestart"]
}

```

Sarus support at runtime

The command `sarus ssh-keygen` will call the hook without creating a container, passing the appropriate arguments to generate dedicated keys to be used by containers.

The `com.hooks.ssh.enabled=true` annotation that enables the hook is automatically generated by Sarus if the `--ssh` command line option is passed to **sarus run**.

Important: The SSH hook currently does not implement a poststop functionality and requires the use of a private PID namespace to cleanup the Dropbear daemon. Thus, the hook currently requires the use of a *private PID namespace* for the container. Thus, the `--ssh` option of **sarus run** implies `--pid=private`, and is incompatible with the use of `--pid=host`.

Slurm global sync hook

Sarus also includes the source code for a hook specifically targeting the Slurm Workload Manager. This hook synchronizes the startup of containers launched through Slurm, ensuring that all Slurm nodes have spawned a container before starting the user-requested application in any of them. This kind of synchronization is useful, for example, when used in conjunction with the *SSH hook*: if the containers are not all available, some of them might try to establish connections with containers that have yet to start, causing the whole job step execution to fail.

When activated, the hook will retrieve information about the current Slurm job step and node by reading three environment variables: `SLURM_JOB_ID`, `SLURM_NTASKS` and `SLURM_PROCID`. The hook will then create a job-specific synchronization directory inside the Sarus local repository of the user. Inside the synchronization directory, two subdirectories will be created: `arrival` and `departure`. In the `arrival` directory each hook from a Slurm task will create a file with its Slurm process ID as part of the filename, then periodically check if files from all other Slurm tasks are present in the directory. The check is performed every 0.1 seconds. When all processes have signaled their arrival by creating a file, the hooks proceed to signal their departure by creating a file in the `departure` directory, and then exit the hook program. The hook associated with the Slurm process ID 0 waits for all other hooks to depart and then cleans up the synchronization directory.

The arrival/departure strategy has been implemented to prevent edge-case race conditions: for example, if the file from the last arriving hook is detected and the cleanup started before the last arriving hook has checked for the presence of all other files, this situation would result in the deadlock of the last arriving hook. Having all hooks signal their departure before cleaning up the synchronization directory avoids this problem.

Hook installation

The hook is written in C++ and it will be compiled when building Sarus without the need of additional dependencies. Sarus's installation scripts will also automatically install the hook in the `$CMAKE_INSTALL_PREFIX/bin` directory. In short, no specific action is required to install the Slurm global sync hook.

Sarus configuration

The program is meant to be run as a **prestart** hook and does not accept any argument. The following environment variables must be defined:

- `HOOK_BASE_DIR`: Absolute base path to the directory where the hook will create and access the synchronization files. The sync directory will be located in `<HOOK_BASE_DIR>/<username>/.oci-hooks/slurm-global-sync`.
- `PASSWD_FILE`: Absolute path to a password file (`PASSWD(5)`). The file is used by the hook to retrieve the username of the user.

The following is an example of [OCI hook JSON configuration file](#) enabling the Slurm global sync hook:

```
{
  "version": "1.0.0",
  "hook": {
    "path": "/opt/sarus/bin/slurm_global_sync_hook",
    "env": [
      "HOOK_BASE_DIR=/home",
      "PASSWD_FILE=/opt/sarus/etc/passwd"
    ]
  },
  "when": {
    "annotations": {
      "^com.hooks.slurm-global-sync.enabled$": "^true$"
    }
  },
  "stages": ["prestart"]
}
```


Sarus support at runtime

The `com.hooks.slurm-global-sync.enabled=true` annotation that enables the hook is automatically generated by Sarus if the `--ssh` command line option is passed to **sarus run** to make sure that containers will not attempt to ssh into other containers that have yet to start.

Timestamp hook

The OCI specifications do not define any requirement on exposing information about the inner workings of runtimes and hooks to the user. This can make determining the startup overhead of a standard container runtime difficult.

Sarus bundles a hook which leaves a timestamp on a logfile, accompanied by a configurable message. Since the OCI Runtime Specification mandates hooks to be executed in the order they are entered during configuration, interleaving this hook between other hooks generates useful data to measure the time spent by each component.

The timestamp has the following format:

```
[<UNIX time in seconds.nanoseconds>] [<hostname>-<hook PID>] [hook] [INFO] Timestamp
hook: <optional configurable message>
```

The following is an actual timestamp example:

```
[1552438146.449463] [nid07641-16741] [hook] [INFO] Timestamp hook: After-runtime
```

This hook does not alter the container in any way, and is primarily meant as a tool for developers and system administrators working with OCI hooks.

Hook installation

The hook is written in C++ and it will be compiled when building Sarus without the need of additional dependencies. Sarus's installation scripts will also automatically install the hook in the `$CMAKE_INSTALL_PREFIX/bin` directory. In short, no specific action is required to install the Timestamp hook.

Sarus configuration

The Timestamp hook can be configured to run at any of the container lifecycle phases supported for hook execution (prestart, poststart, poststop), since it is not tied to the workings of other hooks or the container application.

The hook optionally supports the following environment variable:

- **TIMESTAMP_HOOK_MESSAGE**: String to display as part of the timestamp printed to the target file. This variable is optional, and it is meant to differentiate between subsequent invocations of the timestamp hook for the same container.

The following is an example of [OCI hook JSON configuration file](#) enabling the Timestamp hook:

```
{
  "version": "1.0.0",
  "hook": {
    "path": "/opt/sarus/bin/timestamp_hook",
    "env": [
      "TIMESTAMP_HOOK_MESSAGE=After-runtime"
    ]
  },
}
```

(continues on next page)

(continued from previous page)

```

    "when": {
        "always": true
    },
    "stages": ["prestart"]
}

```

As mentioned above, the real value of the Timestamp hook lies in interleaving it between other hooks in order to have a measurement of the elapsed time. For example, using other hooks described in this documentation and creating multiple Timestamp hook JSON configuration files:

```

$ ls /opt/sarus/etc/hooks.d
00-timestamp-hook.json
01-glibc-hook.json
02-timestamp-hook.json
03-nvidia-container-toolkit.json
04-timestamp-hook.json
05-mpi-hook.json
06-timestamp-hook.json
07-ssh-hook.json
08-timestamp-hook.json
09-slurm-global-sync-hook.json
10-timestamp-hook.json

```

The previous example could produce an output in the logfile like the following:

```

[775589.671527655] [hostname-12385] [hook] [INFO] Timestamp hook: After-runtime
[775589.675871678] [hostname-12386] [hook] [INFO] Timestamp hook: After-glibc-hook
[775589.682727735] [hostname-12392] [hook] [INFO] Timestamp hook: After-NVIDIA-hook
[775589.685961371] [hostname-12393] [hook] [INFO] Timestamp hook: After-MPI-hook
[775589.690460309] [hostname-12394] [hook] [INFO] Timestamp hook: After-SSH-hook
[775589.693946863] [hostname-12396] [hook] [INFO] Timestamp hook: After-SLURM-global-
↪ sync-hook

```

Sarus support at runtime

The hook is activated by setting the `TIMESTAMP_HOOK_LOGFILE` variable in the *container* environment to the absolute path to the logfile where the hook has to print its timestamp. Note that the target logfile does not need to exist in the container's filesystem, since the OCI Runtime Specification mandates hooks to execute in the runtime namespace. If the variable is not set, the hook exits without performing any action.

When launching jobs with many containers (e.g. for an MPI application), it is advisable to point the Timestamp hook to a different file for each container, in order to avoid filesystem contention and obtain cleaner measurements. The following example shows one way to achieve this in a batch script for the Slurm Workload Manager.

```

#!/usr/bin/env bash
#SBATCH --job-name="sarus"
#SBATCH --nodes=<NNODES>
#SBATCH --ntasks-per-node=<NTASKS_PER_NODE>
#SBATCH --output=job.out
#SBATCH --time=00:05:00

echo "SLURM_JOBID="$SLURM_JOBID

```

(continues on next page)

(continued from previous page)

```
echo "START_TIME=`date +%s`"

srun bash -c 'file=<JOB_DIR>/out.procid_${SLURM_PROCID}; TIMESTAMP_HOOK_LOGFILE=${file}.
↳timestamp-hook sarus --verbose run --mpi <image> <application> &>${file}.sarus'

echo "END_TIME=`date +%s`"
```

The Timestamp hook does not require any direct support from the Sarus engine, but it relies on the presence of the `TIMESTAMP_HOOK_LOGFILE` variable in the container in order to work. Given the way Sarus creates the *container environment*, the variable can be set in two ways:

- in the host environment (like in the example above), having it automatically transferred inside the container by Sarus;
- directly in the container by using the `-e/--env` option of **sarus run**.

AMD GPU hook

The AMD GPU hook provides access to ROCm AMD GPU devices inside the container.

Hook installation

The hook is written in C++ and is compiled when building Sarus without the need for additional dependencies. The Sarus installation scripts automatically install the hook in the `$CMAKE_INSTALL_PREFIX/bin` directory. Thus, there is no specific action required to install the AMD GPU hook.

Hook configuration

The hook is designed to run as a **prestart** hook and does not accept any arguments. To enable the AMD GPU hook, use the following example of an OCI hook JSON configuration file:

```
{
  "version": "1.0.0",
  "hook": {
    "path": "/opt/sarus/bin/amdgpu_hook"
  },
  "when": {
    "always": true
  },
  "stages": ["prestart"]
}
```

It is generally recommended to configure the hook with the `"when": {"always": true}` condition, as it can automatically determine whether it should take action.

Hook support at runtime

The hook expects to find the following devices:

```
/dev/kfd

/dev/dri/card0
/dev/dri/renderD128
...
/dev/dri/cardN
/dev/dri/renderD<128+N>

/dev/by-path/pci-<bus>:00.0-card0
/dev/by-path/pci-<bus>:00.0-render0
...
/dev/by-path/pci-<bus>:00.0-cardN
/dev/by-path/pci-<bus>:00.0-renderN
```

If the `/dev/kfd` device is not present, the hook is automatically deactivated.

The hook uses the environment variable `ROCR_VISIBLE_DEVICES` to identify the GPU(s) that are made available within the container. If the variable is not present or empty, all GPU(s) are made available within the container. For more information about `ROCR_VISIBLE_DEVICES`, please refer to the following link: https://rocmdocs.amd.com/en/latest/ROCm_System_Managment/ROCm-System-Managment.html#rocr-visible-devices

1.5 User guides

1.5.1 User guide

Basic end-to-end user workflow

Steps to do **on your workstation**:

1. Build the container image using the Docker tool and Dockerfiles.
2. Push the Docker image into Docker Hub registry (<https://hub.docker.com>).

Steps to do **on the HPC system**:

3. Pull the Docker image from the Docker Hub registry using Sarus.
4. Run the image at scale with Sarus.

An explanation of the different steps required to deploy a Docker image using Sarus follows.

1. Develop the Docker image

First, the user has to build a container image. This boils down to writing a Dockerfile that describes the container, executing the Docker command line tool to build the image, and then running the container to test it. Below you can find an overview of what the development process looks like. We provide a brief introduction to using Docker, however, for a detailed explanation please refer to the [Docker Get Started guide](#).

Let's start with a simple example. Consider the following Dockerfile where we install Python on a Debian Jessie base image:

```
FROM debian:jessie
RUN apt-get -y update && apt-get install -y python
```

Once that the user has written the Dockerfile, the container image can be built:

```
$ docker build -t hello-python .
```

The previous step will take the content of the Dockerfile and build our container image. The first entry, `FROM debian:jessie`, will specify a base Linux distribution image as a starting point for our container (in this case, Debian Jessie), the image of which Docker will try to fetch from its registry if it is not already locally available. The second entry `RUN apt-get -y update && apt-get install -y python` will execute the command that follows the `RUN` instruction, updating the container with the changes done to its software environment: in this case, we are installing Python using the `apt-get` package manager.

Once the build step has finished, we can list the available container images using:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-python	latest	2e57316387c6	3 minutes ago	224 MB

We can now spawn a container from the image we just built (tagged as `hello-python`), and run Python inside the container (`python --version`), so as to verify the version of the installed interpreter:

```
$ docker run --rm hello-python python --version
Python 2.7.9
```

One of the conveniences of building containers with Docker is that this process can be carried out solely on the user's workstation/laptop, enabling quick iterations of building, modifying, and testing of containerized applications that can then be easily deployed on a variety of systems, greatly improving user productivity.

Note: Reducing the size of the container image, besides saving disk space, also speeds up the process of importing it into Sarus later on. The easiest ways to limit image size are cleaning the package manager cache after installations and deleting source codes and other intermediate build artifacts when building software manually. For practical examples and general good advice on writing Dockerfiles, please refer to the official [Best practices for writing Dockerfiles](#).

A user can also access the container interactively through a shell, enabling quick testing of commands. This can be a useful step for evaluating commands before adding them to the Dockerfile:

```
$ docker run --rm -it hello-python bash
root@c5fc1954b19d:/# python --version
Python 2.7.9
root@c5fc1954b19d:/#
```

2. Push the Docker image to the Docker Hub

Once the image has been built and tested, you can log in to [DockerHub](#) (requires an account) and push it, so that it becomes available from the cloud:

```
$ docker login
$ docker push <user name>/<repo name>:<image tag>
```

Note that in order for the push to succeed, the image has to be correctly tagged with the same `<repository name>/<image name>:<image tag>` identifier you intend to push to. Images can be tagged at build-time, supplying the `-t` option to **docker build**, or afterwards by using **docker tag**. In the case of our example:

```
$ docker tag hello-python <repo name>/hello-python:1.0
```

The image tag (the last part of the identifier after the colon) is optional. If absent, Docker will set the tag to `latest` by default.

3. Pull the Docker image from Docker Hub

Now the image is stored in the Docker Hub and you can pull it into the HPC system using the **sarus pull** command followed by the image identifier:

```
$ sarus pull <repo name>/hello-python:1.0
```

While performing the pull does not require specific privileges, it is generally advisable to run **sarus pull** on the system's compute nodes through the workload manager: compute nodes often have better hardware and, in some cases like Cray XC systems, large RAM filesystems, which will greatly reduce the pull process time and will allow to pull larger images.

Should you run into problems because the pulled image doesn't fit in the default filesystem, you can specify an alternative temporary directory with the `--temp-dir` option.

You can use **sarus images** to list the images available on the system:

```
$ sarus images
REPOSITORY          TAG          IMAGE ID      CREATED        SIZE
↳ SERVER
<repo name>/hello-python  1.0         6bc9d2cd1831  2018-01-19T09:43:04  40.16MB
↳ docker.io
```

4. Run the image at scale with Sarus

Once the image is available to Sarus we can run it at scale using the workload manager. For example, if using SLURM:

```
$ srun -N 1 sarus run <repo name>/hello-python:1.0 python --version
Python 2.7.9
```

As with Docker, containers can also be used through a terminal, enabling quick testing of commands:

```
$ srun -N 1 --pty sarus run -t debian bash
$ cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 9 (stretch)"
```

(continues on next page)

(continued from previous page)

```
NAME="Debian GNU/Linux"
VERSION_ID="9"
VERSION="9 (stretch)"
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"

$ exit
```

The `--pty` option to `srun` and the `-t/--tty` option to `sarus run` are needed to properly setup the pseudo-terminals in order to achieve a familiar user experience.

You can tell the previous example was run inside a container by querying the specifications of your host system. For example, the OS of Cray XC compute nodes is based on SLES and not Debian:

```
$ srun -N 1 cat /etc/os-release
NAME="SLES"
VERSION="12"
VERSION_ID="12"
PRETTY_NAME="SUSE Linux Enterprise Server 12"
ID="sles"
ANSI_COLOR="0;32"
CPE_NAME="cpe:/o:suse:sles:12"
```

Additional features

Pulling images from 3rd party registries

By default, Sarus tries to pull images from [Docker Hub](#). To pull an image from a registry different from Docker Hub, enter the server address as part of the image reference. For example, to access the NVIDIA GPU Cloud:

```
$ srun -N 1 sarus pull --login nvcr.io/nvidia/k8s/cuda-sample:nbody
username      :$oauthtoken
password      :
...
```

To work with images not pulled from Docker Hub (including the removal detailed in a later section), you need to enter the image reference as displayed by the **sarus images** command in the first two columns:

```
$ sarus images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
↪ SERVER
nvcr.io/nvidia/k8s/cuda-sample  nbody        29e2298d9f71     2019-01-14T12:22:25  91.
↪ 88MB             nvcr.io

$ srun -N1 sarus run nvcr.io/nvidia/k8s/cuda-sample:nbody cat /usr/local/cuda/version.txt
CUDA Version 9.0.176
```

Pulling images from private repositories

Remote registries may host image repositories which are not public, but require credentials in order to access their content. To retrieve images from a private repository, the **sarus pull** command offers several options.

The `--login` option allows to enter credentials through an interactive prompt after launching the command:

```
$ srunk -N 1 sarus pull --login <privateRepo>/<image>:<tag>
username: user
password:
...
```

`--login` also supports piping to stdin, at the condition that user name and password are separated by a newline character:

```
$ srunk -N 1 printf '<user>\n<password>' | sarus pull --login <privateRepo>/<image>:<tag>
```

The `-u/--user` option allows to provide the user name as part of the command line, while the `--password-stdin` option reads the password from stdin. These two options complement each other naturally:

```
$ srunk -N 1 cat passwordFile.txt | sarus pull --username <user> --password-stdin
-><privateRepo>/<image>:<tag>
```

It is also possible to combine the `-u/--user` option with `--login` (for example to automatically populate the username field in the interactive prompt); on the other hand, the `--password-stdin` option cannot be used in conjunction with `--login`.

Managing the registry authentication file

Sarus internally relies on [Skopeo](#) to download images from remote registries. When using the `--login` option, Sarus passes credentials to Skopeo through a `containers-auth.json(5)` authentication file, which is generated using data only for the specific repository being accessed.

The authentication file is created within the `${XDG_RUNTIME_DIR}/sarus` directory if the `XDG_RUNTIME_DIR` environment variable is defined and its value is the path to an existing directory. Otherwise, the file is created in the Sarus local repository for the current user.

Note: The path to the local repository for a given user can be obtained from the `localRepositoryBaseDir` parameter in the Sarus *configuration file* according to the pattern `<localRepositoryBaseDir>/<user name>/<image>:<tag>`.

The file is owned by the user who launched the **sarus pull** command, and is set to have owner-only read and write access to prevent exposing the registry credentials.

The authentication file generated by Sarus is intended to be specific for each pull invocation, and is automatically removed at the end of a successful pull process. If an image pull terminates abnormally, the file may be left lingering in the filesystem. In such case, it can be removed manually by the owner, or it will be overwritten and removed by the next successful **sarus pull** of a private image.

Pulling images by digest (immutable identifier)

Container images are usually pulled using a *tag*, which is an arbitrary label to differentiate images within the same repository. Image tags are mutable, and can potentially point to different images at different times, for example in the case an image is rebuilt.

Sarus supports the capability to pull images using a *digest*, an immutable identifier which uniquely and consistently points to a specific version of an image. Digests are useful to increase clarity and reproducibility in container workflows by allowing to reference exact software stack versions.

As [defined](#) by the OCI Image Specification, digests take the form of a string using the `<algorithm>:<encoded>` pattern. The `algorithm` portion indicates the cryptographic algorithm used for the digest, while the `encoded` portion represents the result of the hash function.

To pull an image by digest, append the digest to the image name using `@` as separator:

```
$ sarus pull
↳ debian@sha256:039f72a400b48c272c6348f0a3f749509b18e611901a21379abc7eb6edd53392
# image : docker.io/library/
↳ debian@sha256:039f72a400b48c272c6348f0a3f749509b18e611901a21379abc7eb6edd53392
# cache directory : "/home/<user>/sarus/cache"
# temp directory : "/tmp"
# images directory : "/home/<user>/sarus/images"
# image digest :
↳ sha256:039f72a400b48c272c6348f0a3f749509b18e611901a21379abc7eb6edd53392
Getting image source signatures
Copying blob 5492f66d2700 done
Copying config 3c3ca0ede6 done
Writing manifest to image destination
Storing signatures
> unpacking OCI image
> make squashfs image: "/home/<user>/sarus/images/docker.io/library/debian/sha256-
↳ 039f72a400b48c272c6348f0a3f749509b18e611901a21379abc7eb6edd53392.squashfs"
```

It is possible to combine tag and digest in the argument of the **sarus pull** command. In this case, Sarus proceeds to pull the image indicated by the digest and completely ignores the tag. This behavior is consistent with other container tools like Docker, Podman and Buildah:

```
$ sarus pull alpine:3.15.
↳ 2@sha256:73c155696fe65b68696e6ea24088693546ac468b3e14542f23f0efbde289cc97
# image : docker.io/library/alpine:3.15.
↳ 2@sha256:73c155696fe65b68696e6ea24088693546ac468b3e14542f23f0efbde289cc97
# cache directory : "/home/<user>/sarus/cache"
# temp directory : "/tmp"
# images directory : "/home/<user>/sarus/images"
# image digest :
↳ sha256:73c155696fe65b68696e6ea24088693546ac468b3e14542f23f0efbde289cc97
Getting image source signatures
Copying blob 3aa4d0bbde19 done
Copying config e367198082 done
Writing manifest to image destination
Storing signatures
> unpacking OCI image
> make squashfs image: "/home/<user>/sarus/images/docker.io/library/alpine/sha256-
↳ 73c155696fe65b68696e6ea24088693546ac468b3e14542f23f0efbde289cc97.squashfs"
```

Even if the tag is ignored by the pull, it can still serve as a visual aid for for users writing or reading the command, helping to understand what image the digest is pointing to.

Pulling images using a proxy

Sarus delegates communication with remote registries to [Skopeo](#), which in turn uses the Golang standard library to implement http connections. The usage of a proxy by the Golang [http package](#) can be controlled through a set of [environment variables](#).

The HTTP_PROXY and HTTPS_PROXY variables (or their lowercase versions) are used to indicate proxies for the respective protocols. For https requests, HTTPS_PROXY takes precedence over HTTP_PROXY.

To connect to a specific registry without going through the proxy, the NO_PROXY or no_proxy environment variables can be used. Such variables should contain a list of comma-separated values for which a proxy connection is not used.

The values of the environment variables may be either a complete URL or in a host[:port] format, in which case the http scheme is assumed. The supported schemes are http, https, and socks5. For example:

```
http_proxy=socket5://127.0.0.1:3128
https_proxy=https://127.0.0.1:3128
no_proxy=localhost,127.0.0.1,*.docker.io
```

Pulling images from insecure registries

Images from remote registries are downloaded through [Skopeo](#), which by default uses secure connections and the TLS protocol.

To enable image pulls from a registry unable to provide the required TLS certificates, the registry must be declared as "insecure" in a [registry configuration file](#) for container tools.

The main registry configuration file for a given user is located in `${HOME}/.config/containers/registries.conf`; if such file does not exist, `/etc/containers/registries.conf` is used instead.

In addition to the main file, specific drop-in configuration files can be created in `${HOME}/.config/containers/registries.conf.d` or in `/etc/containers/registries.conf.d`. Drop-in configuration files are loaded after the main configuration, and their values will overwrite any previous setting. For more details about drop-in configuration directories, please refer to the [containers-registries.conf.d\(5\)](#) manpage.

The following example shows how to create a drop-in registry configuration file defining an insecure registry located at `localhost:5000`:

```
$ mkdir -p ${HOME}/.config/containers/registries.conf.d
$ cat <<EOF > ${HOME}/.config/containers/registries.conf.d/insecure-localhost.conf
> [[registry]]
> prefix = "localhost:5000"
> insecure = true
> location = "localhost:5000"
> EOF
```

Notice the `insecure = true` setting. For the full details about the format and the available settings of a container registry configuration file, please refer to the [containers-registries.conf\(5\)](#) manpage.

Once an insecure registry has been entered in a configuration file, it is possible to pull images from it:

```
$ sarus pull localhost:5000/library/alpine
# image      : localhost:5000/library/alpine:latest
# cache directory : "/home/docker/.sarus/cache"
# temp directory  : "/tmp"
# images directory : "/home/docker/.sarus/images"
# image digest   : ↪
↪ sha256:e7d88de73db3d3fd9b2d63aa7f447a10fd0220b7cbf39803c803f2af9ba256b3
Getting image source signatures
Copying blob 59bf1c3509f3 done
Copying config d539cd357a done
Writing manifest to image destination
Storing signatures
> unpacking OCI image
> make squashfs image: "/home/docker/.sarus/images/localhost:5000/library/alpine/latest.
↪ squashfs"

$ sarus images
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE ↪
↪ SERVER
localhost:5000/library/alpine  latest      d539cd357acb     2022-03-15T17:24:08  2.61MB ↪
↪ localhost:5000
```

Warning: Please note that the use of insecure registries represents a significant security risk, and as such should only be used in exceptional cases such as local testing.

Download cache

During image pulls, Sarus stores individual image components (like filesystem layers and OCI configuration files) downloaded from registries in a cache directory, so they can be reused by subsequent pull commands.

The location of the cache directory for the current user is displayed at the beginning of the **sarus pull** command output, alongside image properties and other directories used by Sarus. The location of the cache can also be obtained from the *localRepositoryBaseDir* parameter in the *sarus.json configuration file*, using the following path format `<localRepositoryBaseDir>/<username>/.sarus/cache`.

The contents of the download cache can be deleted at any time to free up storage space.

Loading images from tar archives

If you do not have access to a remote Docker registry, or you're uncomfortable with uploading your images to the cloud in order to pull them, Sarus offers the possibility to load images from tar archives generated by *docker save*.

First, save an image to a tar archive using Docker *on your workstation*:

```
$ docker save --output debian.tar debian:jessie
$ ls -sh
total 124M
124M debian.tar
```

Then, transfer the archive to the HPC system and use the **sarus load** command, followed by the archive filename and the reference you want to give to the Sarus image:

```
$ sarus images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE          SERVER

$ srun sarus load ./debian.tar my_debian
> expand image layers ...
> extracting          : /tmp/debian.tar/
↳ 7e5c6402903b327fc62d1144f247c91c8e85c6f7b64903b8be289828285d502e/layer.tar
> make squashfs ...
> create metadata ...
# created: <user home>/sarus/images/load/library/my_debian/latest.squashfs
# created: <user home>/sarus/images/load/library/my_debian/latest.meta

$ sarus images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE          SERVER
↳ SERVER
load/library/my_debian  latest      2fe79f06fa6d    2018-01-31T15:08:56  47.04MB
↳ load
```

The image is now ready to use. Notice that the origin server for the image has been labeled `load` to indicate this image has been loaded from an archive.

Similarly to **sarus pull**, we recommend to load tar archives from compute nodes. Should you run out of space while unpacking the image, **sarus load** also accepts the `--temp-dir` option to specify an alternative unpacking directory.

As with images from 3rd party registries, to use or remove loaded images you need to enter the image reference as displayed by the **sarus images** command in the first two columns (`repository[:tag]`).

Displaying image digests

Images pulled by digest do not have a tag associated to them. In order to run or remove such images, it is necessary to provide the full digest after the image name. The digests of the images available in the Sarus local repository can be displayed using the `--digests` option of the **sarus images** command:

```
$ sarus images --digests
REPOSITORY  TAG          DIGEST          SIZE          SERVER
↳          IMAGE ID          CREATED
alpine      latest      ↳          2.61MB          docker.io
↳ sha256:73c155696fe65b68696e6ea24088693546ac468b3e14542f23f0efbde289cc97  e3671980822d↳
↳ 2022-03-25T13:17:13
fedora      latest      ↳          50.03MB          docker.io
↳ sha256:36af84ba69e21c9ef86a0424a090674c433b2b80c2462e57503886f1d823abe8  04d13a5c8de5↳
↳ 2022-03-25T13:17:57
ubuntu     <none>      ↳
↳ sha256:dcc176d1ab45d154b767be03c703a35fe0df16cfb1cc7ea5dd3b6f9af99b6718  4f4768f23ea4↳
↳ 2022-03-25T13:21:40  26.41MB          docker.io
```

Running images by digest

To run images pulled by digest, append the digest to the image name using @ as separator:

```
$ sarus images --digests
REPOSITORY    TAG          DIGEST
↳            IMAGE ID      CREATED      SIZE      SERVER
alpine        <none>
↳ sha256:73c155696fe65b68696e6ea24088693546ac468b3e14542f23f0efbde289cc97  e3671980822d
↳ 2022-03-25T14:28:45    2.61MB      docker.io

$ sarus run
↳ alpine@sha256:73c155696fe65b68696e6ea24088693546ac468b3e14542f23f0efbde289cc97 cat /
↳ etc/os-release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.15.2
PRETTY_NAME="Alpine Linux v3.15"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```

As with the **sarus pull** command, if both tag and digest are specified the tag is ignored and the image is looked up using the digest:

```
$ sarus run alpine:l.
↳ @sha256:73c155696fe65b68696e6ea24088693546ac468b3e14542f23f0efbde289cc97 cat /etc/os-
↳ release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.15.2
PRETTY_NAME="Alpine Linux v3.15"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```

Removing images

To remove an image from Sarus's local repository, use the **sarus rmi** command:

```
$ sarus images
REPOSITORY    TAG          IMAGE ID      CREATED      SIZE      SERVER
library/debian latest      6bc9d2cd1831  2018-01-31T14:11:27  40.17MB    docker.io

$ sarus rmi debian:latest
removed docker.io/library/debian/latest

$ sarus images
REPOSITORY    TAG          IMAGE ID      CREATED      SIZE      SERVER
```

To remove images pulled from 3rd party registries or images loaded from local tar archives you need to enter the image reference as displayed by the **sarus images** command:

```
$ sarus images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
↳SERVER
load/library/my_debian  latest      2fe79f06fa6d    2018-01-31T15:08:56  47.04MB
↳load

$ sarus rmi load/library/my_debian
removed load/library/my_debian/latest
```

To remove images pulled by digest, append the digest to the image name using @ as separator:

```
$ sarus rmi_
↳ubuntu@sha256:dcc176d1ab45d154b767be03c703a35fe0df16cfb1cc7ea5dd3b6f9af99b6718
removed image docker.io/library/
↳ubuntu@sha256:dcc176d1ab45d154b767be03c703a35fe0df16cfb1cc7ea5dd3b6f9af99b6718
```

Environment

Environment variables within containers are set by combining several sources, in the following order (later entries override earlier entries):

1. Host environment of the process calling Sarus
2. Environment variables defined in the container image, e.g., Docker ENV-defined variables
3. Modification of variables related to the *NVIDIA Container Toolkit*
4. Modifications (set/prepend/append/unset) specified by the system administrator in the Sarus configuration file. See [here](#) for details.
5. Environment variables defined using the `-e/--env` option of **sarus run**. The option can be passed multiple times, defining one variable per option. The first occurring `=` (equals sign) character in the option value is treated as the separator between the variable name and its value:

```
$ srun sarus run -e SARUS_CONTAINER=true debian bash -c 'echo $SARUS_CONTAINER'
SARUS_CONTAINER=true

$ srun sarus run --env=CLI_VAR=cli_value debian bash -c 'echo $CLI_VAR'
CLI_VAR=cli_value

$ srun sarus run --env NESTED=innerName=innerValue debian bash -c 'echo $NESTED'
NESTED=innerName=innerValue
```

If an `=` is not provided in the option value, Sarus considers the string as the variable name, and takes the value from the corresponding variable in the host environment. This can be used to override a variable set in the image with the value from the host. If no `=` is provided and a matching variable is not found in the host environment, the option is ignored and the variable is not set in the container.

Accessing host directories from the container

System administrators can configure Sarus to automatically mount facilities like parallel filesystems into every container. Refer to your site documentation or system administrator to know which resources have been enabled on a specific system.

Mounting custom files and directories into the container

By default, Sarus creates the container filesystem environment from the image and host system as specified by the system administrator. It is possible to request additional paths from the host environment to be mapped to some other path within the container using the `--mount` option of the **sarus run** command:

```
$ srunk -N 1 --pty sarus run --mount=type=bind,source=/path/to/my/data,destination=/data -
↪t debian bash
```

The previous command would cause `/path/to/my/data` on the host to be mounted as `/data` within the container. This mount option can be specified multiple times, one for each mount to be performed. `--mount` accepts a comma-separated list of `<key>=<value>` pairs as its argument, much alike the Docker option with the same name (for reference, see the [official Docker documentation on bind mounts](#)). As with Docker, the order of the keys is not significant. A detailed breakdown of the possible flags follows in the next subsections.

Mandatory flags

- **type**: represents the type of the mount. Currently, only `bind` (for bind-mounts) is supported.
- **source** (required): Absolute path accessible from the user *on the host* that will be mounted in the container. Can alternatively be specified as `src`.
- **destination**: Absolute path to where the filesystem will be made available inside the container. If the directory does not exist, it will be created. It is possible to overwrite other bind mounts already present in the container, however, the system administrator retains the power to disallow user-requested mounts to any location at their discretion. May alternatively be specified as `dst` or `target`.

Bind mounts

In addition to the mandatory flags, regular bind mounts can optionally add the following flag:

- **readonly** (optional): Causes the filesystem to be mounted as read-only. This flag takes no value.

The following example demonstrates the use of a custom read-only bind mount.

```
$ ls -l /input_data
drwxr-xr-x.  2 root root    57 Feb  7 10:49 ./
drwxr-xr-x. 23 root root  4096 Feb  7 10:49 ../
-rw-r--r--.  1 root root 1048576 Feb  7 10:49 data1.csv
-rw-r--r--.  1 root root 1048576 Feb  7 10:49 data2.csv
-rw-r--r--.  1 root root 1048576 Feb  7 10:49 data3.csv

$ echo "1,2,3,4,5" > data4.csv

$ srunk -N 1 --pty sarus run --mount=type=bind,source=/input_data,destination=/input,
↪readonly -t debian bash
```

(continues on next page)

(continued from previous page)

```
$ ls -l /input
-rw-r--r--. 1 root 0 1048576 Feb  7 10:49 data1.csv
-rw-r--r--. 1 root 0 1048576 Feb  7 10:49 data2.csv
-rw-r--r--. 1 root 0 1048576 Feb  7 10:49 data3.csv
-rw-r--r--. 1 root 0      10 Feb  7 10:52 data4.csv

$ cat /input/data4.csv
1,2,3,4,5

$ touch /input/data5.csv
touch: cannot touch '/input/data5.csv': Read-only file system

$ exit
```

Note: Bind-mounting FUSE filesystems into Sarus containers

By default, all FUSE filesystems are accessible only by the user who mounted them; this restriction is enforced by the kernel itself. Sarus is a privileged application setting up the container as the root user in order to perform some specific actions.

To allow Sarus to access a FUSE mount point on the host, in order to bind mount it into a container, use the FUSE option `allow_root` when creating the mount point. For example, when creating an [EncFS](#) filesystem:

```
$ encfs -o allow_root --nocache $PWD/encfs.enc/ /tmp/encfs.dec/
$ sarus run -t --mount=type=bind,src=/tmp/encfs.dec,dst=/var/tmp/encfs ubuntu ls -l /var/
→ tmp
```

It is possible to pass `allow_root` if the option `user_allow_other` is defined in `/etc/fuse.conf`, as stated in the [FUSE manpage](#).

Mounting custom devices into the container

Devices can be made available inside containers through the `--device` option of **sarus run**. The option can be entered multiple times, specifying one device per option. By default, device files will be mounted into the container at the same path they have on the host. A different destination path can be entered using a colon (:) as separator from the host path. All paths used in the option value must be absolute:

```
$ srunk sarus run --device=/dev/fuse debian ls -l /dev/fuse
crw-rw-rw-. 1 root root 10, 229 Aug 17 17:54 /dev/fuse

$ srunk sarus run --device=/dev/fuse:/dev/container_fuse debian ls -l /dev/container_fuse
crw-rw-rw-. 1 root root 10, 229 Aug 17 17:54 /dev/container_fuse
```

When working with device files, the `--mount` option should not be used since access to custom devices is disabled by default through the container's device cgroup. The `--device` option, on the other hand, will also whitelist the requested devices in the cgroup, making them accessible within the container. By default, the option will grant read, write and `mknod` permissions to devices; this behavior can be controlled by adding a set of flags at the end of an option value, still using a colon as separator. The flags must be a combination of the characters `rwm`, standing for *read*, *write* and *mknod* access respectively; the characters may come in any order, but must not be repeated. The access flags can be entered independently from the presence of a custom destination path.

The full syntax of the option is thus `--device=host-device[:container-device][:permissions]`

The following example shows how to mount a device with read-only access:

```
$ srun --pty sarus run -t --device=/dev/example:r debian bash

$ echo "hello" > /dev/example
bash: /dev/example: Operation not permitted

$ exit
```

Important: Sarus and the `--device` option cannot grant more permissions to a device than those which have been allowed on the host. For example, if in the host a device is set for read-only access, then Sarus cannot enable write or `mknod` access.

This is enforced by the implementation of device cgroups in the Linux kernel. For more details, please refer to the [kernel documentation](#).

Image entrypoint and default arguments

Sarus fully supports image entrypoints and default arguments as defined by the [OCI Image Specification](#).

The entrypoint of an image is a list of arguments that will be used as the command to execute when the container starts; it is meant to create an image that will behave like an executable file.

The image default arguments will be passed to the entrypoint if no argument is provided on the command line when launching a container. If the entrypoint is not present, the first default argument will be treated as the executable to run.

When creating container images with Docker, the entrypoint and default arguments are set using the `ENTRYPOINT` and `CMD` instructions respectively in the Dockerfile. For example, this file will generate an image printing arguments to the terminal by default:

```
FROM debian:stretch

ENTRYPOINT ["/bin/echo"]
CMD ["Hello world"]
```

After building such image (we'll arbitrarily call it `echo`) and importing it into Sarus, we can run it without passing any argument:

```
$ srun sarus run <image repo>/echo
Hello world
```

Entering a command line argument will override the default arguments passed to the entrypoint:

```
$ srun sarus run <image repo>/echo Foobar
Foobar
```

The image entrypoint can be changed by providing a value to the `--entrypoint` option of **sarus run**. It is important to note that when changing the entrypoint the default arguments get discarded as well:

```
$ srun sarus run --entrypoint=cat <image repo>/echo /etc/os-release
PRETTY_NAME="Debian GNU/Linux 9 (stretch)"
NAME="Debian GNU/Linux"
VERSION_ID="9"
```

(continues on next page)

(continued from previous page)

```
VERSION="9 (stretch)"
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

The entrypoint can be removed by passing an empty value to `--entrypoint`. This is useful, for example, for inspecting and debugging containers:

```
$ srunk --pty sarvus run --entrypoint "" -t <image repo>/echo bash
$ env | grep ^PATH
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

$ exit
```

Note: Using the "adjacent value" style to remove an image entrypoint (i.e. `--entrypoint=""`) is not supported. Please pass the empty string value separated by a whitespace.

Working directory

The working directory inside the container can be controlled using the `-w/--workdir` option of the **sarvus run** command:

```
$ srunk -N 1 --pty sarvus run --workdir=/path/to/workdir -t debian bash
```

If the path does not exist, it is created inside the container.

If the `-w/--workdir` option is not specified but the image defines a working directory, the container process will start there. Otherwise, the process will start in the container's root directory (`/`). Using image-defined working directories can be useful, for example, for simplifying the command line when launching containers.

When creating images with Docker, the working directory is set using the **WORKDIR** instruction in the Dockerfile.

PID namespace

The PID namespace for the container can be controlled through the `--pid` option of **sarvus run**. Currently, the supported values for the option are:

- **host**: use the host's PID namespace for the container. This allows to transparently support MPI implementations relying on the ranks having different PIDs when running on the same physical host, or using shared memory technologies like Cross Memory Attach (CMA); (**default**)
- **private**: create a new PID namespace for the container. Having a private PID namespace can be also referred as "using PID namespace isolation" or simply "using PID isolation".

```
$ sarvus run --pid=private alpine:3.14 ps -o pid,comm
PID    COMMAND
1      ps
```

Note: Consider using an *init process* when running with a private PID namespace if you need to handle signals or run many processes into the container.

Adding an init process to the container

By default, Sarus only executes the user-specified application within the container. When using a *private PID namespace*, the container process is assigned PID 1 in the new namespace. The PID 1 process has unique features in Linux: most notably, the process will ignore signals by default and zombie processes will not be reaped inside the container (see [1], [2] for further reference).

If you need to handle signals or reap zombie processes (which can be useful when executing several different processes in long-running containers), you can use the `--init` option to run an init process inside the container:

```
$ srunk -N 1 sarus run --pid=private --init alpine:3.14 ps -o pid,comm
PID  COMMAND
   1  init
   7  ps
```

Sarus uses *tini* as its default init process.

Warning: Some HPC applications may be subject to performance losses when run with an init process. Our internal benchmarking tests with *tini* showed overheads of up to 2%.

Setting OCI annotations

OCI annotations are defined in the OCI Runtime Specification as a mean to provide arbitrary metadata for the container in the form of a key-value map. Annotation keys usually express a hierarchical namespace structure, with domains separated by `.` (full stop) characters.

Annotations can be useful to control how Sarus selects OCI hooks or, depending on their implementation, certain features of OCI hooks and OCI runtimes.

To set an annotation when running a container, use the `--annotation` option of **sarus run**. The option can be passed multiple times, defining one annotation per option. The first occurring `=` (equals sign) character in the option value is treated as the separator between the annotation key and its value.

```
$ srunk -N 1 sarus run --annotation com.documentation.example.key=value debian true
```

Annotations set from the Sarus command line take precedence over other annotations, for example coming from the container image (sometimes also known as "image labels") or created automatically by Sarus itself. For example, the hooks shipped alongside Sarus use the `com.hooks.logging.level` annotation to determine their verbosity level. The annotation is created internally by Sarus to match the verbosity of the engine. A different verbosity level for the hooks can be defined by overriding the annotation as follows:

```
# Set Sarus hooks to print debug-level messages (level 0)
# while Sarus keeps its normal verbosity
$ srunk -N 1 sarus run --annotation=com.hooks.logging.level=0 debian true
```

Please refer to your site documentation or your system administrator to learn about custom features related to OCI annotations on a specific system.

Verbosity levels and help messages

To run a command in verbose mode, enter the `--verbose` global option before the command:

```
$ srun sarus --verbose run debian:latest cat /etc/os-release
```

To run a command printing extensive details about internal workings, enter the `--debug` global option before the command:

```
$ srun sarus --debug run debian:latest cat /etc/os-release
```

To print a general help message about Sarus, use `sarus --help`.

To print information about a command (e.g. command-specific options), use `sarus help <command>`:

```
$ sarus help run
Usage: sarus run [OPTIONS] REPOSITORY[:TAG] [COMMAND] [ARG...]
```

Run a **command** **in** a new container

Note: REPOSITORY[:TAG] has to be specified as displayed by the `"sarus images"` command.

Options:

<code>--centralized-repository</code>	Use centralized repository instead of the local one
<code>-t [--tty]</code>	Allocate a pseudo-TTY in the container
<code>--entrypoint arg</code>	Overwrite the default ENTRYPOINT of the image
<code>--mount arg</code>	Mount custom directories into the container
<code>-m [--mpi]</code>	Enable MPI support
<code>--ssh</code>	Enable SSH in the container

Support for container customization through hooks

Sarus allows containers to be customized by other programs or scripts leveraging the interface defined by the Open Container Initiative Runtime Specification for POSIX-platform hooks (OCI hooks for short). These customizations are especially amenable to HPC use cases, where the dedicated hardware and highly-tuned software adopted by high-performance systems are in contrast with the infrastructure-agnostic nature of software containers. OCI hooks provide a solution to open access to these resources inside containers.

The hooks which is possible to enable on a given Sarus installation are configured by the system administrators. The **sarus hooks** command can be used to list the currently configured hooks:

```
$ sarus hooks
```

NAME	PATH	STAGES
03-nvidia-container-runtime-hook	/usr/bin/nvidia-container-runtime-hook	↳
↳prestart		
05-mpi-hook	/opt/sarus/default/bin/mpi_hook	↳
↳prestart		
07-ssh-hook	/opt/sarus/default/bin/ssh_hook	↳
↳prestart		
09-slurm-global-sync-hook	/opt/sarus/default/bin/slurm_global_sync_hook	↳
↳prestart		

Please refer to your site documentation or your system administrator for information about the conditions to enable hooks on a specific system.

The following subsections illustrate a few cases of general interest for HPC from an end-user perspective.

Native MPI support (MPICH-based)

Sarus comes with a hook able to import native MPICH-based MPI implementations inside the container. This is useful in case the host system features a vendor-specific or high-performance MPI stack based on MPICH (e.g. Intel MPI, Cray MPT, MVAPICH) which is required to fully leverage a high-speed interconnect.

To take advantage of this feature, the MPI installed in the container (and dynamically linked to your application) needs to be *ABI-compatible* with the MPI on the host system. Taking as an example the Piz Daint Cray XC50 supercomputer at CSCS, to best meet the required ABI compatibility we recommend that the container application uses one of the following MPI implementations:

- [MPICH v3.1.4](#) (February 2015)
- [MVAPICH2 2.2](#) (September 2016)
- Intel MPI Library 2017 Update 1

The following is an example Dockerfile to create a Debian image with MPICH 3.1.4:

```
FROM debian:jessie

RUN apt-get update && apt-get install -y \
    build-essential \
    wget \
    --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

RUN wget -q http://www.mpich.org/static/downloads/3.1.4/mpich-3.1.4.tar.gz \
    && tar xf mpich-3.1.4.tar.gz \
    && cd mpich-3.1.4 \
    && ./configure --disable-fortran --enable-fast=all,03 --prefix=/usr \
    && make -j$(nproc) \
    && make install \
    && ldconfig \
    && cd .. \
    && rm -rf mpich-3.1.4 \
    && rm mpich-3.1.4.tar.gz
```

Note: Applications that are statically linked to MPI libraries will not work with the native MPI support provided by the hook.

Once the system administrator has configured the hook, containers with native MPI support can be launched by passing the `--mpi` option to the **sarus run** command, e.g.:

```
$ srun -N 16 -n 16 sarus run --mpi <repo name>/<image name> <mpi_application>
```

If multiple hooks are configured in the system, the `--mpi-type` option of **sarus run** can be used to choose a specific hook, for example:

```
$ srun -N 16 -n 16 sarus run --mpi-type=mpich-libfabric <repo name>/<image name> <mpi_
↪ application>
```

When multiple hooks are configured, the system administrator has the capability to define one of them as the default, which could be accessed through just the `--mpi` option. This provides compatibility with workflows which do not use the `--mpi-type` option for reasons of portability or legacy.

The MPI-related hooks configured on the system can be listed through the `--mpi` option of the **sarus hooks** command, for example:

```
$ sarus hooks --mpi
NAME                MPI TYPE
05-mpich-hook       ^mpich$ (default)
051-mpich-ofi-hook  ^mpich-libfabric$
```

If the value passed to the `sarus run --mpi-type` option matches one of the regular expressions under the `MPI TYPE` column, the corresponding hook is enabled. The `(default)` qualifier alongside an MPI type indicates the default MPI hook for the Sarus installation, which can be enabled just with the `sarus run --mpi` option.

NVIDIA GPU support

NVIDIA provides access to GPU devices and their driver stacks inside OCI containers through the [NVIDIA Container Toolkit hook](#).

When Sarus is configured to use this hook, the GPU devices to be made available inside the container can be selected by setting the `CUDA_VISIBLE_DEVICES` environment variable in the host system. Such action is often performed automatically by the workload manager or other site-specific software (e.g. the SLURM workload manager sets `CUDA_VISIBLE_DEVICES` when GPUs are requested via the [Generic Resource Scheduling plugin](#)). Be sure to check the setup provided by your computing site.

The container image needs to include a CUDA runtime that is suitable for both the target container applications and the available GPUs in the host system. One way to achieve this is by basing your image on one of the official Docker images provided by NVIDIA, i.e. the Dockerfile should start with a line like this:

```
FROM nvidia/cuda:8.0
```

Note: To check all the available CUDA images provided by NVIDIA, visit <https://hub.docker.com/r/nvidia/cuda/>

When developing GPU-accelerated images on your workstation, we recommend using [nvidia-docker](#) to run and test containers using an NVIDIA GPU.

SSH connection within containers

Sarus also comes with a hook which enables support for SSH connections within containers.

When Sarus is configured to use this hook, you must first run the command `sarus ssh-keygen` to generate the SSH keys that will be used by the SSH daemons and the SSH clients in the containers. It is sufficient to generate the keys just once, as they are persistent between sessions.

It is then possible to execute a container passing the `--ssh` option to **sarus run**, e.g. `sarus run --ssh <image> <command>`. Using the previously generated the SSH keys, the hook will instantiate an `sshd` and setup a custom `ssh` binary inside each container created with the same command.

Within a container spawned with the `--ssh` option, it is possible to connect into other containers by simply issuing the `ssh` command available in the default search `PATH`. E.g.:

```
ssh <hostname of other node>
```

The custom `ssh` binary will take care of using the proper keys and non-standard port in order to connect to the remote container.

When the `ssh` program is called without a command argument, it will open a login shell into the remote container. In this situation, the SSH hook attempts to reproduce the environment variables which were defined upon the launch of the remote container. The aim is to replicate the experience of actually accessing a shell in the container as it was created.

Warning: The SSH hook currently does not implement a poststop functionality and requires the use of a private PID namespace to cleanup the Dropbear daemon. Thus, the hook currently requires the use of a *private PID namespace* for the container. Thus, the `--ssh` option of **sarus run** implies `--pid=private`, and is incompatible with the use of `--pid=host`.

OpenMPI communication through SSH

The MPICH-based MPI hook described above does not support OpenMPI libraries. As an alternative, OpenMPI programs can communicate through SSH connections created by the SSH hook.

To run an OpenMPI program using the SSH hook, we need to manually provide a list of hosts and explicitly launch `mpirun` only on one node of the allocation. We can do so with the following commands:

```
salloc -C gpu -N4 -t5
srun hostname > $SCRATCH/hostfile
srun sarus run --ssh \
  --mount=src=/users,dst=/users,type=bind \
  --mount=src=$SCRATCH,dst=$SCRATCH,type=bind \
  ethscs/openmpi:3.1.3 \
  bash -c 'if [ $SLURM_PROCID -eq 0 ]; then mpirun --hostfile $SCRATCH/hostfile -
↪npernode 1 /openmpi-3.1.3/examples/hello_c; else sleep 10; fi'
```

Upon establishing a remote connection, the SSH hook provides a `$PATH` covering the most used default locations: `/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin`. If your OpenMPI installation uses a custom location, consider using an absolute path to `mpirun` and the `--prefix` option as advised in the [official OpenMPI FAQ](#).

Glibc replacement

Sarus's source code includes a hook able to inject glibc libraries from the host inside the container, overriding the glibc of the container.

This is useful when injecting some host resources (e.g. MPI libraries) into the container and said resources depend on a newer glibc than the container's one.

The host glibc stack to be injected is configured by the system administrator.

If Sarus is configured to use this hook, the glibc replacement can be activated by passing the `--glibc` option to **sarus run**. Since native MPI support is the most common occurrence of host resources injection, the hook is also implicitly activated when using the `--mpi` option.

Even when the hook is configured and activated, the glibc libraries in the container will only be replaced if the following conditions apply:

- the container's libraries are older than the host's libraries;
- host and container glibc libraries have the same soname and are ABI compatible.

Running MPI applications without the native MPI hook

The *MPI replacement mechanism* controlled by the `--mpi` option is not mandatory to run distributed applications in Sarus containers. It is possible to run containers using the MPI implementation embedded in the image, foregoing the performance of custom high-performance hardware.

This can be useful in a number of scenarios:

- the software stack in the container should not be altered in any way
- non-performance-critical testing
- impossibility to satisfy ABI compatibility for native hardware acceleration

Sarus can be launched as a normal MPI program, and the execution context will be propagated to the container application:

```
mpiexec -n <number of ranks> sarus run <image> <MPI application>
```

The important aspect to consider is that the process management system from the host must be able to communicate with the MPI libraries in the container.

MPICH-based MPI implementations by default use the [Hydra process manager](#) and the [PMI-2 interface](#) to communicate between processes. OpenMPI by default uses the [OpenRTE \(ORTE\) framework](#) and the [PMIx interface](#), but can be configured to support PMI-2.

Note: Additional information about the support provided by PMIx for containers and cross-version use cases can be found here: <https://openpmix.github.io/support/faq/how-does-pmix-work-with-containers>

As a general rule of thumb, corresponding MPI implementations (e.g. using an MPICH-compiled `mpiexec` on the host to launch Sarus containers featuring MPICH libraries) should work fine together. As mentioned previously, if an OpenMPI library in the container has been configured to support PMI, the container should also be able to communicate with an MPICH-compiled `mpiexec` from the host.

The following is a minimal Dockerfile example of building OpenMPI 4.0.2 with PMI-2 support on Ubuntu 18.04:

```
FROM ubuntu:18.04

RUN apt-get update && apt-get install -y \
    build-essential \
    ca-certificates \
    automake \
    autoconf \
    libpmi2-0-dev \
    wget \
    --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

RUN wget https://download.open-mpi.org/release/open-mpi/v4.0/openmpi-4.0.2.tar.gz \
```

(continues on next page)

(continued from previous page)

```

&& tar xf openmpi-4.0.2.tar.gz \
&& cd openmpi-4.0.2 \
&& ./configure --prefix=/usr --with-pmi=/usr/include/slurm-wlm --with-pmi-libdir=/
↳usr/lib/x86_64-linux-gnu CFLAGS=-I/usr/include/slurm-wlm \
&& make -j$(nproc) \
&& make install \
&& ldconfig \
&& cd .. \
&& rm -rf openmpi-4.0.2.tar.gz openmpi-4.0.2

```

When running under the Slurm workload manager, the process management interface can be selected with the `--mpi` option to `srun`. The following example shows how to run the *OSU point-to-point latency test* from the Sarus cookbook on CSCS' Piz Daint Cray XC50 system without native interconnect support:

```

$ srun -C gpu -N2 -t2 --mpi=pmi2 sarus run ethscs/mpich:ub1804_cuda92_mpi314_osu ./osu_
↳latency
###MPI-3.0
# OSU MPI Latency Test v5.6.1
# Size          Latency (us)
0                6.82
1                6.80
2                6.80
4                6.75
8                6.79
16               6.86
32               6.82
64               6.82
128              6.85
256              6.87
512              6.92
1024             9.77
2048             10.75
4096             11.32
8192             12.17
16384            14.08
32768            17.20
65536            29.05
131072           57.25
262144           83.84
524288           139.52
1048576          249.09
2097152          467.83
4194304          881.02

```

Notice that an `--mpi=pmi2` option was passed to `srun` but *not* to **sarus run**.

1.5.2 ABI compatibility and its implications

The *native MPICH hook* injects host libraries into the container, effectively replacing the compatible MPI implementation provided by the container image. This is done to achieve optimal performance: while container images are ideally *infrastructure agnostic* to maximize portability, the interconnect technologies found on HPC systems are often proprietary, requiring vendor-specific MPI software to be used at their full potential, or in some cases just to access the network hardware.

In order to allow the container applications to work seamlessly after the replacement, the container MPI libraries and the host MPI libraries must have compatible *application binary interfaces (ABI)*.

An ABI is an interface which defines interactions between software components at the machine code level, for example between an application executable and a shared object library. It is conceptually opposed to an *API (application programming interface)*, which instead defines interactions at the source code level.

If two libraries implement the same ABI, applications can be dynamically linked to either of them and still work seamlessly.

In 2013, the developers of several MPI implementations based on the MPICH library announced an *initiative* to maintain ABI compatibility between their implementations. At the time of writing, the software adhering to such collaboration are:

- MPICH v3.1 (Released February 2014)
- Intel® MPI Library v5.0 (Released June 2014)
- Cray MPT v7.0.0 (Released June 2014)
- MVAPICH2 2.0 (Release June 2014)
- Parastation MPI 5.1.7-1 (Released December 2016)
- RIKEN MPI 1.0 (Released August 2016)

Applications linked to one of the above libraries can work with any other implementation on the list, assuming the ABI versions of the libraries linked at compile time and linked at runtime are compatible.

The MPICH ABI Initiative establishes the *naming convention* for complying libraries. The ABI version of a binary can be determined by the string of numbers trailing the filename. For example, in the case of MPICH v3.1.1, which has ABI version 12.0.1:

```
$ ls -l /usr/local/lib
total 9356
lrwxrwxrwx. 1 root root      13 Mar 16 13:30 libfmpich.so -> libmpifort.so
-rw-r--r--. 1 root root 5262636 Mar 16 13:30 libmpi.a
-rwxr-xr-x. 1 root root    990 Mar 16 13:30 libmpi.la
lrwxrwxrwx. 1 root root      16 Mar 16 13:30 libmpi.so -> libmpi.so.12.0.1
lrwxrwxrwx. 1 root root      16 Mar 16 13:30 libmpi.so.12 -> libmpi.so.12.0.1
-rwxr-xr-x. 1 root root 2649152 Mar 16 13:30 libmpi.so.12.0.1
lrwxrwxrwx. 1 root root      9 Mar 16 13:30 libmpich.so -> libmpi.so
lrwxrwxrwx. 1 root root      12 Mar 16 13:30 libmpichcxx.so -> libmpicxx.so
lrwxrwxrwx. 1 root root      13 Mar 16 13:30 libmpichf90.so -> libmpifort.so
-rw-r--r--. 1 root root 305156 Mar 16 13:30 libmpicxx.a
-rwxr-xr-x. 1 root root    1036 Mar 16 13:30 libmpicxx.la
lrwxrwxrwx. 1 root root      19 Mar 16 13:30 libmpicxx.so -> libmpicxx.so.12.0.1
lrwxrwxrwx. 1 root root      19 Mar 16 13:30 libmpicxx.so.12 -> libmpicxx.so.12.0.1
-rwxr-xr-x. 1 root root 185336 Mar 16 13:30 libmpicxx.so.12.0.1
-rw-r--r--. 1 root root 789026 Mar 16 13:30 libmpifort.a
-rwxr-xr-x. 1 root root    1043 Mar 16 13:30 libmpifort.la
```

(continues on next page)

(continued from previous page)

```
lrwxrwxrwx. 1 root root      20 Mar 16 13:30 libmpifort.so -> libmpifort.so.12.0.1
lrwxrwxrwx. 1 root root      20 Mar 16 13:30 libmpifort.so.12 -> libmpifort.so.12.0.1
-rwxr-xr-x. 1 root root 364832 Mar 16 13:30 libmpifort.so.12.0.1
lrwxrwxrwx. 1 root root       9 Mar 16 13:30 libmpl.so -> libmpi.so
lrwxrwxrwx. 1 root root       9 Mar 16 13:30 libopa.so -> libmpi.so
drwxr-xr-x. 2 root root       9 Mar 16 13:30 pkgconfig
```

The initiative also defines [update rules](#) for the version strings: assuming a starting version of `libmpi.so.12.0.0`, updates to the library implementation which do not change the public interface would result in `libmpi.so.12.0.1`. Conversely, the addition of new features, without altering existing interface, would result in `libmpi.so.12.1.0`. Interface breaking changes would result in a bump to the first number in the string: `libmpi.so.13.0.0`.

Within the boundary of a compatible interface version, one could perform the following type of replacements:

- **Forward replacement:** replacing the library an application was originally linked to with a *newer* version (e.g. `12.0.x` is replaced by `12.1.x`). This is *safe*, since the newer version maintains the interface of the older one. An application can still successfully access the functions and symbols found at compilation time.
- **Backward replacement:** replacing the library an application was originally linked to with an *older* version (e.g. `12.1.x` is replaced by `12.0.x`). This is *NOT safe*, since the new features/functions which resulted in the minor version bump are not present in the older library. If an application tries to access functions which are only present in the newer version, this will result in an application failure (crash).

Armed with this knowledge, let's go full circle and return to our starting point. The *native MPICH hook* will mount host MPI libraries over corresponding libraries into the container to enable transparent native performance. Performing this action over a container image with an MPI implementation newer than the configured host one might prevent container applications from running. For this reason, before performing the mounts the hook verifies the ABI string compatibility between the involved libraries. If non-matching major version are detected, the hook will stop execution and return an error. If non-matching minor versions are detected when performing a backward replacement, the hook will print a warning but will proceed in the attempt to let the container application run.

The host MPI implementation to be injected is system-specific and is *configured by the system administrator*.

Important: Please refer to the documentation or contacts provided by your computing site to learn more about the compatible MPI versions on a given system.

1.5.3 Creating custom Docker images with CUDA

The base images provided by NVIDIA in [Docker Hub](#) only offer flavors based on Ubuntu and CentOS. If you want to build a CUDA-enabled image on a different distribution, we offer the following advice:

Installing the CUDA Toolkit

- **Package manager installer:** repository installers (to be used through the system package manager) are available for Fedora, OpenSUSE, RHEL and SLES (and also for Ubuntu and CentOS, if you don't want to use NVIDIA's images). For detailed installation instructions, refer to the official CUDA Toolkit Documentation (<https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#package-manager-installation>).

Please note that installing the default package options (e.g. `cuda` or `cuda-toolkit`) will add a significant amount of resources to your image (more than 1GB). Significant size savings can be achieved by selectively installing only the packages with the parts of the Toolkit that you need. Please consult your system package manager documentation in order to list the available packages inside the CUDA repositories.

- **Runfile install:** some distributions are not covered by the CUDA package manager and have to perform the installation through the standalone runfile installer. One such case is Debian, which is also used as base for several official Docker Hub images (e.g. `Python`). For detailed installation instructions, refer to the official CUDA Toolkit Documentation (<https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#runfile>). We advise to supply the `--silent` and `--toolkit` options to the installer, to avoid installing the CUDA drivers as well.

The standalone installer will add a significant amount of resources to your image, including documentation and SDK samples. If you are really determined to reduce the size of your image, you can selectively `rm -rf` the parts of the Toolkit that you don't need, but be careful about not deleting libraries and tools that may be used by applications in the container!

Controlling the NVIDIA Container Runtime

The NVIDIA Container Runtime (at the heart of `nvidia-docker v2`) is controlled through several environment variables. These can be part of the container image or be set by **docker run** with the `-e`, `--env` or `--env-file` options (please refer to the [official documentation](#) for the full syntax of these flags). A brief description of the most useful variables follows:

- **NVIDIA_VISIBLE_DEVICES:** This variable controls which GPUs will be made accessible inside the container. The values can be a list of indexes (e.g. `0,1,2`), `all`, or `none`. If the variable is empty or unset, the runtime will behave as default Docker.
- **NVIDIA_DRIVER_CAPABILITIES:** This option controls which driver libraries and binaries will be mounted inside the container. It accepts as values a comma-separated list of driver features (e.g. `utility,compute`) or the string `all` (for all driver capabilities). An empty or unset variable will default to minimal driver capabilities (`utility`).
- **NVIDIA_REQUIRE_CUDA:** A logical expression to define a constraint on the CUDA driver version required by the container (e.g. `"cuda>=8.0"` will require a driver supporting the CUDA Toolkit version 8.0 and later). Multiple constraints can be expressed in a single environment variable: space-separated constraints are ORed, comma-separated constraints are ANDed.

A full list of the environment variables looked up by the NVIDIA Container Runtime and their possible values is available [here](#).

As an example, in order to work correctly with the NVIDIA runtime, an image including a CUDA 8.0 application could feature these instructions in its Dockerfile:

```
ENV NVIDIA_VISIBLE_DEVICES all
ENV NVIDIA_DRIVER_CAPABILITIES compute,utility
ENV NVIDIA_REQUIRE_CUDA "cuda>=8.0"
```

Alternatively, the variables can be specified (or even overwritten) from the command line:

```
$ docker run --runtime=nvidia -e NVIDIA_VISIBLE_DEVICES=all -e NVIDIA_DRIVER_
↳CAPABILITIES=compute,utility --rm nvidia/cuda nvidia-smi
```

The official `nvidia/cuda` images already include a set of these environment variables. This means that all the Dockerfiles that do a `FROM nvidia/cuda` will automatically inherit them and thus will work seamlessly with the NVIDIA Container Runtime.

1.6 Developer documentation

1.6.1 Class diagram

The class diagram below illustrates the internal architecture of Sarus. Please note that here the goal is to provide an overview of the system's architecture and the class diagram is just a simplified representation of the system: many classes and details of minor importance were omitted.

1.6.2 Running unit and integration tests

Unit tests

In order to run Sarus unit tests, it is advised to disable the security checks in the *sarus.json* file.

Disabling security checks prevents some tests from failing because some files (e.g. artifacts to test JSON parsing and validation) are not root-owned and located in root-owned directories. The unit tests for security checks individually re-enable the functionality to reliably verify its effectiveness.

The unit tests are written using the `CppUTest` framework, which the build scripts are able to retrieve and compile automatically.

The tests are run from the build directory with the help of `CTest`, the test driver program from the `CMake` suite. We differentiate between normal tests and tests that require root privileges (e.g. those performing filesystem mounts). The latter are identified by the suffix `AsRoot` in the name of the test executable.

```
# Run normal unit tests
$ CTEST_OUTPUT_ON_FAILURE=1 ctest --exclude-regex 'AsRoot'

# Run 'AsRoot' unit tests
$ sudo CTEST_OUTPUT_ON_FAILURE=1 ctest --tests-regex 'AsRoot'
```

Generating coverage data

If the build was configured with the `CMake` toolchain file `gcc-gcov.cmake`, the unit tests executables automatically generate `gcov` files with raw coverage data. We can process and summarize these data using `gcov` and the `gcovr` utility:

Note: To yield reliable results, it is advised to collect unit test coverage data only when the build has been performed in "Debug" configuration.

```
# Assuming that we are in the project's root directory and Sarus was built in the
# 'build' subdirectory
root_dir=$(pwd)
build_dir=$(pwd)/build
mkdir ${build_dir}/gcov
cd ${build_dir}/gcov
gcov --preserve-paths $(find ${build_dir}/src -name "*.gcno" |grep -v test |tr '\n' ' ')
gcovr -r ${root_dir}/src -k -g --object-directory ${build_dir}/gcov
```

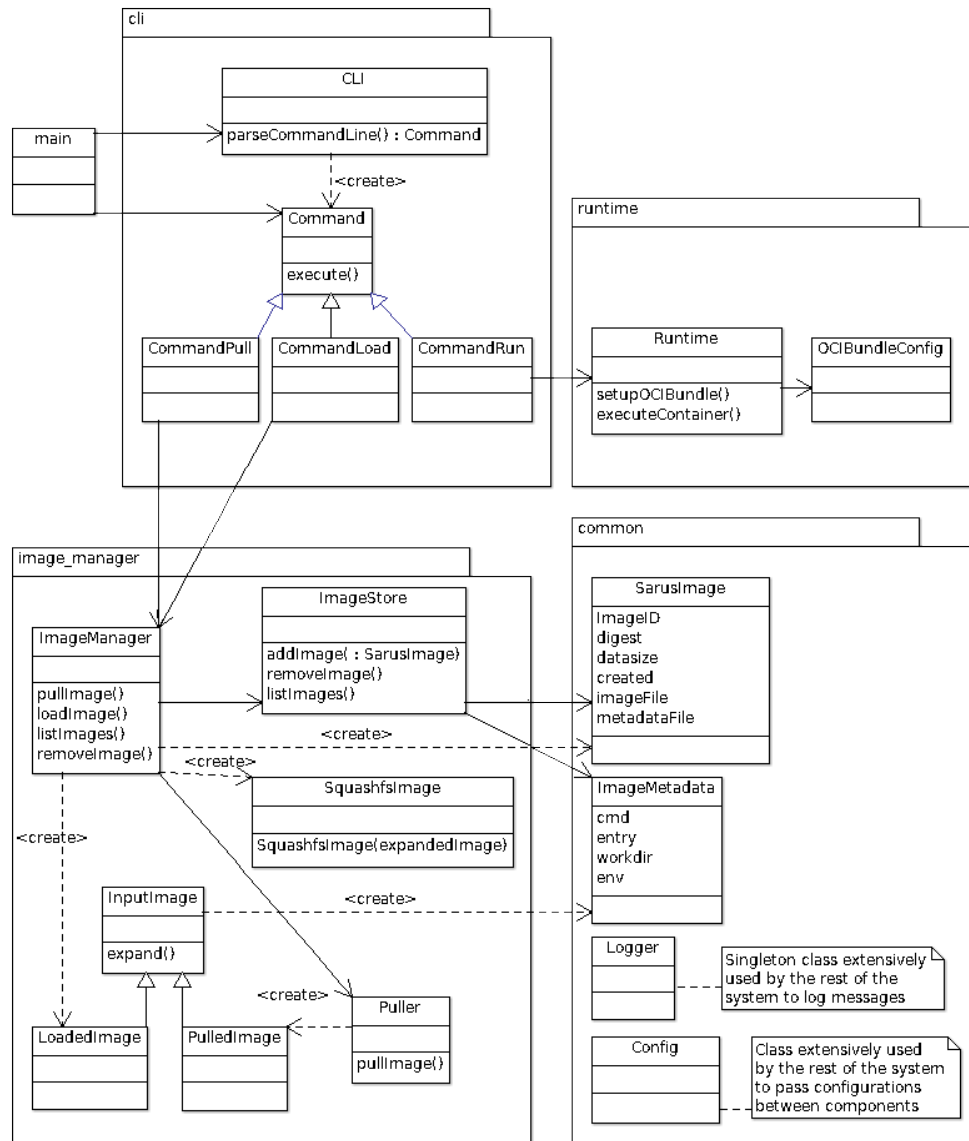


Fig. 4: Sarus class diagram

Integration tests

Integration tests use Python 3 and the packages indicated in the [Requirements page](#). Sarus must be correctly installed and configured on the system in order to successfully perform integration testing. Before running the tests, we need to re-target the centralized repository to a location that is writable by the current user (this is not necessary if running integration tests as root):

```
$ mkdir -p ~/sarus-centralized-repository
$ sudo sed -i -e 's@"centralizedRepositoryDir": "*"@"centralizedRepositoryDir": "/home/
↳ docker/sarus-centralized-repository"@' /opt/sarus/etc/sarus.json
```

Note: Integration tests are not exposed to the risk of failing when runtime security checks are enabled, like unit tests are. To test a configuration more similar to a production deployment, re-enable security checks in the *sarus.json* file.

We can run the tests from the parent directory of the related Python scripts:

```
$ cd <sarus project root dir>/CI/src
$ PYTHONPATH=$(pwd):$PYTHONPATH CMAKE_INSTALL_PREFIX=/opt/sarus/ pytest -v -m 'not asroot
↳ ' integration_tests/
$ sudo PYTHONPATH=$(pwd):$PYTHONPATH CMAKE_INSTALL_PREFIX=/opt/sarus/ pytest -v -m
↳ asroot integration_tests/
```

1.6.3 SSH Hook

Introduction

This document is meant to be read by developers in order to understand how Sarus provides SSH within containers.

Overview of the user interface

First of all, the user needs to run the command `sarus ssh-keygen`. This command generates the SSH keys that will be used by the SSH daemons as well as the SSH clients in the containers.

Then the user can execute a container passing the `--ssh` option, e.g. `sarus run --ssh <image> <command>`. This will first check that the user previously generated the SSH keys, and then will instantiate an SSH daemon per container and setup a custom `ssh` executable inside the containers.

Within a container spawned with the `--ssh` option, the user can `ssh` into other containers by simply issuing the `ssh` command available in the default search `PATH`. E.g.

```
ssh <hostname of other node>
```

The custom `ssh` executable will take care of using the proper keys and non-standard port in order to connect to the remote container.

Configuration by the system administrator

The following is an example of OCI hook JSON configuration file enabling the SSH hook:

```
{
  "version": "1.0.0",
  "hook": {
    "path": "/opt/sarus/bin/ssh_hook",
    "env": [
      "HOOK_BASE_DIR=/home",
      "PASSWD_FILE=/opt/sarus/etc/passwd",
      "DROPBEAR_DIR=/opt/sarus/dropbear",
      "SERVER_PORT=15263"
    ],
    "args": [
      "ssh_hook",
      "start-ssh-daemon"
    ]
  },
  "when": {
    "annotations": {
      "^com.hooks.ssh.enabled$": "^true$"
    }
  },
  "stages": ["prestart"]
}
```

Architecture

Most of the ssh-related logic is encapsulated in the SSH hook. The SSH hook is an executable binary that performs different ssh-related operations depending on the CLI command that receives as parameter from the container engine.

The nitty gritty

The custom SSH software

The SSH hook uses a custom statically-linked version of [Dropbear](#). At build time, if the CMake's parameter `ENABLE_SSH=TRUE` is specified, the custom Dropbear is built and installed under the Sarus's installation directory. Dropbear is built according to the instructions provided in the `<sarus source dir>/dep/build_dropbear.sh` script. Such script also creates a `localoptions.h` header, overriding some of the default compile-time options of Dropbear.

How the SSH keys are generated

When the command `sarus ssh-keygen` is issued, the command object `cli::CommandSshkeygen` gets executed which in turn executes the SSH hook with the `keygen` CLI argument.

The hook performs the following operations:

1. Read from the environment variables the hook base directory.
2. Read from the environment variables the location of the `passwd` file.
3. Get the username from the `passwd` file and use it to determine the user's hook directory (where the SSH keys are stored).
4. Read from the environment variables the location of the custom SSH software.
5. Execute the program `dropbearkey` to generate two keys in the user's hook directory, using the ECDSA algorithm. One key (`dropbear_ecdsa_host_key`) will be used by the SSH daemon, the other key (`id_dropbear`) will be used by the SSH client.

How the existence of the SSH keys is checked

When the command `sarus run --ssh <image> <command>` is issued, the command object `cli::CommandRun` gets executed which in turn executes the SSH hook with the `check-user-has-sshkeys` CLI argument.

The hook performs the following operations:

1. Read from the environment variables the hook base directory.
2. Read from the environment variables the location of the `passwd` file.
3. Get the username from the `passwd` file and use it to determine the user's hook directory (where the SSH keys are stored).
4. Check that the user's hook directory contains the SSH keys.

How the SSH daemon and SSH client are setup in the container

When the command `"sarus run --ssh <image> <command>"` is issued, Sarus sets up the OCI bundle and executes `runc`. Then `runc` executes the OCI prestart hooks specified in `sarus.json`. The system administrator should have specified the SSH hook with the `"start-ssh-daemon"` CLI argument.

The hook performs the following operations:

1. Read from the environment variables the hook base directory.
2. Read from the environment variables the location of the `passwd` file.
3. Read from the environment variables the location of the custom SSH software.
4. Read from the environment variables the port number to be used by the SSH daemon.
5. Read from stdin the container's state as defined in the OCI specification.
6. Enter the container's mount namespaces in order to access the container's OCI bundle.
7. Enter the container's pid namespace in order to start the `sshd` process inside the container.
8. Read the user's UID/GID from the OCI bundle's `config.json`
9. Get the username from the `passwd` file supplied by step 2.
10. Determine the user's hook directory (where the SSH keys are stored) using data from steps 1. and 9.

11. Determine the location of the SSH keys in the container. See *Determining the location of the SSH keys inside the container*.
12. Copy the custom SSH software into the container (SSH daemon, SSH client).
13. Setup the location for SSH keys into the container. If the path from 11. does not exist, it is created. If the user mounted their home dir from the host, future actions from the hook could alter the host's home dir. For this reason, the hook performs an overlay mount over the keys location. This allows the hook to copy the Dropbear keys into the container without tampering with the user's original `~/.ssh`.
14. Copy the Dropbear SSH keys into the container.
15. Patch the container's `/etc/passwd` if it does not feature a user shell which exists inside the container. This check is necessary since with Sarus the `passwd` file is copied from the host system.
16. Create a shell script exporting the environment from the OCI bundle. See *Reproducing the remote environment in a login shell*
17. Create a shell script in `/etc/profile.d`. See *Reproducing the remote environment in a login shell*
18. Chroot to the container, drop privileges, start the SSH daemon inside the container.
19. Create a shell script `/usr/bin/ssh` which transparently uses the Dropbear SSH client (**dbclient**) and the proper keys and port number to establish SSH sessions.

Determining the location of the SSH keys inside the container

Dropbear looks inside `~/.ssh` to find the client key, the ``authorized_keys and known_hosts files. The location of the home directory is determined from the running user's entry in the container's /etc/passwd file.`

The SSH hook reads the container's `passwd` file as well and extracts the location of the home directory from that file to ensure the correct operation of Dropbear. At the moment of copying the SSH keys into the container, the `~/.ssh` path is created if it does not exist. This prevents possible issues with Sarus, which replaces the `/etc/passwd` of the container (see point 3. of *Root filesystem*): such file might indicate a user home path which does not initially exist inside the container.

The possibility for a working user to not have a sane home directory inside `/etc/passwd` has also to be taken into account: if the home directory field is empty, for example, the SSH software will still log the user into `/`. Debian-based distributions use `/nonexistent` for users which are not supposed to have a home dir. Red Hat and Alpine assign the root path as home even to the `nobody` user. Dropbear is currently not able to handle these corner cases: if the home field is empty, Dropbear will fail to find its key even if they are created under `/.ssh`. On the other hand, if the `passwd` file specifies a `/nonexistent` home directory field, Dropbear will still look for `/nonexistent/.ssh`, which is a location that is not meant to exist, even if the SSH hook could create it.

Because of the behavior described above, in case of empty or `/nonexistent` home directory field in `/etc/passwd`, the SSH hook exits with an error.

Reproducing the remote environment in a login shell

When opening a login shell into a remote container, the SSH hook also attempts to reproduce an environment that is as close as possible to the one defined in the OCI bundle of the remote container.

This feature is implemented through 2 actions which take place when the hook is executed by the OCI runtime, before the container process is started:

1. A shell script exporting all the environment variables defined in the OCI bundle is created in `/opt/oci-hooks/dropbear/environment`.

2. A shell script is created in `/etc/profile.d/ssh-hook.sh`. This script will source `/opt/oci-hooks/dropbear/environment` (thus replicating the environment from the OCI bundle) if the `SSH_CONNECTION` environment variable is defined:

```
#!/bin/sh
if [ \"$SSH_CONNECTION\" ]; then
    . /opt/oci-hooks/dropbear/environment
fi
```

Dropbear sets the `SSH_CONNECTION` variable upon connecting to a remote host. If the `ssh` program is called without a command to execute on the remote, it will start a login shell. As it is custom, login shells will source `/etc/profile`, which in turn will source every script under `/etc/profile.d`.

Printing debug messages from Dropbear

Dropbear gives the possibility to use the `-v` command line option on both the server and client programs (**dropbear** and **dbclient** respectively) to print debug messages about its internal workings. The `-v` option is only available if the `DEBUG_TRACE` symbol is defined as part of the compile-time options in `localoptions.h`:

```
/* Include verbose debug output, enabled with -v at runtime. */
#define DEBUG_TRACE 1
```

The `<saros source dir>/dep/build_dropbear.sh` script defines `DEBUG_TRACE`, so in a container launched using the SSH hook it is immediately possible to print debug messages by passing `-v` to the SSH client wrapper script:

```
$ ssh -v <remote container host> <command>
```

To obtain debug messages from the Dropbear server, a possibility is to launch an container with an interactive shell, kill the **dropbear** process started by the SSH hook and re-launch the server adding the `-v` option:

```
$ /opt/oci-hooks/dropbear/bin/dropbear -E -r <user home dir>/.ssh/dropbear_ecdsa_host_
↪key -p <port number from the hook config> -v
```

Dropbear's server and client programs can print even more detailed messages if the `DROPBEAR_TRACE2` environment variable is defined:

```
$ export DROPBEAR_TRACE2=1
```

1.7 Sarus Cookbook

This cookbook is a collection of documented use case examples of the Sarus container runtime for HPC. Each example is accompanied by material and information sufficient to reproduce it. In this sense, the examples also serve as recipes demonstrating how to build containers with features like MPI and CUDA, and how to leverage those features from Sarus. The examples are ordered by increasing complexity, providing a smooth learning curve.

1.7.1 Test Setup

The examples and performance measurements reported in this document were carried out on [Piz Daint](#), a hybrid Cray XC50/XC40 system in production at the Swiss National Supercomputing Centre (CSCS) in Lugano, Switzerland. The system compute nodes are connected by the Cray Aries interconnect under a Dragonfly topology, notably providing users access to hybrid CPU-GPU nodes. Hybrid nodes are equipped with an Intel Xeon E5-2690v3 processor, 64 GB of RAM, and a single NVIDIA Tesla P100 GPU with 16 GB of memory. The software environment on Piz Daint at the time of writing is the Cray Linux Environment 6.0.UP07 (CLE 6.0) using Environment Modules to provide access to compilers, tools, and applications. The default versions for the NVIDIA CUDA and MPI software stacks are, respectively, CUDA version 9.1, and Cray MPT version 7.7.2.

We install and configure Sarus on Piz Daint to use the native MPI and NVIDIA Container Runtime hooks, and to mount the container images from a Lustre parallel filesystem.

Performance measurements

In tests comparing native and container performance numbers, each data point presents the average and standard deviation of 50 runs, to produce statistically relevant results, unless otherwise noted. For a given application, all repetitions at each node count for both native and container execution were performed on the same allocated set of nodes.

1.7.2 CUDA N-body

A fast n-body simulation is included as part of the [CUDA Software Development Kit samples](#). The CUDA n-body sample code simulates the gravitational interaction and motion of a group of bodies. The code is written with CUDA and C and can make efficient use of multiple GPUs to calculate all-pairs gravitational interactions. More details of the implementation can be found in this article by Lars Nyland et al.: [Fast N-Body Simulation with CUDA](#).

We use this sample code to show that Sarus is able to leverage the NVIDIA Container Runtime hook in order to provide containers with native performance from NVIDIA GPUs present in the host system.

Test Case

For this test case, we run the code with $n = 200,000$ bodies using double-precision floating-point arithmetic on 1 Piz Daint compute node, featuring a single Tesla P100 GPU.

Running the container

We run the container using the Slurm Workload Manager and Sarus:

```
srun -Cgpu -N1 -t1 \  
sarus run ethscs/cudasamples:9.2 \  
/usr/local/cuda/samples/bin/x86_64/linux/release/nbody \  
-benchmark -fp64 -numbodies=200000
```

A typical output will look like:

```
Run "nbody -benchmark [-numbodies=<numBodies>]" to measure performance.  
-fullscreen      (run n-body simulation in fullscreen mode)  
-fp64            (use double precision floating point values for simulation)  
-hostmem         (stores simulation data in host memory)  
-benchmark       (run benchmark to measure performance)  
-numbodies=<N>   (number of bodies (>= 1) to run in simulation)
```

(continues on next page)

(continued from previous page)

```
-device=<d>      (where d=0,1,2.... for the CUDA device to use)
-numdevices=<i>  (where i=(number of CUDA devices > 0) to use for simulation)
-compare        (compares simulation results running once on the default
                  GPU and once on the CPU)
-cpu            (run n-body simulation on the CPU)
-tipsy=<file.bin> (load a tipsy model file for simulation)
```

NOTE: The CUDA Samples are not meant for performance measurements.
Results may vary when GPU Boost is enabled.

```
> Windowed mode
> Simulation data stored in video memory
> Double precision floating point simulation
> 1 Devices used for simulation
GPU Device 0: "Tesla P100-PCIE-16GB" with compute capability 6.0

> Compute 6.0 CUDA device: [Tesla P100-PCIE-16GB]
Warning: "number of bodies" specified 200000 is not a multiple of 256.
Rounding up to the nearest multiple: 200192.
200192 bodies, total time for 10 iterations: 3927.009 ms
= 102.054 billion interactions per second
= 3061.631 double-precision GFLOP/s at 30 flops per interaction
```

Running the native Application

We compile and run the same code on Piz Daint using a similar Cuda Toolkit version (cudatoolkit/9.2).

Container image and Dockerfile

The container image *ethscs/cudasamples:9.2* (based on Nvidia cuda/9.2) used for this test case can be pulled from CSCS DockerHub or be rebuilt with this Dockerfile:

```
1 FROM nvidia/cuda:9.2-devel
2
3 RUN apt-get update && apt-get install -y --no-install-recommends \
4     cuda-samples-$CUDA_PKG_VERSION && \
5     rm -rf /var/lib/apt/lists/*
6
7 RUN (cd /usr/local/cuda/samples/1_Uutilities/bandwidthTest && make)
8 RUN (cd /usr/local/cuda/samples/1_Uutilities/deviceQuery && make)
9 RUN (cd /usr/local/cuda/samples/1_Uutilities/deviceQueryDrv && make)
10 RUN (cd /usr/local/cuda/samples/1_Uutilities/p2pBandwidthLatencyTest && make)
11 RUN (cd /usr/local/cuda/samples/1_Uutilities/topologyQuery && make)
12 RUN (cd /usr/local/cuda/samples/5_Simulations/nbody && make)
13
14 CMD ["/usr/local/cuda/samples/1_Uutilities/deviceQuery/deviceQuery"]
```

Required OCI hooks

- NVIDIA Container Runtime hook

Benchmarking results

We report the gigaflops per second performance attained by the two applications in the following table:

	Average	Std. deviation
Native	3059.34	5.30
Container	3058.91	6.29

The results show that containers deployed with Sarus and the NVIDIA Container Runtime hook can achieve the same performance of the natively built CUDA application, both in terms of average value and variability.

1.7.3 OSU Micro benchmarks

The OSU Micro Benchmarks ([OMB](#)) are a widely used suite of benchmarks for measuring and evaluating the performance of MPI operations for point-to-point, multi-pair, and collective communications. These benchmarks are often used for comparing different MPI implementations and the underlying network interconnect.

We use OMB to show that Sarus is able to provide the same native MPI high performance to containerized applications when using the native MPICH hook. As indicated in the documentation for the hook, the only conditions required are:

- The MPI installed in the container image must comply to the requirements of the [MPICH ABI Compatibility Initiative](#). ABI compatibility and its implications are further discussed [here](#).
- The application in the container image must be dynamically linked with the MPI libraries.

Test cases

Latency

The `osu_latency` benchmark measures the min, max and the average latency of a ping-pong communication between a sender and a receiver where the sender sends a message and waits for the reply from the receiver. The messages are sent repeatedly for a variety of data sizes in order to report the average one-way latency. This test allows us to observe any possible overhead from enabling the MPI support provided by Sarus.

All-to-all

The `osu_alltoall` benchmark measures the min, max and the average latency of the `MPI_Alltoall` blocking collective operation across N processes, for various message lengths, over a large number of iterations. In the default version, this benchmark report the average latency for each message length up to 1MB. We run this benchmark from a minimum of 2 nodes up to 128 nodes, increasing the node count in powers of two.

Running the container

We run the container using the Slurm Workload Manager and Sarus.

Latency

```
sarus pull ethscs/mvapich:ub1804_cuda92_mpi22_osu
srun -C gpu -N2 -t2 \
  sarus run --mpi ethscs/mvapich:ub1804_cuda92_mpi22_osu \
  /usr/local/libexec/osu-micro-benchmarks/mpi/pt2pt/osu_latency
```

A typical output looks like:

```
# OSU MPI Latency Test v5.3.2
# Size          Latency (us)
0                1.11
1                1.11
2                1.09
4                1.09
8                1.09
16               1.10
32               1.09
64               1.10
128              1.11
256              1.12
512              1.15
1024             1.39
2048             1.67
4096             2.27
8192             4.21
16384            5.12
32768            6.73
65536            10.07
131072           16.69
262144           29.96
524288           56.45
1048576          109.28
2097152          216.29
4194304          431.85
```

Since the Dockerfiles use the WORKDIR instruction to set a default working directory, we can use that to simplify the terminal command:

```
srun -C gpu -N2 -t2 \
  sarus run --mpi ethscs/osu-mb:5.3.2-mpich3.1.4 \
  ./osu_latency
```

All-to-all

```
srun -C gpu -N2 -t2 \  
  sarus run --mpi ethscs/osu-mb:5.3.2-mpich3.1.4 \  
  ../collective/osu_alltoall
```

A typical output looks like:

```
# OSU MPI All-to-All Personalized Exchange Latency Test v5.3.2  
# Size      Avg Latency(us)  
1           5.46  
2           5.27  
4           5.22  
8           5.21  
16          5.18  
32          5.18  
64          5.17  
128         11.35  
256         11.64  
512         11.72  
1024        12.03  
2048        12.87  
4096        14.52  
8192        15.77  
16384       19.78  
32768       28.89  
65536       49.38  
131072      96.64  
262144     183.23  
524288     363.35  
1048576    733.93
```

Running the native application

We compile the OSU micro benchmark suite natively using the Cray Programming Environment (PrgEnv-cray) and linking against the optimized Cray MPI (cray-mpich) libraries.

Container images and Dockerfiles

We built the OSU benchmarks on top of several images containing MPI, in order to demonstrate the effectiveness of the MPI hook regardless of the ABI-compatible MPI implementation present in the images:

MPICH

The container image `ethcscs/mpich:ub1804_cuda92_mpi314_osu` (based on `mpich/3.1.4`) used for this test case can be pulled from CSCS [DockerHub](#) or be rebuilt with this [Dockerfile](#).

MVAPICH

The container image `ethcscs/mvapich:ub1804_cuda92_mpi22_osu` (based on `mvapich/2.2`) used for this test case can be pulled from CSCS [DockerHub](#) or be rebuilt with this [Dockerfile](#). On the Cray, the supported Cray MPICH ABI is 12.0 (`mvapich>2.2` requires ABI/12.1 hence is not currently supported).

OpenMPI

As OpenMPI is not part of the MPICH ABI Compatibility Initiative, `sarus run --mpi` with OpenMPI is not supported. Documentation can be found on this dedicated page: [OpenMPI with SSH launcher](#).

Intel MPI

Because the Intel MPI license limits general redistribution of the software, we do not share the Docker image `ethcscs/intelmpi` used for this test case. Provided the Intel installation files (such as archive and license file) are available locally on your computer, you could build your own image with this example [Dockerfile](#).

Required OCI hooks

- Native MPI hook (MPICH-based)

Benchmarking results

Latency

Consider now the following Figure that compares the average and standard deviation of the `osu_latency` test results for the four tested configurations. It can be observed that Sarus with the native MPI hook allows containers to transparently access the accelerated networking hardware on Piz Daint and achieve the same performance as the natively built test.

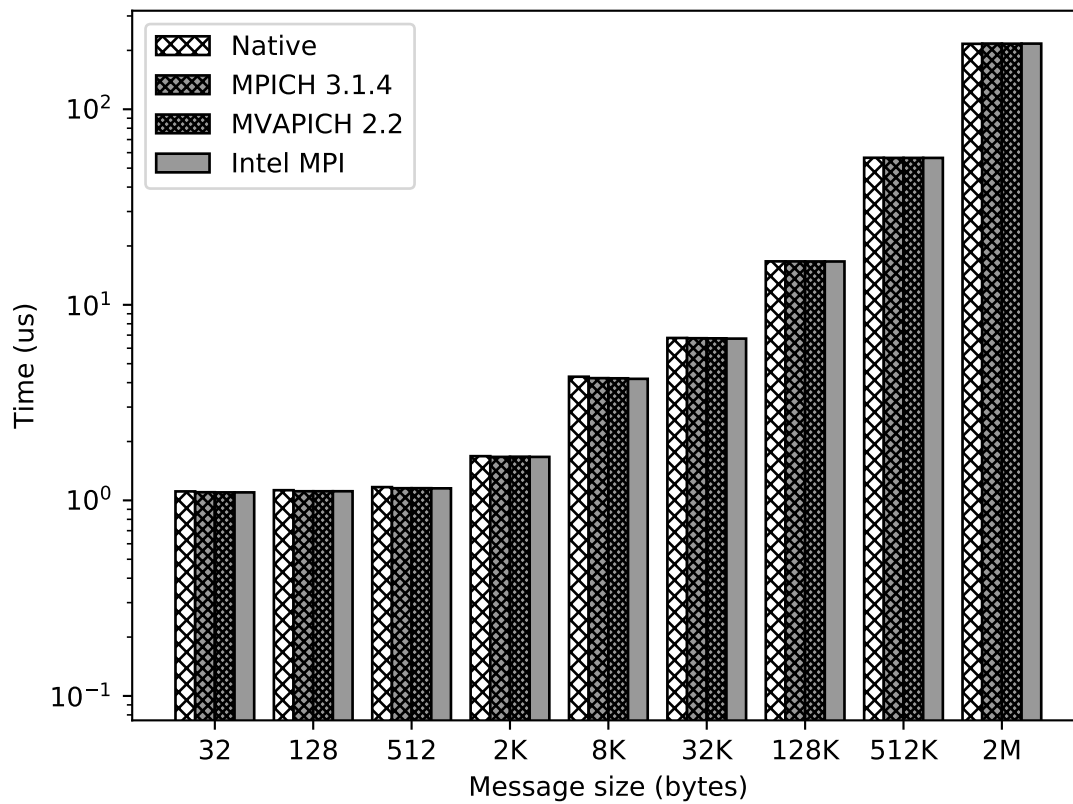


Fig. 5: Results of the OSU Latency benchmark for the native MPI and three different containers with ABI-compliant MPI libraries. The MPI in the container is replaced at runtime by the native MPICH MPI hook used by Sarvus.

All-to-all

We run the `osu_alltoall` benchmark only for two applications: native and container with MPICH 3.1.4. We collect latency values for 1kB, 32kB, 65kB and 1MB message sizes, computing averages and standard deviation. The results are displayed in the following Figure:

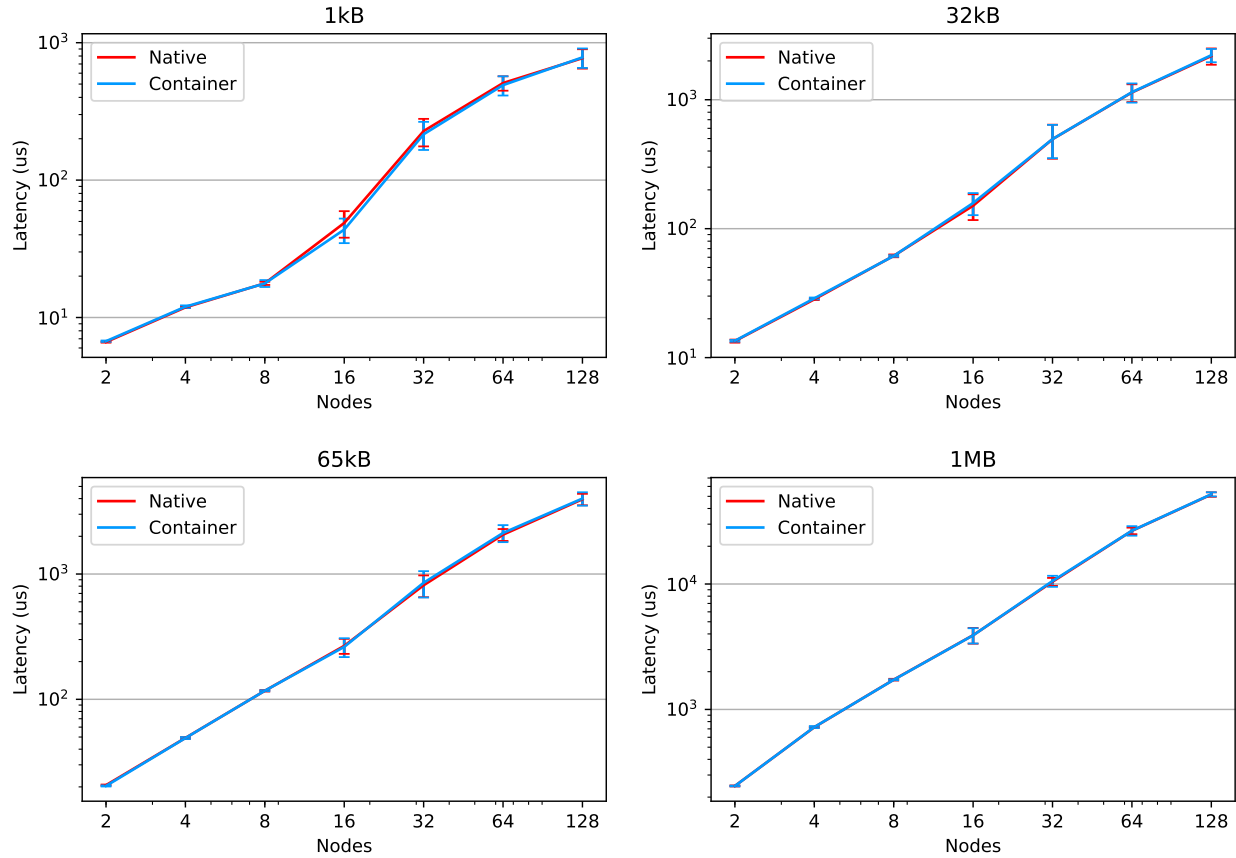


Fig. 6: Results of the OSU All-to-all benchmark for the native MPI and MPICH 3.1.4 container. The MPI in the container is replaced at runtime by the native MPICH MPI hook used by Sarús.

We observe that the results from the container are very close to the native results, for both average values and variability, across the node counts and message sizes. The average value of the native benchmark for 1kB message size at 16 nodes is slightly higher than the one computed for the container benchmark.

It is worthy to note that the results of this benchmark are heavily influenced by the topology of the tested set of nodes, especially regarding their variability. This means that other tests using the same node counts may achieve significantly different results. It also implies that results at different node counts are only indicative and not directly relatable, since we did not allocate the same set of nodes for all node counts.

1.7.4 NVIDIA GPUDirect RDMA

GPUDirect is a technology that enables direct RDMA to and from GPU memory. This means that multiple GPUs can directly read and write CUDA host and device memory, without resorting to the use of host memory or the CPU, resulting in significant data transfer performance improvements.

We will show here that Sarus is able to leverage the GPUDirect technology.

Test case

This sample C++ code performs an MPI_Allgather operation using CUDA device memory and GPUDirect. If the operation is carried out successfully, the program prints a success message to standard output.

Running the container

Before running this code with Sarus, two environment variables must be set: `MPICH_RDMA_ENABLED_CUDA` and `LD_PRELOAD`

`MPICH_RDMA_ENABLED_CUDA`: allows the MPI application to pass GPU pointers directly to point-to-point and collective communication functions, as well as blocking collective communication functions.

`LD_PRELOAD`: allows to load the specified cuda library from the compute node before all others.

This can be done by passing a string command to bash:

```
srun -C gpu -N4 -t2 sarus run --mpi \
    ethcses/mpich:ub1804_cuda92_mpi314_gpudirect-all_gather
    bash -c 'MPICH_RDMA_ENABLED_CUDA=1 LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libcuda.so ./
    ↪all_gather'
```

A successful output looks like:

```
Success!
```

Running the native application

```
export MPICH_RDMA_ENABLED_CUDA=1
srun -C gpu -N4 -t2 ./all_gather
```

A successful output looks like:

```
Success!
```

Container image and Dockerfile

The container image (based on `cuda/9.2` and `mpich/3.1.4`) used for this test case can be pulled from CSCS [DockerHub](#) or be rebuilt with this Dockerfile:

```

1 # docker build -f Dockerfile -t \
2 #   ethcscs/mpich:ub1804_cuda92_mpi314_gpudirect-all_gather .
3 FROM ethcscs/mpich:ub1804_cuda92_mpi314
4
5 COPY all_gather.cpp /opt/mpi_gpudirect/all_gather.cpp
6 WORKDIR /opt/mpi_gpudirect
7 RUN mpicxx -g all_gather.cpp -o all_gather -I/usr/local/cuda/include -L/usr/local/cuda/
   ↪ lib64 -lcudart

```

Required OCI hooks

- NVIDIA Container Runtime hook
- Native MPI hook (MPICH-based)

1.7.5 OpenMPI with SSH launcher

OpenMPI is an MPI implementation with widespread adoption and support throughout the HPC community, and is featured in several official Docker images as well.

The substantial differences with MPICH-based MPI implementations make the native MPICH hook provided by Sarus ineffective on container images using OpenMPI.

This example will showcase how to run an OpenMPI container application by using an SSH launcher and the SSH Hook.

Test case

We use the "Hello world in C" program included with the OpenMPI codebase in the `examples/hello_c.c` file. We do not measure performance with this test and we do not make comparisons with a native implementation.

The program code at the time of writing follows for reference:

```

/*
 * Copyright (c) 2004-2006 The Trustees of Indiana University and Indiana
 *                           University Research and Technology
 *                           Corporation. All rights reserved.
 * Copyright (c) 2006      Cisco Systems, Inc. All rights reserved.
 *
 * Sample MPI "hello world" application in C
 */

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

int main(int argc, char* argv[])
{
    int rank, size, len;

```

(continues on next page)

(continued from previous page)

```
char version[MPI_MAX_LIBRARY_VERSION_STRING];

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Get_library_version(version, &len);
printf("Hello, world, I am %d of %d, (%s, %d)\n",
      rank, size, version, len);
MPI_Finalize();

return 0;
}
```

Container image and Dockerfile

We start from the official image for Debian 8 "Jessie", install the required build tools, OpenMPI 3.1.3, and compile the example code using the MPI compiler.

```
FROM debian:jessie

RUN apt-get update \
    && apt-get install -y ca-certificates \
        file \
        g++ \
        gcc \
        gfortran \
        make \
        gdb \
        strace \
        realpath \
        wget \
        --no-install-recommends

RUN wget -q https://download.open-mpi.org/release/open-mpi/v3.1/openmpi-3.1.3.tar.gz \
    && tar xf openmpi-3.1.3.tar.gz \
    && cd openmpi-3.1.3 \
    && ./configure --with-slurm=no --prefix=/usr/local \
    && make all install
#./configure --with-slurm=no --prefix=/usr/local

RUN ldconfig
RUN cd /openmpi-3.1.3/examples && mpicc hello_c.c -o hello_c
```

Used OCI hooks

- SSH hook
- SLURM global sync hook

Running the container

To run an OpenMPI program using the SSH hook, we need to manually provide a list of hosts and explicitly launch mpirun only on one node of the allocation. We can do so with the following commands:

```
salloc -C gpu -N4 -t5
sarus ssh-keygen
srun hostname > $SCRATCH/hostfile
srun sarus run --ssh \
  --mount=src=$SCRATCH,dst=$SCRATCH,type=bind \
  ethscs/openmpi:3.1.3 \
  bash -c 'if [ $SLURM_PROCID -eq 0 ]; then mpirun --hostfile $SCRATCH/hostfile -
↳ npernode 1 /openmpi-3.1.3/examples/hello_c; else sleep infinity; fi'
```

Warning: Permanently added '[nid02182]:15263,[148.187.40.151]:15263' (RSA) to the list
↳ of known hosts.

Warning: Permanently added '[nid02180]:15263,[148.187.40.149]:15263' (RSA) to the list
↳ of known hosts.

Warning: Permanently added '[nid02181]:15263,[148.187.40.150]:15263' (RSA) to the list
↳ of known hosts.

Hello, world, I am 0 of 4, (Open MPI v3.1.3, package: Open MPI root@74cce493748b
↳ Distribution, ident: 3.1.3, repo rev: v3.1.3, Oct 29, 2018, 112)

Hello, world, I am 3 of 4, (Open MPI v3.1.3, package: Open MPI root@74cce493748b
↳ Distribution, ident: 3.1.3, repo rev: v3.1.3, Oct 29, 2018, 112)

Hello, world, I am 2 of 4, (Open MPI v3.1.3, package: Open MPI root@74cce493748b
↳ Distribution, ident: 3.1.3, repo rev: v3.1.3, Oct 29, 2018, 112)

Hello, world, I am 1 of 4, (Open MPI v3.1.3, package: Open MPI root@74cce493748b
↳ Distribution, ident: 3.1.3, repo rev: v3.1.3, Oct 29, 2018, 112)

1.7.6 GROMACS

GROMACS is a molecular dynamics package with an extensive array of modeling, simulation and analysis capabilities. While primarily developed for the simulation of biochemical molecules, its broad adoption includes reaserch fields such as non-biological chemistry, metadynamics and mesoscale physics. One of the key aspects characterizing GROMACS is the strong focus on high performance and resource efficiency, making use of state-of-the-art algorithms and optimized low-level programming techniques for CPUs and GPUs.

Test case

As test case, we select the 3M atom system from the [HECBioSim](#) benchmark suite for Molecular Dynamics:

```
A pair of hEGFR tetramers of 1IVO and 1NQL:
  * Total number of atoms = 2,997,924
  * Protein atoms = 86,996 Lipid atoms = 867,784 Water atoms = 2,041,230 Ions = 1,
  ↪ 914
```

The simulation is carried out using single precision, 1 MPI process per node and 12 OpenMP threads per MPI process. We measured runtimes for 4, 8, 16, 32 and 64 compute nodes. The input file to download for the test case is [3000k-atoms/benchmark.tpr](#).

Running the container

Assuming that the `benchmark.tpr` input data is present in a directory which Sarus is configured to automatically mount inside the container (here referred by the arbitrary variable `$INPUT`), we can run the container on 16 nodes as follows:

```
srun -C gpu -N16 srun sarus run --mpi \
  ethscs/gromacs:2018.3-cuda9.2-mpich3.1.4 \
  /usr/local/gromacs/bin/mdrun_mpi -s ${INPUT}/benchmark.tpr -ntomp 12
```

A typical output will look like:

```
:-) GROMACS - mdrun_mpi, 2018.3 (-:
...
Using 4 MPI processes
Using 12 OpenMP threads per MPI process

On host nid00001 1 GPU auto-selected for this run.
Mapping of GPU IDs to the 1 GPU task in the 1 rank on this node: PP:0
NOTE: DLB will not turn on during the first phase of PME tuning
starting mdrun 'Her1-Her1'
10000 steps,      20.0 ps.

              Core t (s)   Wall t (s)      (%)
Time:         20878.970    434.979    4800.0
              (ns/day)    (hour/ns)
Performance:      3.973      6.041

GROMACS reminds you: "Shake Yourself" (YES)
```

If the system administrator did not configure Sarus to mount the input data location during container setup, we can use the `--mount` option:

```
srun -C gpu -N16 sarus run --mpi \
  --mount=type=bind,src=<path-to-input-directory>,dst=/gromacs-data \
  ethscs/gromacs:2018.3-cuda9.2-mpich3.1.4 \
  /usr/local/gromacs/bin/mdrun_mpi -s /gromacs-data/benchmark.tpr -ntomp 12
```


Running the native application

CSCS provides and supports GROMACS on Piz Daint. This [documentation page](#) gives more details on how to run GROMACS as a native application. For this test case, the GROMACS/2018.3-CrayGNU-18.08-cuda-9.1 modulefile was loaded.

Container image and Dockerfile

The container image ethcscs/gromacs:2018.3-cuda9.2_mpi3.1.4 (based on cuda/9.2 and mpich/3.1) used for this test case can be pulled from CSCS [DockerHub](#) or be rebuilt with this Dockerfile:

```

1  #!/bin/sh
2  FROM ethcscs/mpich:ub1804_cuda92_mpi314
3
4  ## Uncomment the following lines if you want to build mpi yourself:
5  ## RUN apt-get update \
6  ## && apt-get install -y --no-install-recommends \
7  ##     wget \
8  ##     gfortran \
9  ##     zlib1g-dev \
10 ##     libopenblas-dev \
11 ## && rm -rf /var/lib/apt/lists/*
12 ##
13 ## # Install MPICH
14 ## RUN wget -q http://www.mpich.org/static/downloads/3.1.4/mpich-3.1.4.tar.gz \
15 ## && tar xf mpich-3.1.4.tar.gz \
16 ## && cd mpich-3.1.4 \
17 ## && ./configure --disable-fortran --enable-fast=all,03 --prefix=/usr \
18 ## && make -j$(nproc) \
19 ## && make install \
20 ## && ldconfig
21
22 # Install CMake (apt installs cmake/3.10.2, we want a more recent version)
23 RUN mkdir /usr/local/cmake \
24 && cd /usr/local/cmake \
25 && wget -q https://cmake.org/files/v3.12/cmake-3.12.4-Linux-x86_64.tar.gz \
26 && tar -xzf cmake-3.12.4-Linux-x86_64.tar.gz \
27 && mv cmake-3.12.4-Linux-x86_64 3.12.4 \
28 && rm cmake-3.12.4-Linux-x86_64.tar.gz \
29 && cd /
30
31 ENV PATH=/usr/local/cmake/3.12.4/bin/:${PATH}
32
33 # Install GROMACS (apt install gromacs/2018.1, we want a more recent version)
34 RUN wget -q http://ftp.gromacs.org/pub/gromacs/gromacs-2018.3.tar.gz \
35 && tar xf gromacs-2018.3.tar.gz \
36 && cd gromacs-2018.3 \
37 && mkdir build && cd build \
38 && cmake -DCMAKE_BUILD_TYPE=Release \
39         -DGMX_BUILD_OWN_FFTW=ON -DREGRESSIONTEST_DOWNLOAD=ON \
40         -DGMX_MPI=on -DGMX_GPU=on -DGMX_SIMD=AVX2_256 \
41         -DGMX_BUILD_MDRUN_ONLY=on \
42         .. \

```

(continues on next page)

(continued from previous page)

```

43 && make -j6 \
44 && make check \
45 && make install \
46 && cd ../../ \
47 && rm -fr gromacs-2018.3*

```

Required OCI hooks

- NVIDIA Container Runtime hook
- Native MPI hook (MPICH-based)

Results

We measure wall clock time (in seconds) and performance (in ns/day) as reported by the application logs. The speedup values are computed using the wall clock time averages for each data point, taking the native execution time at 4 nodes as baseline. The results of our experiments are illustrated in the following figure:

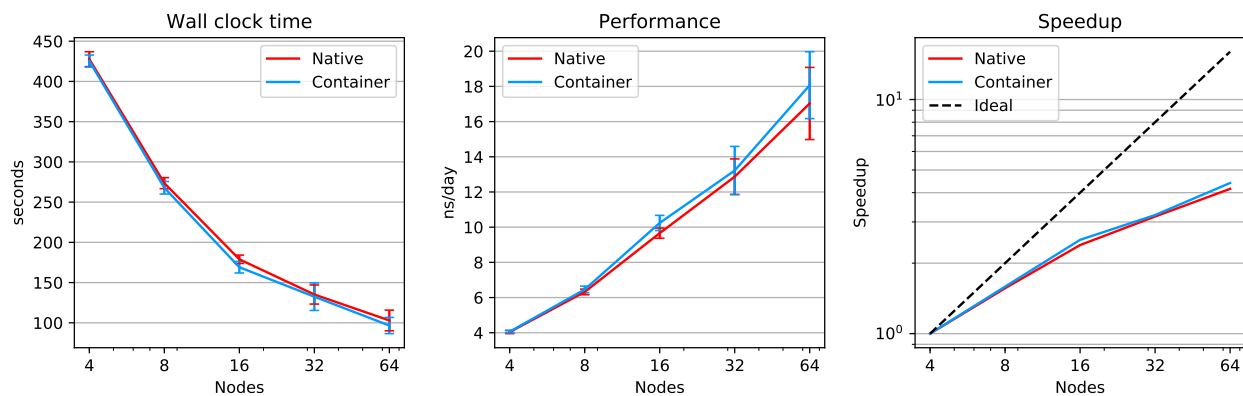


Fig. 7: Comparison of wall clock execution time, performance, and speedup between native and Sarus-deployed container versions of GROMACS on Piz Daint.

We observe the container application being up to 6% faster than the native implementation, with a small but consistent performance advantage and comparable standard deviations across the different node counts.

1.7.7 TensorFlow with Horovod

TensorFlow is a software framework providing an API to express numerical computations using data flow graphs. It also provides an implementation to run those computations on a broad array of platforms, from mobile devices to large systems with heterogeneous environments. While usable in a variety of scientific domains, it is mainly intended for the development of machine-learning (ML) models, with a focus on deep neural networks. The development of TensorFlow was started internally at Google Inc., and the software was released as open source in November 2015.

Horovod is a framework developed by Uber Technologies Inc. to perform distributed training of deep neural networks on top of another ML framework, like TensorFlow, Keras, or PyTorch. Notably, it allows to replace TensorFlow's own parameter server architecture for distributed training with communications based on an MPI model, leveraging ring-allreduce algorithms for improved usability and performance.

Horovod 0.15.1 with CUDA 9.x

As test case, we select the `tf_cnn_benchmark` scripts from the Tensorflow project for benchmarking convolutional neural networks. We use a `ResNet-50 model` with a batch size of 64 and the synthetic image data which the benchmark scripts are able to generate autonomously. We performed runs from a minimum of 2 nodes up to 128 nodes, increasing the node count in powers of two.

Running the container

Assuming that the tensorflow-benchmark code is present in a directory which Sarus is configured to automatically mount inside the container (here referred by the arbitrary variable `$INPUT`), we can run the container application as follows:

```

srun -C gpu -N4 -t5 sarus run --mpi \
    ethscs/horovod:0.15.1-tf1.7.0-cuda9.0-mpich3.1.4-no-nccl \
    python ${INPUT}/tensorflow-benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
    --model resnet50 --batch_size 64 --variable_update horovod

```

If the system administrator did not configure Sarus to mount the input data location during container setup, we can use the `--mount` option:

```

srun -C gpu -N4 -t5 sarus run --mpi \
    --mount=type=bind,src=<path-to-parent-directory>/tensorflow-benchmarks/scripts/,dst=/
    tf-scripts \
    ethscs/horovod:0.15.1-tf1.7.0-cuda9.0-mpich3.1.4-no-nccl \
    python /tf-scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
    --model resnet50 --batch_size 64 --variable_update horovod

```

Native application

For the native implementation, we use TensorFlow 1.7.0, built using Cray Python 3.6, the Python extensions provided by the Cray Programming Environment 18.08, CUDA 9.1 and cuDNN 7.1.4. These installations are performed by CSCS staff and are available on Piz Daint through environment modules. We complete the software stack by creating a virtual environment with Cray Python 3.6 and installing Horovod 0.15.1 through the `pip` utility. The virtual environment is automatically populated with system-installed packages from the loaded environment modules.

Container image and Dockerfile

We start from the reference Dockerfile provided by Horovod for version 0.15.1 and modify it to use Python 3.5, TensorFlow 1.7.0, CUDA 9.0, cuDNN 7.0.5. These specific versions of CUDA and cuDNN are required because they are the ones against which the version of TensorFlow available through `pip` has been built. We also replace OpenMPI with MPICH 3.1.4 and remove the installation of OpenSSH, as the containers will be able to communicate between them thanks to Slurm and the native MPI hook. Finally, we instruct Horovod not to use NVIDIA's NCCL library for any MPI operation, because NCCL is not available natively on Piz Daint. The final Dockerfile is the following:

```

1 FROM nvidia/cuda:9.0-devel-ubuntu16.04
2
3 # TensorFlow version is tightly coupled to CUDA and cuDNN so it should be selected
   ↪ carefully

```

(continues on next page)

(continued from previous page)

```

4  ENV HOROVOD_VERSION=0.15.1
5  ENV TENSORFLOW_VERSION=1.7.0
6  ENV PYTORCH_VERSION=0.4.1
7  ENV CUDNN_VERSION=7.0.5.15-1+cuda9.0
8
9  # NCCL_VERSION is set by NVIDIA parent image to "2.3.7"
10 ENV NCCL_VERSION=2.3.7-1+cuda9.0
11
12 # Python 2.7 or 3.5 is supported by Ubuntu Xenial out of the box
13 ARG python=3.5
14 ENV PYTHON_VERSION=${python}
15
16 RUN apt-get update && apt-get install -y --no-install-recommends \
17     build-essential \
18     cmake \
19     git \
20     curl \
21     vim \
22     wget \
23     ca-certificates \
24     libcudnn7=${CUDNN_VERSION} \
25     libnccl2=${NCCL_VERSION} \
26     libnccl-dev=${NCCL_VERSION} \
27     libjpeg-dev \
28     libpng-dev \
29     python${PYTHON_VERSION} \
30     python${PYTHON_VERSION}-dev
31
32 RUN ln -s /usr/bin/python${PYTHON_VERSION} /usr/bin/python
33
34 RUN curl -O https://bootstrap.pypa.io/get-pip.py && \
35     python get-pip.py && \
36     rm get-pip.py
37
38 # Install TensorFlow, Keras and PyTorch
39 RUN pip install tensorflow-gpu==${TENSORFLOW_VERSION} keras h5py torch==${PYTORCH_
    ↪VERSION} torchvision
40
41 # Install MPICH 3.1.4
42 RUN cd /tmp \
43     && wget -q http://www.mpich.org/static/downloads/3.1.4/mpich-3.1.4.tar.gz \
44     && tar xf mpich-3.1.4.tar.gz \
45     && cd mpich-3.1.4 \
46     && ./configure --disable-fortran --enable-fast=all,03 --prefix=/usr \
47     && make -j$(nproc) \
48     && make install \
49     && ldconfig \
50     && cd .. \
51     && rm -rf mpich-3.1.4 mpich-3.1.4.tar.gz \
52     && cd /
53
54 # Install Horovod, temporarily using CUDA stubs

```

(continues on next page)

(continued from previous page)

```

55 RUN ldconfig /usr/local/cuda-9.0/targets/x86_64-linux/lib/stubs && \
56     HOROVOD_WITH_TENSORFLOW=1 HOROVOD_WITH_PYTORCH=1 pip install --no-cache-dir horovod==
57     ↪ ${HOROVOD_VERSION} && \
58     ldconfig
59 # Set default NCCL parameters
60 RUN echo NCCL_DEBUG=INFO >> /etc/nccl.conf
61
62 # Download examples
63 RUN apt-get install -y --no-install-recommends subversion && \
64     svn checkout https://github.com/uber/horovod/trunk/examples && \
65     rm -rf /examples/.svn
66
67 WORKDIR "/examples"

```

Used OCI hooks

- NVIDIA Container Runtime hook
- Native MPI hook (MPICH-based)

Results

We measure the performance in images/sec as reported by the application logs and compute speedup values using the performance averages for each data point, taking the native performance at 2 nodes as baseline. The results are showcased in the following Figure:

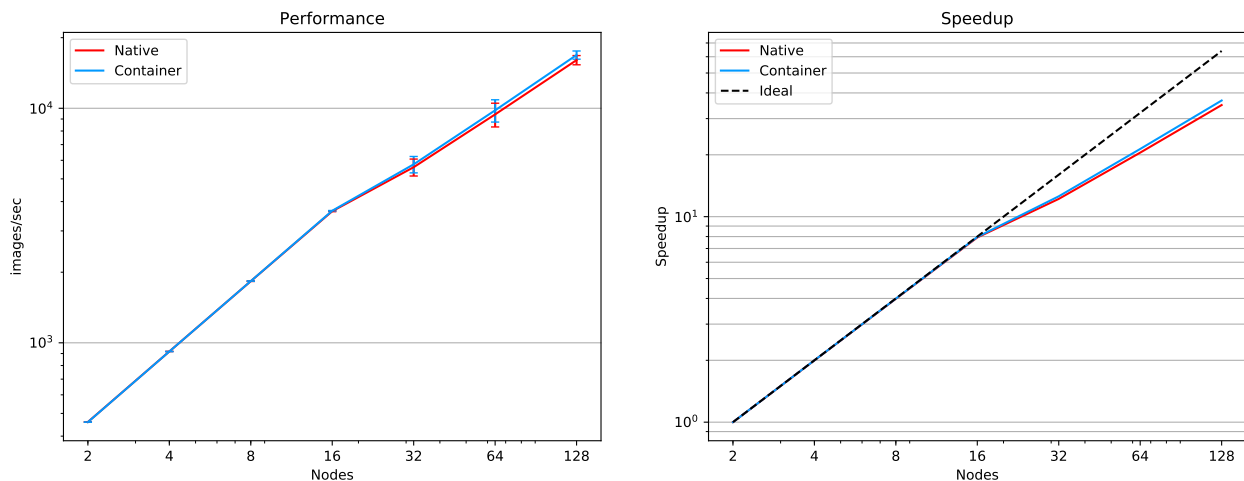


Fig. 8: Comparison of performance and speedup between native and Sarus-deployed container versions of TensorFlow with Horovod on Piz Daint.

We observe the container application closely matching the native installation when running on up to 16 nodes, with performance differences and normalized standard deviations less than 0.5%. From 32 nodes upwards, the container application shows a small performance advantage, up to 5% at 128 nodes, with both implementations maintaining close standard deviation values.

Horovod 0.16.x with CUDA 10.0

In this test case, we select again the `tf_cnn_benchmark` scripts from the Tensorflow project but now we test all four different models that the benchmark supports, namely the *alexnet*, *inception3*, *resnet50* and *vgg16*. The batch size is again 64 and for each of the models we use a node range of 1 to 12 nodes.

Running the container

If the tensorflow-benchmark code is present in a directory which Sarbus is configured to automatically mount inside the container (here referred by the arbitrary variable `$INPUT`), we can run the container application as follows:

```
srun -C gpu -N4 sarbus run --mpi \
    ethscs/horovod:0.16.1-tf1.13.1-cuda10.0-mpich3.1.4-nccl \
    python ${INPUT}/tensorflow-benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
    --model resnet50 --batch_size 64 --variable_update horovod
```

Alternatively, the `--mount` option can be used:

```
srun -C gpu -N4 -t5 sarbus run --mpi \
    --mount=type=bind,src=<path-to-parent-directory>/tensorflow-benchmarks/scripts/,dst=/
    tf-scripts \
    ethscs/horovod:0.16.1-tf1.13.1-cuda10.0-mpich3.1.4-nccl \
    python /tf-scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
    --model resnet50 --batch_size 64 --variable_update horovod
```

The above commands are using the `resnet50` model. Using the `--model` option it is possible to run the benchmarks with the other models as well.

Native application

For the native implementation, we use Horovod 0.16.0 with TensorFlow 1.12.0, built using Cray Python 3.6, the Python extensions provided by the Cray Programming Environment 19.03, CUDA 10.0, cuDNN 7.5.6 and NCCL 2.4.2. These installations are performed by CSCS staff and are available on Piz Daint through environment modules.

Container image and Dockerfile

We start from the reference Dockerfile provided by Horovod for version 0.16.1 and modify it to use Python 3.5, TensorFlow 1.13.1, CUDA 10.0, cuDNN 7.5.0 and NCCL 2.4.2. These specific versions of CUDA and cuDNN are required because they are the ones against which the version of TensorFlow available through pip has been built. We also replace OpenMPI with MPICH 3.1.4 and remove the installation of OpenSSH, as the containers will be able to communicate thanks to Slurm and the native MPI hook. Finally, we instruct Horovod to use NVIDIA's NCCL library for every MPI operation by adding the appropriate environment variables to the `/etc/nccl.conf` configuration file. The resulting Dockerfile is the following:

```
1 FROM nvidia/cuda:10.0-devel-ubuntu16.04
2
3 # Define the software versions
4 ENV HOROVOD_VERSION=0.16.1 \
5     TENSORFLOW_VERSION=1.13.1 \
6     CUDNN_VERSION=7.5.0.56-1+cuda10.0 \
```

(continues on next page)

(continued from previous page)

```

7     NCCL_VERSION=2.4.2-1+cuda10.0
8
9     # Python version
10    ARG python=3.5
11    ENV PYTHON_VERSION=${python}
12
13    # Install the necessary packages
14    RUN apt-get update && \
15        apt-get install -y --no-install-recommends \
16        cmake git curl vim wget ca-certificates \
17        libibverbs-dev \
18        libcudnn7=${CUDNN_VERSION} \
19        libnccl2=${NCCL_VERSION} \
20        libnccl-dev=${NCCL_VERSION} \
21        libjpeg-dev \
22        libpng-dev \
23        python${PYTHON_VERSION} python${PYTHON_VERSION}-dev
24
25    # Create symbolic link in order to make the installed python default
26    RUN ln -s /usr/bin/python${PYTHON_VERSION} /usr/bin/python
27
28    # Download pip bootstrap script and install pip
29    RUN curl -O https://bootstrap.pypa.io/get-pip.py && \
30        python get-pip.py && \
31        r    m get-pip.py
32
33    # Install Tensorflow, Keras and h5py
34    RUN pip install tensorflow-gpu==${TENSORFLOW_VERSION} keras h5py
35
36    # Install MPICH 3.1.4
37    RUN cd /tmp \
38        && wget -q http://www.mpich.org/static/downloads/3.1.4/mpich-3.1.4.tar.gz \
39        && tar xf mpich-3.1.4.tar.gz \
40        && cd mpich-3.1.4 \
41        && ./configure --disable-fortran --enable-fast=all,03 --prefix=/usr \
42        && make -j$(nproc) \
43        && make install \
44        && ldconfig \
45        && cd .. \
46        && rm -rf mpich-3.1.4 mpich-3.1.4.tar.gz \
47        && cd /
48
49    # Install Horovod
50    RUN ldconfig /usr/local/cuda-10.0/targets/x86_64-linux/lib/stubs && \
51        HOROVOD_GPU_ALLREDUCE=NCCL HOROVOD_WITH_TENSORFLOW=1 pip install --no-cache-dir \
52        ↪horovod==${HOROVOD_VERSION} && \
53        l    dconfig
54
55    # NCCL configuration
56    RUN echo NCCL_DEBUG=INFO >> /etc/nccl.conf && \
57        echo NCCL_IB_HCA=ipogif0 >> /etc/nccl.conf && \
58        echo NCCL_IB_CUDA_SUPPORT=1 >> /etc/nccl.conf

```

Used OCI hooks

- NVIDIA Container Runtime hook
- Native MPI hook (MPICH-based)

Results

We measure the performance in images/sec as reported by the application logs by taking the mean value based on 5 different runs for each model and node number. The results are showcased in the following Figure:

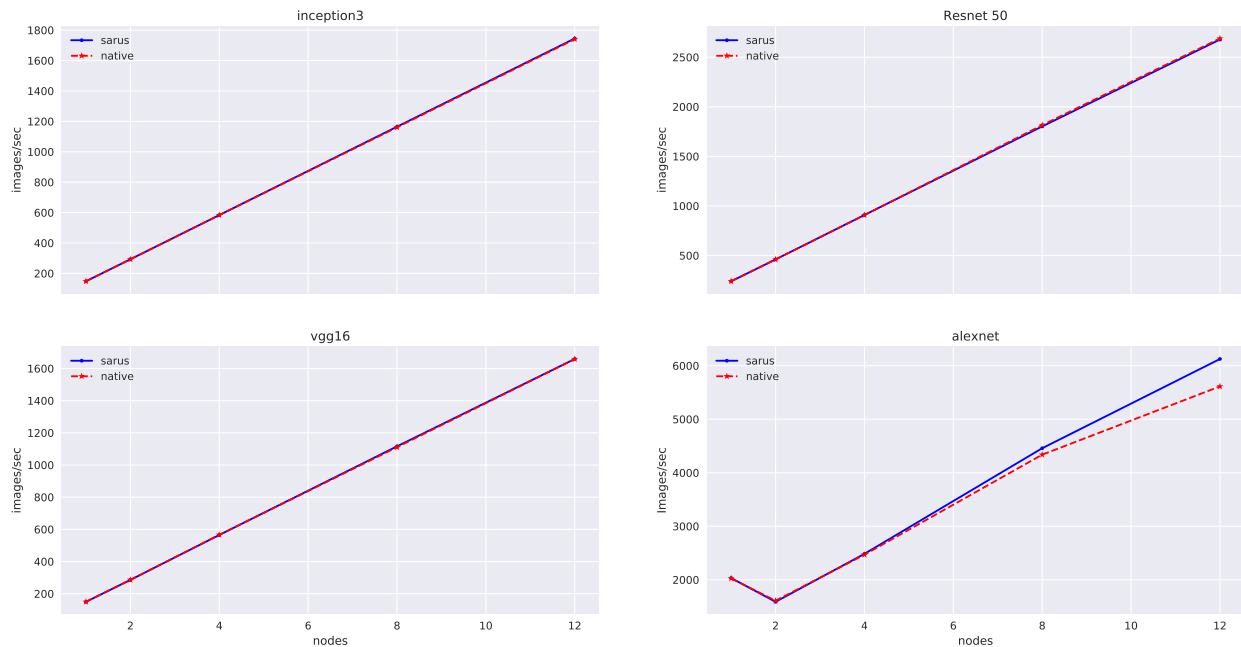


Fig. 9: Comparison of performance between native and Sarus-deployed container versions of TensorFlow with Horovod on Piz Daint.

We observe that performance of the container-based horovod-tensorflow is identical to the native one. A slight increased performance of the containerized solution is observed only for the alexnet model as the number of nodes increases.

1.7.8 PyFR

PyFR is a Python code for solving high-order computational fluid dynamics problems on unstructured grids. It leverages symbolic manipulation and runtime code generation in order to run different high-performance backends on a variety of hardware platforms. The characteristics of Flux Reconstruction make the method suitable for efficient execution on modern streaming architectures, and PyFR has been demonstrated to achieve good portability¹ and scalability on some of the world's most powerful HPC systems: most notably, a contribution based on PyFR was selected as one of the finalists for the 2016 ACM Gordon Bell Prize².

¹ F.D. Witherden, B.C. Vermeire, P.E. Vincent, Heterogeneous computing on mixed unstructured grids with PyFR, Computers & Fluids, Volume 120, 2015, Pages 173-186, ISSN 0045-7930, <https://doi.org/10.1016/j.compfluid.2015.07.016>

² P. Vincent, F. Witherden, B. Vermeire, J. S. Park and A. Iyer, "Towards Green Aviation with Python at Petascale", SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, 2016, pp. 1-11. <https://doi.org/10.1109/SC.2016.1>

Test case

As test case, we select the 2D Euler vortex example bundled with the PyFR source code. We run the example on 4 nodes with GPU acceleration enabled. We do not measure performance with this test and we do not make comparisons with a native implementation.

Running the container

We assume that a host scratchpad directory is configured to be mounted automatically inside the container by Sarus, and is accessible at the path defined by `$SCRATCH`. Documentation can be found on this dedicated page: [Mounting custom files and directories into the container](#). For instance: `--mount=type=bind,source=$SCRATCH,destination=$SCRATCH`

First step is to use an interactive container to prepare the simulation data:

```
# Launch interactive container
srun -C gpu -N1 -t10 --pty sarus run --tty \
    ethscs/pyfr:1.8.0-cuda9.2-ubuntu16.04 bash
```

then within the container, prepare the data:

```
1  #!/bin/bash
2
3  # From the container:
4  ## copy the example data to the scratch directory
5  mkdir $SCRATCH/pyfr/
6  cd PyFR-1.8.0/examples/
7  cp -r euler_vortex_2d/ $SCRATCH/pyfr/euler_vortex_2d
8
9  ## Convert mesh data to PyFR format
10 cd $SCRATCH/pyfr/euler_vortex_2d
11 pyfr import euler_vortex_2d.msh euler_vortex_2d.pyfrm
12
13 ## Partition the mesh and exit the container
14 pyfr partition 4 euler_vortex_2d.pyfrm .
```

and terminate the interactive session:

```
exit
```

Now that the data is ready, we can launch the multi-node simulation. Notice that we use the `--pty` option to `srun` in order to visualize and update correctly PyFR's progress bar (which we request with the `-p` option):

```
srun -C gpu -N4 -t1 --pty sarus run \
    --mpi \
    ethscs/pyfr:1.8.0-cuda9.2-ubuntu16.04 \
    pyfr run -b cuda -p \
    $SCRATCH/pyfr/euler_vortex_2d/euler_vortex_2d.pyfrm \
    $SCRATCH/pyfr/euler_vortex_2d/euler_vortex_2d.ini
```

A typical output will look like:

```
100.0% [======>] 100.00/100.00 daint: 00:00:29 rem: 00:00:00
```

Container image and Dockerfile

The container image *ethcscs/pyfr:1.8.0-cuda9.2-ubuntu16.04* (based on Nvidia cuda/9.2) used for this test case can be pulled from CSCS [DockerHub](#) or be rebuilt with this Dockerfile:

```

1 FROM nvidia/cuda:9.2-devel-ubuntu16.04
2
3 LABEL com.pyfr.version="1.8.0"
4 LABEL com.python.version="3.5"
5
6 # Install system dependencies
7 # Metis is a library for mesh partitioning:
8 # http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
9 RUN apt-get update && apt-get install -y \
10     unzip \
11     wget \
12     build-essential \
13     gfortran-5 \
14     strace \
15     realpath \
16     libopenblas-dev \
17     liblapack-dev \
18     python3-dev \
19     python3-setuptools \
20     python3-pip \
21     libhdf5-dev \
22     libmetis-dev \
23     --no-install-recommends && \
24     rm -rf /var/lib/apt/lists/*
25
26 # Install MPICH 3.1.4
27 RUN wget -q http://www.mpich.org/static/downloads/3.1.4/mpich-3.1.4.tar.gz && \
28     tar xvf mpich-3.1.4.tar.gz && \
29     cd mpich-3.1.4 && \
30     ./configure --disable-fortran --prefix=/usr && \
31     make -j$(nproc) && \
32     make install && \
33     cd .. && \
34     rm -rf mpich-3.1.4.tar.gz mpich-3.1.4 && \
35     ldconfig
36
37 # Create new user
38 RUN useradd docker
39 WORKDIR /home/docker
40
41 # Install Python dependencies
42 RUN pip3 install numpy>=1.8 \
43     pytools>=2016.2.1 \
44     mako>=1.0.0 \
45     appdirs>=1.4.0 \
46     mpi4py>=2.0 && \
47     pip3 install pycuda>=2015.1 \
48     h5py>=2.6.0 && \

```

(continues on next page)

(continued from previous page)

```

49  wget -q -O GiMMiK-2.1.tar.gz \
50      https://github.com/vincentlab/GiMMiK/archive/v2.1.tar.gz && \
51  tar -xvzf GiMMiK-2.1.tar.gz && \
52  cd GiMMiK-2.1 && \
53  python3 setup.py install && \
54  cd .. && \
55  rm -rf GiMMiK-2.1.tar.gz GiMMiK-2.1
56
57  # Set base directory for pyCUDA cache
58  ENV XDG_CACHE_HOME /tmp
59
60  # Install PyFR
61  RUN wget -q -O PyFR-1.8.0.zip http://www.pyfr.org/download/PyFR-1.8.0.zip && \
62      unzip -qq PyFR-1.8.0.zip && \
63      cd PyFR-1.8.0 && \
64      python3 setup.py install && \
65      cd .. && \
66      rm -rf PyFR-1.8.0.zip
67
68  CMD ["pyfr --help"]

```

Used OCI hooks

- NVIDIA Container Runtime hook
- Native MPI hook (MPICH-based)

References

1.7.9 Intel Cluster Edition 19.01

This section provides the instructions to build Intel Parallel Studio 2019 Update 1 based on the Centos 7 distribution: Intel Parallel Studio provides a complete environment to build an optimised application, since it features compilers, communication and mathematical libraries.

Please note that the Intel license prevents the redistribution of the software, therefore the license file needs to be available locally on the system where you are building the image: the license file is called *intel_license_file.lic* in the example Dockerfile provided below and it will be copied inside the container during the installation.

Furthermore, you need to create a local configuration file with the instructions required to proceed with the silent installation of Intel Parallel Studio: the name of the file is *intel19.01_silent.cfg* in the example below and it will provide the installation of the Intel C, C++ and Fortran Compiler together with the Intel MPI library and the Intel MKL.

Container image and Dockerfile

The starting point is the official image of CentOS 7: on top of that image, you need to install the required build tools and proceed with the silent installation of the Intel compiler.

```
FROM centos:7

SHELL ["/bin/bash", "--login", "-c"]

RUN yum install -y cpio

WORKDIR /usr/local/src

# install Intel Compiler and Intel MPI
COPY intel_license_file.lic /opt/intel/licenses/USE_SERVER.lic
ADD intel19.01_silent.cfg .
ADD parallel_studio_xe_2019_update1_cluster_edition_online.tgz .

RUN cd parallel_studio_xe_2019_update1_cluster_edition_online \
    && ./install.sh --ignore-cpu -s /usr/local/src/intel19.01_silent.cfg

RUN echo -e "source /opt/intel/bin/compilervars.sh intel64 \nsource /opt/intel/mkl/bin/\
↳mklvars.sh intel64 \nsource /opt/intel/impi/2019.1.144/intel64/bin/mpivars.sh release" \
↳>> /etc/profile.d/intel.sh \
    && echo -e "/opt/intel/lib/intel64 \n/opt/intel/mkl/lib/intel64 \n/opt/intel/impi/\
↳2019.1.144/intel64/lib \n/opt/intel/impi/2019.1.144/intel64/lib/release \n/opt/intel/\
↳impi/2019.1.144/intel64/libfabric/lib \n/opt/intel/impi/2019.1.144/intel64/libfabric/\
↳lib/prov" > /etc/ld.so.conf.d/intel.conf \
    && ldconfig
```

Please note that using the login shell will source the profile files and therefore update the paths to retrieve compiler executables and libraries: however it will have an effect only on the commands executed within the Dockerfile, serving as a reminder to use it when building applications with the Intel image.

The instructions above will copy the local license file *intel_license_file.lic* to */opt/intel/licenses/USE_SERVER.lic*, since the license file of the example will use a server to authenticate: please check that the address of the nameserver provided in *DOCKER_OPTS* will be able to resolve the name of the license server and that it is consistent with the nameservers listed in */etc/resolv.conf*.

The following silent configuration file provides a minimal list of the Intel COMPONENTS necessary for the installation: the advantage of the minimal list is the reduced amount of disk space required while building the image:

```
ACCEPT_EULA=accept
CONTINUE_WITH_OPTIONAL_ERROR=yes
PSET_INSTALL_DIR=/opt/intel
CONTINUE_WITH_INSTALLDIR_OVERWRITE=yes
COMPONENTS=;intel-clck__x86_64;intel-icc__x86_64;intel-ifort__x86_64;intel-mkl-core-c__
↳x86_64;intel-mkl-cluster-c__noarch;intel-mkl-gnu-c__x86_64;intel-mkl-core-f__x86_64;
↳intel-mkl-cluster-f__noarch;intel-mkl-gnu-f__x86_64;intel-mkl-f__x86_64;intel-imb__x86_
↳64;intel-mpi-sdk__x86_64
PSET_MODE=install
ACTIVATION_LICENSE_FILE=/opt/intel/licenses/USE_SERVER.lic
ACTIVATION_TYPE=license_server
AMPLIFIER_SAMPLING_DRIVER_INSTALL_TYPE=kit
AMPLIFIER_DRIVER_ACCESS_GROUP=vtune
```

(continues on next page)

(continued from previous page)

```

AMPLIFIER_DRIVER_PERMISSIONS=666
AMPLIFIER_LOAD_DRIVER=no
AMPLIFIER_C_COMPILER=none
AMPLIFIER_KERNEL_SRC_DIR=none
AMPLIFIER_MAKE_COMMAND=/usr/bin/make
AMPLIFIER_INSTALL_BOOT_SCRIPT=no
AMPLIFIER_DRIVER_PER_USER_MODE=no
SIGNING_ENABLED=yes
ARCH_SELECTED=INTEL64

```

The docker command line used to build the Intel image described in the Dockerfile above (save it as *intel19.01-cuda10.1.docker*) is the following: `` docker build --network=host -f intel19.01-cuda10.1.docker -t intel:19.01-cuda10.1 . `` The command line argument `--network=host` sets the networking mode for the *RUN* instructions during the build phase to match the host network configuration.

1.8 Glossary

This page contains the definitions of technical terms and concepts as they are used in the scope of Sarus and throughout its documentation.

container A Linux process isolated from the rest of the system through [Linux namespaces](#) and, potentially, other technologies like [cgroups](#), [SELinux](#) or [AppArmor](#).

Its filesystem and execution context primarily come from a [container image](#).

A container can be thought as the runtime instantiation of a container image.

container engine A software that interacts with users, imports [container images](#), and, from the point of view of the users, runs [containers](#).

When it comes to container execution, engines are generally tasked with preparing the mount point and metadata for the container, combining the contents of the image with additional instructions provided by user input or by system administrator settings.

Under the hood, many engines delegate actual container creation and running to a [container runtime](#).

Sarus is technically defined as a container engine.

container image A standalone, portable, self-sufficient unit of software which provides filesystem content and meta-data for the creation of a [container](#).

container runtime A program that interacts at low level with the kernel to create, start or modify containers.

The container runtime usually consumes the mount point and metadata provided by a [container engine](#). For testing purposes, a runtime can also be used directly, with individually crafted filesystem and metadata for a container.

Sarus uses [runc](#) as its default container runtime.

digest An immutable identifier which uniquely points to a specific version of an image within a [repository](#).

As defined by the [OCI Image Specification](#), digests take the form of a string using the `<algorithm>:<encoded>` pattern. The `algorithm` portion indicates the cryptographic algorithm used for the digest, while the `encoded` portion represents the result of the hash function.

In an [OCI image](#), every element is identified by a digest. The digest of the [image manifest](#) is considered as the digest of the image itself, and therefore also referred to as "image digest".

In the scope of Sarus, the "image digest" is the digest of the image **as stored in the *registry* where it was pulled from**. This is meant to help users relate local and remote images.

image manifest In a Docker or OCI image, the manifest is a JSON file listing the elements which compose the container image, that is the filesystem layers and image configuration file.

A manifest describes a single image for a specific architecture and operating system.

For the formal definition of the OCI Image Manifest, please refer to [this page](#).

image reference A string referencing a *container image* within a *registry* or *repository*. An image reference takes the following form:

```
[[server/]namespace/]image[:tag][@digest]
```

The **server** component represents the location of the registry to which the image belongs. This component is similar to a URL, but does not contain a protocol specifier; it can specify a port number, separated by a colon (for example `test.registry.io:5000`). When a **server** component is not specified in the command line, Sarus defaults to `docker.io` (Docker Hub).

The **namespace** component represents a namespace which may exist within a given registry and is used to subdivide or categorize images. Namespaces might be used by registries to represent organizations, user groups or individual users. Multiple nested namespaces can be specified by using the `/` character as separator. When a **namespace** component is not specified in the command line, Sarus defaults to `library`, which is the namespace for official images in Docker Hub.

The **image** component represents the name of the repository where the image is stored. By extension, this may also be referred to as the "image name".

The group `server/namespace/image` is also known as "full name" or "fully qualified image name". On the other hand, when the **server** or **namespace** components are not present, the group is known as "short name" or "unqualified image name".

The **tag** component represents a *tag*. When a **tag** component is not specified in the command line, Sarus defaults to `latest`.

The **digest** component represents a *digest*.

Formally, an image reference is defined by the following grammar:

```
reference           := name [ ":" tag ] [ "@" digest ]
name               := [domain '/' ] path-component [ '/' path-component ]*
domain             := domain-component [ '.' domain-component ]* [ ':' port-number ]
domain-component   := /([a-zA-Z0-9] | [a-zA-Z0-9] [a-zA-Z0-9-]* [a-zA-Z0-9-]
                    ↪ 9) ) /
port-number        := /[0-9]+ /
path-component     := alpha-numeric [separator alpha-numeric]*
alpha-numeric      := /[a-z0-9]+ /
separator          := /[_.] | __ | [-]* /

tag               := /[\\w][\\w.-]{0,127} /

digest            := digest-algorithm ":" digest-hex
digest-algorithm   := digest-algorithm-component [ digest-algorithm-
                    ↪ separator digest-algorithm-component ]*
digest-algorithm-separator := /[+.-_]/
digest-algorithm-component := /[A-Za-z][A-Za-z0-9]*/
digest-hex        := /[0-9a-fA-F]{32,} / ; At least 128 bit digest
                    ↪ value
```

(continues on next page)

(continued from previous page)

registry A hosted service (usually remote or cloud-based) which contains *repositories* of images. A registry responds to an API which enables clients to query and exchange data with the registry. The most popular examples of registry APIs are the [Docker Registry HTTP API V2](#) and the [OCI Distribution Spec API](#).

repository A collection of container images, usually representing a similar software stack or serving a similar purpose. Repositories can be hosted remotely in a *registry* or in a local filesystem.

Different images in a repository can be labeled with a *tag*.

tag A string used to label an *image* within a *repository*, in order to differentiate it from other images. Tags are mutable, and a tag can point to different images at different moments in time. In this regard, tags are conceptually similar to Git branches.

Many container tools (including Sarus) use the tag `latest` as default when a specific tag is not provided to an operation involving an image.

Tags are often used to identify specific software versions, releases or branches. Conventionally, the tag `latest` points to the latest software version in the repository, but it may also represent the latest *stable* software version.

1.8.1 Further reading

- [A Practical Introduction to Container Terminology](#) (Red Hat Developer portal)
- [Docker glossary](#)

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

BIBLIOGRAPHY

[unshare-manpage] <http://man7.org/linux/man-pages/man2/unshare.2.html>

[mount-namespace-manpage] http://man7.org/linux/man-pages/man7/mount_namespaces.7.html

[ShifterCUG2015] Jacobsen, D.M., Canon, R.S., “Contain This, Unleashing Docker for HPC”, Cray Users Group-Conference 2015 (CUG’15), <https://www.nersc.gov/assets/Uploads/cug2015udi.pdf>

INDEX

C

- container, [105](#)
- container engine, [105](#)
- container image, [105](#)
- container runtime, [105](#)

D

- digest, [105](#)

I

- image manifest, [106](#)
- image reference, [106](#)

R

- registry, [107](#)
- repository, [107](#)

T

- tag, [107](#)