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 contraints 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 hybrid partition, with nodes featuring Intel Haswell CPUs and NVIDIA Pascal GPUs.

`--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

Installing Sarus

You can quickly install Sarus through a standalone archive 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/1.3.2/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/1.3.2-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.

Also 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
> ```

4. 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.

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 --reservation=<reservation name>
$ srun sarus pull debian

srun: job 760756 queued and waiting for resources
srun: job 760756 has been allocated resources
# image     : index.docker.io/library/debian/latest
# cacheDir  : /scratch/snx3000/<user>/.sarus/cache
# tmpDir    : /dev/shm
# imageDir  : /scratch/snx3000/<user>/.sarus/images
> save image layers ...
> pulling        :
sha256:cc1a78bfd46becbfc3abb8a74d9a70a0e0dc7a5809bbd12e814f9382db003707
> completed      :
sha256:cc1a78bfd46becbfc3abb8a74d9a70a0e0dc7a5809bbd12e814f9382db003707
> expand image layers ...
> extracting     :
/scratch/snx3000/<user>/.sarus/cache/sha256:cc1a78bfd46becbfc3abb8a74d9a70a0e0
dc7a5809bbd12e814f9382db003707.tar
> make squashfs ...
> create metadata ...
# created:
/scratch/snx3000/<user>/.sarus/images/index.docker.io/library/debian/latest.sq
uashfs
# created:
/scratch/snx3000/<user>/.sarus/images/index.docker.io/library/debian/latest.me
ta

$ 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
library/debian   latest       e29dbf6781ec   2018-12-19T18:00:55   40.38MB
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 --reservation=<reservation name>
$ srun sarus run debian cat /etc/os-release

PRETTY_NAME="Debian GNU/Linux 9 (stretch)"
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/"

$ srun cat /etc/os-release

NAME="SLES"
VERSION="12-SP2"
VERSION_ID="12.2"
PRETTY_NAME="SUSE Linux Enterprise Server 12 SP2"
ID="sles"
ANSI_COLOR="0;32"
CPE_NAME="cpe:/o:suse:sles:12:sp2"

$ 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` has to be used in order to properly setup the connection to the terminal. In this example we use the official Docker image for Python 3.7 to also showcase the capability of writing files inside containers.

EXAMPLE:

```
$ salloc -N 1 -C gpu --reservation=<reservation name>
$ srun sarus pull python:3.7

[ sarus pull output ]

$ srun --pty sarus run --tty python:3.7 bash
$ cat /etc/os-release

PRETTY_NAME="Debian GNU/Linux 9 (stretch)"
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/"

$ python --version
```

```
Python 3.7.1

$ 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 :program: `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
library/alpine   latest       f33084b4b603   2018-12-20T14:54:53   2.06MB
index.docker.io
library/debian   latest       e29dbf6781ec   2018-12-19T18:00:55   40.38MB
index.docker.io

$ sarus rmi python:3.7
removed index.docker.io/library/alpine/latest

$ sarus images
REPOSITORY       TAG          DIGEST         CREATED               SIZE
SERVER
library/debian   latest       e29dbf6781ec   2018-12-19T18:00:55   40.38MB
index.docker.io
```

Running MPI containers

6. Run with the container (image?) MPI

```
$ salloc -C gpu -N 2 --reservation=<reservation name>
$ 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



$exit
```

> **Note:** Working directory 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

```
$ salloc -C gpu -N 2 --reservation=<reservation name>
$ srun --mpi=pmi2 sarus run ethcscs/osu-mb:5.7-mpich3.1.4-cuda11.3.0-centos8
./osu_latency

$exit
```

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

8. Run with NVIDIA GPUDirect RDMA

```
$ salloc -C gpu -N 2 --reservation=<reservation name>
$ 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

$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 -N 4 --reservation=<reservation name>

# 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 hybrid CPU/GPU partition.

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

```
$ salloc -N 1 -C gpu --reservation=<reservation name>
$ 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:9.2`.

    EXAMPLE:

```
$ salloc -N 1 -C gpu --reservation=<reservation name>
$ srun sarus pull ethcscs/cudasamples:9.2
```

```
[ sarus pull output ]

$ srun sarus run ethcscs/cudasamples:9.2
/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          9.2 / 9.2
  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 = 9.2, CUDA Runtime
Version = 9.2, 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 --reservation=<reservation name>
$ srun sarus run ethcscs/cudasamples:9.2
/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: 3925.279 ms
= 102.099 billion interactions per second
= 3062.980 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

... Intro to application and test case ...

12. Prepare the test case

```
$ salloc -N 1 -C gpu --reservation=<reservation name>
$ srun sarus pull quay.io/madeeks/gromacs:2020.4-cuda11.2.0-mpich3.1.4-
ubuntu20.04
$ 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

    Explain `--mount` and `--workdir` options

```
$ salloc -N 4 -C gpu --reservation=<reservation name>
$ srun sarus run --mpi --mount=type=bind,src=$SCRATCH,dst=$SCRATCH --
workdir=$PWD quay.io/madeeks/gromacs:2020.4-cuda11.2.0-mpich3.1.4-ubuntu20.04
gmx_mpi mdrun -s $PWD/lignocellulose-rf.tpr -g gromacs.log -ntomp 1 -nsteps
100

# 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

```
$ salloc -N 4 -C gpu --reservation=<reservation name>
$ CRAY_CUDA_MPS=1 srun --ntasks-per-node=12 sarus run --mpi --
mount=type=bind,src=$SCRATCH,dst=$SCRATCH --workdir=$PWD
quay.io/madeeks/gromacs:2020.4-cuda11.2.0-mpich3.1.4-ubuntu20.04 gmx_mpi mdrun
-s $PWD/lignocellulose-rf.tpr -g gromacs.log -ntomp 1 -nsteps 1000

$ exit
```

Bonus application: Horovod

```
$ salloc -N 4 -C gpu --reservation=<reservation name>
$ srun -n1 sarus pull quay.io/madeeks/horovod:0.22.0-tf2.5.0-cuda11.2-mpich3.1.4-
ubuntu18.04
$ srun sarus run --mpi quay.io/madeeks/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
```