Step-by-step guide to Sarus on Piz Daint

A note about SLURM commands

Piz Daint uses the SLURM Workload Manager to assign jobs to its compute nodes. In case you are not familiar with basic usage of SLURM, here we provide brief explanations to the commands used throughout this guide:

`salloc` is used to allocate resources for a job in real time. Typically this is used to allocate resources and spawn a shell. The shell is then used to execute `srun` commands to launch parallel tasks.

`srun` is used to submit a job for execution or initiate job steps in real time.

Both these commands support the following options:

`-C` indicates a list of constraints for the nodes where to make an allocation or run a job. In this document we will be using `-C gpu` to indicate we want to run on Piz Daint's GPU partition, with nodes featuring Intel Haswell CPUs and NVIDIA Pascal GPUs.

`-A` associates resources used by a job to a specific account. Required by CSCS policies for accounting purposes.

`--reservation` allocates resources on a specific reservation (please note that if you are taking part to a hands-on session with a dedicated reservation, such reservation will have a limited time duration).

`-n` indicates the total number of tasks to run

`-N` indicates the number of compute nodes to use

Preparing the Sarus environment

```
module load sarus
```

Basic usage

1. Pull a new image from Docker Hub

   You can pull images from Docker Hub into the HPC system with the `sarus pull` command. We strongly recommend to run `sarus pull` on the compute nodes through SLURM, so that Sarus can take advantage of their large RAM filesystem, which will greatly reduce the pull process time and will allow to pull larger images.

   EXAMPLE:

   ```
   $ salloc -N 1 -C gpu -A class02 --reservation=prace
   $ srun sarus pull debian

   # image            : index.docker.io/library/debian:latest
   ```

```
# cache directory  : "/scratch/snx3000/amadonna/.sarus/cache"
# temp directory   : "/tmp"
# images directory : "/scratch/snx3000/amadonna/.sarus/images"
> save image layers ...
> pulling        :
sha256:0bc3020d05f1e08b41f1c5d54650a157b1690cde7fedb1fafbc9cda70ee2ec5c
> completed      :
sha256:0bc3020d05f1e08b41f1c5d54650a157b1690cde7fedb1fafbc9cda70ee2ec5c
> expanding image layers ...
> extracting     :
"/scratch/snx3000/amadonna/.sarus/cache/sha256:0bc3020d05f1e08b41f1c5d54650a15
7b1690cde7fedb1fafbc9cda70ee2ec5c.tar"
> make squashfs image:
"/scratch/snx3000/amadonna/.sarus/images/index.docker.io/library/debian/latest
.squashfs"

$ exit
```

2. Query Sarus images

   You can list the Sarus images available to you on a system with the `sarus images` command. The images displayed here are located in an individual repository, and are not shared with other users.

   EXAMPLE:

```
$ sarus images

REPOSITORY       TAG          DIGEST         CREATED               SIZE
SERVER
debian           latest       be97a73019fb   2021-07-02T12:07:21   45.06MB
index.docker.io
```

3. Run a container with Sarus

   You can run containers using SLURM and the `sarus run` command, specifying the desired image as the first positional argument of the command. The arguments entered after the image's name will be interpreted as the command to be executed inside the container.

   You can check that you are actually running in a container by inspecting `/etc/os-release` on the host system.

   EXAMPLE:

```
$ salloc -N 1 -C gpu -A class02 --reservation=prace
$ srun sarus run debian cat /etc/os-release

PRETTY_NAME="Debian GNU/Linux 10 (buster)"
NAME="Debian GNU/Linux"
VERSION_ID="10"
VERSION="10 (buster)"
VERSION_CODENAME=buster
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

```
$ srun cat /etc/os-release

NAME="SLES"
VERSION="15-SP1"
VERSION_ID="15.1"
PRETTY_NAME="SUSE Linux Enterprise Server 15 SP1"
ID="sles"
ID_LIKE="suse"
ANSI_COLOR="0;32"
CPE_NAME="cpe:/o:suse:sles:15:sp1"

$ exit
```

4. Run a container with an interactive shell

   As with Docker, you can access Sarus containers through an interactive shell. The `-t/--tty` command line option to `sarus run` and the `--pty` flag to `srun` have to be used in order to properly setup the connection to the terminal. In this example we use the official Docker image for Python 3.9 to also showcase the capability of writing files inside containers.

   EXAMPLE:

```
$ salloc -N 1 -C gpu -A class02 --reservation=prace
$ srun sarus pull python:3.9

[ sarus pull output ]

$ srun --pty sarus run -t python:3.9 bash
$ cat /etc/os-release

PRETTY_NAME="Debian GNU/Linux 10 (buster)"
NAME="Debian GNU/Linux"
VERSION_ID="10"
VERSION="10 (buster)"
VERSION_CODENAME=buster
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"

$ python --version
Python 3.9.6

$ echo 'print("Hello world!\n", 3+2+4)' > hello.py
$ ls hello.py

hello.py

$ python hello.py

Hello world!
 9

$ exit  #from the container
$ exit
```

5. Remove Sarus images

   To remove an image from Sarus's local repository, use the `sarus rmi` command. This is useful if the image repository is subject to a storage quota, so you can make room for new images by removing old ones.

   EXAMPLE:

```
$ sarus images
REPOSITORY       TAG          DIGEST         CREATED               SIZE
SERVER
debian           latest       e29dbf6781ec   2021-07-02T12:07:21   40.38MB
index.docker.io
python           3.9          2979bca36f5c   2021-07-02T12:11:15   293.38MB
index.docker.io

$ sarus rmi python:3.9
removed image index.docker.io/library/python:3.9

$ sarus images
REPOSITORY       TAG          DIGEST         CREATED               SIZE
SERVER
debian           latest       e29dbf6781ec   2021-07-02T12:07:21   40.38MB
index.docker.io
```

Running MPI containers

6. Run with the MPI from the image

   Containers with MPI can run unmodified, using the MPI implementation provided in the image, as long as they are compatible with the Process Management Interface (PMI) used by the host MPI launcher to communicate with the rank processes.

   In this example, we use the OSU Micro-benchmarks point-to-point latency test in a container image featuring MPICH 3.1.4, which uses the PMI2 interface. Thus, we use the `--mpi=pmi2` option to `srun` to tell the Slurm workload manager that we specifically want PMI2 to be used.

   EXAMPLE:

```
$ salloc -C gpu -A class02 -N 2 --reservation=prace
$ srun -N1 sarus pull ethcscs/osu-mb:5.7-mpich3.1.4-cuda11.3.0-centos8
$ srun --mpi=pmi2 sarus run ethcscs/osu-mb:5.7-mpich3.1.4-cuda11.3.0-centos8
./osu_latency

# OSU MPI Latency Test v5.7
# Size          Latency (us)
0                       6.91
1                       6.67
2                       6.90
4                       6.74
8                       6.79
```

```
16                      6.78
32                      6.75
64                      6.79
128                     6.79
256                     6.92
512                     7.06
1024                    9.55
2048                   10.42
4096                   11.16
8192                   11.82
16384                  13.28
32768                  16.73
65536                  28.36
131072                 54.84
262144                 82.01
524288                135.85
1048576               242.49
2097152               461.53
4194304               891.67

$exit
```

> **Note:** The working directory set in the image is understood by Sarus, allowing us to use a relative path as the container argument.

> **Note:** Images featuring OpenMPI can work as well using this syntax, at the condition that OpenMPI was built with Slurm and PMI2 support.

7. Run with native MPI

   Since MPICH 3.1.4 is ABI compatible with the Cray MPI found natively on Piz Daint, we can use the `--mpi` option of `sarus run` to replace the image's original MPI libraries with their native counterparts. By having the native MPI implementation inside the container, applications can leverage the full performance of the Cray Aries high-speed interconnect.

   Notice that this example does not use `srun --mpi=pmi2` anymore: since the Cray MPI stack is injected in the image, it can communicate naturally with the Cray PMI technology used by default on Piz Daint.

   EXAMPLE:

```
$ salloc -C gpu -A class02 -N 2 --reservation=prace
$ srun sarus run --mpi ethcscs/osu-mb:5.7-mpich3.1.4-cuda11.3.0-centos8
./osu_latency

# OSU MPI Latency Test v5.7
# Size          Latency (us)
0                       1.16
1                       1.13
2                       1.07
4                       1.07
8                       1.08
16                      1.09
```

```
32                       1.10
64                       1.10
128                      1.10
256                      1.13
512                      1.14
1024                     1.36
2048                     1.64
4096                     2.19
8192                     4.11
16384                    4.97
32768                    6.67
65536                    9.99
131072                  16.62
262144                  29.95
524288                  56.57
1048576                109.38
2097152                217.84
4194304                432.61

$exit
```

Compare the latency performance with the results of the previous point.

8. Run with NVIDIA GPUDirect RDMA

Having the native MPI libraries inside the container doesn't unlock just additional performance, but also advanced features which may not be available int the original image MPI.

For example, on Piz Daint we can leverage direct MPI communications between GPU devices through the NVIDIA GPUDirect RDMA technology. To do so with our latency test, we need to set the `MPICH_RDMA_ENABLED_CUDA=1` environment variable and prepend the OSU test with the `get_local_rank` utility (the latter is a requirement of the OSU benchmarks and is not related to Sarus nor Piz Daint).

EXAMPLE:

```
$ salloc -C gpu -A class02 -N 2 --reservation=prace
$ MPICH_RDMA_ENABLED_CUDA=1 srun sarus run --mpi ethcscs/osu-mb:5.7-
mpich3.1.4-cuda11.3.0-centos8 usr/local/libexec/osu-micro-
benchmarks/get_local_rank ./osu_latency -d cuda D D

# OSU MPI-CUDA Latency Test v5.7
# Send Buffer on DEVICE (D) and Receive Buffer on DEVICE (D)
# Size          Latency (us)
0                        1.10
1                        5.47
2                        5.46
4                        5.45
8                        5.50
16                       5.53
32                       5.55
64                       5.55
128                      5.55
256                      5.57
512                      5.59
1024                     5.64
2048                     5.75
```

```
4096                    6.31
8192                    7.37
16384                   9.53
32768                  13.75
65536                  20.77
131072                 49.03
262144                 82.76
524288                150.01
1048576               211.38
2097152               337.67
4194304               589.62

$exit
```

> **Note:** For some applications to load GPUDirect correctly, the following environment variable should be defined: `LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libcuda.so`.

9. **Bonus:** Run a collective benchmark

   Collective benchmarks can run with more than 2 MPI ranks. As an example, the Allreduce MPI routine combines values from all processes and distributes the result back to all processes. Allreduce has gained importance in recent years due to its usage in implementing distributed Deep Learning algorithms. Run the `osu_allreduce` benchmark program with the different MPI setups shown in the previous points (or even with different node counts), and observe how the results change:

```
$ salloc -C gpu -A class02 -N 4 --reservation=prace

# Container MPI
$ srun --mpi=pmi2 sarus run ethcscs/osu-mb:5.7-mpich3.1.4-cuda11.3.0-centos8
../collective/osu_allreduce

# Native MPI
$ srun sarus run --mpi ethcscs/osu-mb:5.7-mpich3.1.4-cuda11.3.0-centos8
../collective/osu_allreduce

# Native MPI + GPUDirect RDMA
$ MPICH_RDMA_ENABLED_CUDA=1 srun sarus run --mpi ethcscs/osu-mb:5.7-
mpich3.1.4-cuda11.3.0-centos8 /usr/local/libexec/osu-micro-
benchmarks/get_local_rank ../collective/osu_allreduce -d cuda

$exit
```

Running GPU containers

10. Detect GPUs available in the container

    Enabling native GPU support in Sarus on Piz Daint does not require any direct user action, besides running a job on the GPU partition.

    To list the GPU devices available in the container, you can run the `nvidia-smi` utility.

    EXAMPLE:

```
$ salloc -N 1 -C gpu -A class02 --reservation=prace
$ srun sarus run debian nvidia-smi

Thu Dec 20 17:08:26 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.44                 Driver Version: 396.44                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name       Persistence-M| Bus-Id       Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf Pwr:Usage/Cap|         Memory-Usage| GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  On   | 00000000:02:00.0 Off|                    0 |
| N/A   28C    P0    30W / 250W|      0MiB / 16280MiB|      0%   E. Process |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+

$ exit
```

11. Run a GPU application in the container

    With GPUs available, CUDA applications in containers work right out of the box. For example, you can print details about GPU devices using the `deviceQuery` sample provided with the CUDA Toolkit SDK. We have already built an image with compiled CUDA samples, and you can retrieve it from Docker Hub using the identifier `ethcscs/cudasamples:10.0`.

    EXAMPLE:

```
$ salloc -N 1 -C gpu -A class02 --reservation=prace
$ srun sarus pull ethcscs/cudasamples:10.0

[ sarus pull output ]

$ srun sarus run ethcscs/cudasamples:10.0
/usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery

/usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery Starting...
```

```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla P100-PCIE-16GB"
  CUDA Driver Version / Runtime Version          11.2 / 10.0
  CUDA Capability Major/Minor version number:    6.0
  Total amount of global memory:                 16281 MBytes (17071734784
bytes)
  (56) Multiprocessors, ( 64) CUDA Cores/MP:     3584 CUDA Cores
  GPU Max Clock rate:                            1329 MHz (1.33 GHz)
  Memory Clock rate:                             715 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072,
65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048
layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 2 / 0
  Compute Mode:
     < Exclusive Process (many threads in one process is able to use
::cudaSetDevice() with this device) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.2, CUDA Runtime
Version = 10.0, NumDevs = 1
Result = PASS

$ exit
```

You can also run the `nbody` sample which is provided with the CUDA Toolkit SDK.

EXAMPLE:

```
$ salloc -N 1 -C gpu -A class02 --reservation=prace
$ srun sarus run ethcscs/cudasamples:10.0
/usr/local/cuda/samples/5_Simulations/nbody/nbody -benchmark -fp64 -
```

```
numbodies=200000

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)
        -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: 3881.523 ms
= 103.250 billion interactions per second
= 3097.508 double-precision GFLOP/s at 30 flops per interaction

$ exit
```

To see the effect of GPU acceleration, try to run the sample benchmark on the CPU using the `-cpu` option. We advise to greatly reduce the number of bodies specified with the `-numbodies` option to avoid waiting too long.

Real world applications: MPI + GPU + Data I/O

GROMACS is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles. It is primarily designed for biochemical molecules like proteins, lipids and nucleic acids but many groups are also using it for research on non-biological systems, e.g. polymers. GROMACS is heavily optimized and supports both parallel execution with MPI and CUDA GPU acceleration.

As test case, we will use the GROMACS Test Case B from PRACE's Unified European Applications Benchmark Suite. The data models a cellulose and lignocellulosic biomass in an aqueous solution, using reaction-field electrostatics

12. Prepare the test case

```
$ salloc -N 1 -C gpu -A class02 --reservation=prace
$ cd $SCRATCH
$ srun sarus pull ethcscs/gromacs:2020.4-cuda11.2.0-mpich3.1.4-centos8
$ wget https://repository.prace-
ri.eu/ueabs/GROMACS/1.2/GROMACS_TestCaseB.tar.gz
$ tar xf GROMACS_TestCaseB.tar.gz
$ exit
```

13. Run with bind mounts for data I/O

    As is often the case with real-world applications, we need to provide input data and collect output data produced by the software. In our case, this means that we need to exchange data between a host filesystem (where input data is located and where we want to save output data) and a container (where the application software is run). To do so, we use the `--mount` and `--workdir` options of Sarus.

    The `--mount` option will map a path from the host filesystem to another path into the container. It is the primary way to enable data I/O between host and containers. The host path to mount into the container is indicated with the `source` flag, while the path inside the container is indicated with the `destination` flag (if the path does not exist, it will be created). Currently the option only supports bind mounts (`type=bind`). In this example, we will mount the user's personal directory of Piz Daint's `/scratch` filesystem (defined in the `$SCRATCH` environment variable) to the same path in the container. This is consistent with CSCS policies which recommend the scratch filesystem for data actively used/generated in compute jobs.

    The `--workdir` option sets the initial working directory for the container. In this example we will use it as a convenience feature: by default, GROMACS writes output files in its current working directory. By moving the workdir to our location (`--workdir=$PWD`), we can simplify the command line and skip telling GROMACS the explicit paths for each output file.

    EXAMPLE:

```
$ salloc -N 4 -C gpu -A class02 --reservation=prace
$ cd $SCRATCH
$ srun sarus run --mpi --mount=type=bind,source=$SCRATCH,destination=$SCRATCH
--workdir=$PWD ethcscs/gromacs:2020.4-cuda11.2.0-mpich3.1.4-centos8 gmx_mpi
mdrun -s $PWD/lignocellulose-rf.tpr -g gromacs.log -ntomp 1 -nsteps 100

[ GROMACS output ]

# Output files have been generated in the current directory after container
```

```
execution
$ ls -l

total 605928
-rw-r--r-- 1 amadonna csstaff 228836081 May 31 16:38 confout.gro
-rw-r--r-- 1 amadonna csstaff      1928 May 31 16:38 ener.edr
-rw-r--r-- 1 amadonna csstaff     24705 May 31 16:38 gromacs.log
-rw-r--r-- 1 amadonna csstaff 111654018 Oct 25  2016 GROMACS_TestCaseB.tar.gz
-rw-r----- 1 amadonna csstaff 100244496 Aug  2  2013 lignocellulose-rf.BGQ.tpr
-rw-r--r-- 1 amadonna csstaff 100021568 Oct 13  2016 lignocellulose-rf.tpr
-rw-r--r-- 1 amadonna csstaff  79596872 May 31 16:38 state.cpt

$ exit
```

14. Run with multiple MPI ranks per node

    GROMACS is sometimes run with multiple MPI ranks per node in order to increase its efficiency. To do so on Piz Daint, we need to set the `CRAY_CUDA_MPS=1` environment variable, which will allow GPUs to be used by more than one process, and set the `--ntasks-per-node` option to `srun`. In this example we run 12 MPI ranks per node, one for each CPU core available on Piz Daint's compute nodes.

    EXAMPLE:

```
$ salloc -N 4 -C gpu -A class02 --reservation=prace
$ cd $SCRATCH
$ CRAY_CUDA_MPS=1 srun -N4 --ntasks-per-node=12 sarus run --mpi --
mount=type=bind,src=$SCRATCH,dst=$SCRATCH --workdir=$PWD
ethcscs/gromacs:2020.4-cuda11.2.0-mpich3.1.4-centos8 gmx_mpi mdrun -s
$PWD/lignocellulose-rf.tpr -g gromacs.log -ntomp 1 -nsteps 1000

$ exit
```

Bonus application: Horovod

Horovod is a distributed deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet. Notably, it allows to perform communication between training processes using the MPI model, instead of alternative solutions like TensorFlow parameter servers.

In this example we run Horovod's own synthetic benchmark script for convolutional neural networks; at the moment of writing, the script uses by default ResNet 50 as network model and a batch size of 32.

EXAMPLE:

```
$ salloc -N 4 -C gpu -A class02 --reservation=prace
$ srun -N 1 sarus pull ethcscs/horovod:0.22.0-tf2.5.0-cuda11.2-mpich3.1.4-
ubuntu18.04
$ srun sarus run --mpi ethcscs/horovod:0.22.0-tf2.5.0-cuda11.2-mpich3.1.4-
ubuntu18.04 python tensorflow2/tensorflow2_synthetic_benchmark.py --fp16-allreduce

# Optional comparison with native software
$ module load daint-gpu
$ module load Horovod/0.21.0-CrayGNU-20.11-tf-2.4.0
$ wget
```

```
https://raw.githubusercontent.com/horovod/horovod/master/examples/tensorflow2/tens
orflow2_synthetic_benchmark.py
$ MPICH_RDMA_ENABLED_CUDA=1 srun python ./tensorflow2_synthetic_benchmark.py --
fp16-allreduce

$ exit
```

Installing Sarus

You can quickly install Sarus through a standalone archive by following the steps below:

1. Ensure the `mksquashfs` program is available:

```
$ sudo apt-get update
$ sudo apt-get install squashfs-tools
$ which mksquashfs
```

2. Download the latest standalone Sarus archive from the official GitHub Releases:

```
$ wget https://github.com/eth-cscs/sarus/releases/download/1.3.2/sarus-
Release.tar.gz
```

3. Extract Sarus in the installation directory:

```
$ sudo mkdir /opt/sarus
$ sudo tar -C /opt/sarus/ -xf sarus-Release.tar.gz
```

4. Run the configuration script to finalize the installation of Sarus:

```
$ cd /opt/sarus/1.3.2-Release
$ 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.

Note that the configuration script will create a minimal working configuration. For enabling additional features, please refer to the full configuration reference.

As explained by the output of the script, you'll need to persistently add the Sarus binary to your `PATH`. This is usually taken care of through an environment module, if the Environment Modules package is available on the system. A basic alternative is adding a line like `export PATH=/opt/sarus/default/bin:${PATH}` to your `.bashrc` file.

> **Note:** The Sarus binary from the standalone archive looks for SSL certificates into the `/etc/ssl` directory. Depending on the Linux distribution, some certificates may be located in different directories. A possible solution to expose the certificates to Sarus is a symlink. For example, on CentOS 7 and Fedora 31:
>
> ```
> sudo ln -s /etc/pki/ca-trust/extracted/pem/tls-ca-bundle.pem
> /etc/ssl/cert.pem
> ```

5. Ensure required kernel modules are loaded:

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

Additional information to understand and maintain Sarus installations are available in the Post-installation page of the official documentation.