Access

There is only one login node and it can be reached with SSH (Secure Shell) protocol using its hostname:

> login.dgx.cineca.it

The DGX login node is a virtual machine with 2 cpus and a x86_64 architecture, and without GPUs. It is only meant for accessing the system, transfer data, and submitting jobs to the DGX nodes.

Accounting

For accounting information please consult our dedicated section.
The account_no (or project) is important for batch executions. You need to indicate an account_no to be accounted for in the scheduler, using the flag `-A`.

```bash
#SBATCH -A <account_no>
```

With the "saldo -b" command you can list all the account_no associated with your username.

```bash
> saldo -b  (reports projects defined on DGX )
```

Please note that the accounting is in terms of consumed core hours, but it strongly depends also on the requested memory and number of GPUs, please refer to the dedicated section.

## Disks and Filesystems

The storage organization conforms to the CINECA infrastructure (see Section "Data storage and Filesystems") as for the user's areas. In addition to the home directory ($HOME), for each user is defined a scratch area `$CINECA_SCRATCH`, a disk for storing run time data and files. On this DGX cluster there is no `$WORK` directory.

<table>
<thead>
<tr>
<th></th>
<th>Total dimension (TB)</th>
<th>Quota (GB)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$HOME</td>
<td>5</td>
<td>50</td>
<td>permanent/backup, user specific, local</td>
</tr>
<tr>
<td><code>$CINECA_SCRATCH</code></td>
<td>90</td>
<td>no quota</td>
<td>permanent, user specific, local</td>
</tr>
</tbody>
</table>

It is also available a temporary local storage on compute nodes generated when the job starts and accessible via environment variable `$TMPDIR`.

```
TMPDIR=/raid/scratch_local/slurm_job.$SLURM_JOB_ID
```

which can be used exclusively by the job’s owner. During your jobs, you can access the area with the (local) variable `$TMPDIR`. In your sbatch script, we suggest to move the input data of your simulations to the `$TMPDIR` before the beginning of your run and also write on `$TMPDIR` your results. The use of this area improves the I/O operations speed of your code by approximately 30%. However, the directory is removed at the end of the job, hence always remember to save the data stored in such area to a permanent directory in your sbatch script at the end of the run.

On DGX compute nodes the `$TMPDIR` local area has ~14 TB of available space. This quota is shared among all users with active jobs on the same compute nodes. For more details please see the dedicated section of [UG2.5: Data storage and FileSystems](https://www.cineca.it/en/ug2.5-data-storage-and-filesystems).

The `$DRES` filesystem is not mounted on the compute nodes and, at the moment, neither on the login node.

Since all the filesystems are based on GlusterFS the usual unix command "quota" is not working. Differently for other CINECA clusters cindata command is not yet available.

For information about data transfer from other computers please follow the instructions and caveats on the dedicated section [Data storage](https://www.cineca.it/en/data-storage) or the document [Data Management](https://www.cineca.it/en/data-management).

## Modules environment

The DGX module environment is based on lmod and it is minimal, only few modules being available on the cluster:

- HPC SDK: nvidia compilers and libraries
- OpenMPI: OpenMPI 4.0.3 (on compute nodes only)
- AI datasets: common AI datasets, such as ImageNet, available locally on each compute node.
- Singularity: container platform, needed to pull, and execute containers on DGX compute node.

In order to see all the installed module you can type "module av" or "module spider".

We have installed few modules because we want to encourage users to run their simulations using containers technology (via singularity). In most cases you don't need to build your own container, but you can simply pull (download) it from Nvidia, Docker or Singularity repositories.

## AI datasets modules

The AI datasets are organized in three categories: audio, images, videos. The datasets are stored on a shared directory AND on the NVMe memories to provide a significantly faster access inside jobs. Two sets of variables are hence defined by loading these modules, pointing to the location which is also accessible on the login node, and to the location available ONLY on the compute nodes (which we encourage to use inside jobs). For instance, for the imagenet ILSVRC2012 dataset module defines the following variables:

- `$DATA_ILSVRC2012_TRAIN` : training dataset on the compute nodes only
- `$DATA_ILSVRC2012_TRAIN_LOGIN` : training dataset on login and compute nodes
- `$DATA_ILSVRC2012_VAL` : validation dataset on the compute nodes only
- `$DATA_ILSVRC2012_VAL_LOGIN` : validation dataset on login and compute nodes
With the command "module help <module_name>" you will find the list of variables defined for each data module.

**Production environment**

Since DGX is a general purpose system and it is used by several users at the same time, long production jobs must be submitted using a queuing system (scheduler). This guarantees that the access to the resources is as fair as possible. On DGX the available scheduler is SLURM.

DGX cluster is based on a policy of node sharing among different jobs, and users can at most request one full node for their jobs. This means that, at a given time, one physical node can be allocated to multiple jobs of different users. Nevertheless, exclusivity at the level of the single core and GPU is guaranteed by low-level mechanisms.

Roughly speaking, there are two different modes to use an HPC system: Interactive and Batch. For a general discussion see the section Production Environment.

**Interactive**

A serial program can be executed in the standard UNIX way:

```bash
> ./program
```

This is allowed only for very short runs on the login node, since the **interactive environment has a 10 minutes cpu-time limit.** Please do not execute parallel applications on the login node, and keep in mind that it is a virtual machine with 2 cpus and no GPUs.

A serial (or parallel) program, also using GPUs and needing more than 10 minutes can be executed interactively within an "Interactive" SLURM batch job.

A request for resources allocation on the compute nodes is delivered to SLURM with salloc/srun commands, the request is queued and scheduled as any other batch job but, when granted, the standard input, output, and error streams of the interactive job connected to the terminal session from which the request was launched.

For example, to start an interactive session on one node and get the full node in exclusive way (including the eight GPUs) for one hour, launch the command:

```bash
> srun -N1 --exclusive --gres=gpu:8 -A <account_name> -p <partition_name> --time=01:00:00 --pty bash
```

Due to the slurm node configuration (defined with Sockets=2 CoresPerSocket=64 ThreadsPerCore=2 Procs=128 CPUs=256 Gres=gpu:8), the cores are always assigned with both threads (virtual cpus) so to provide a total number of virtual cpus equal to the number of requested tasks. Hence, if you request two tasks per node, two HTs of one physical core will be assigned to the job. If you want two physical cores you also have to specify that each task requests two (virtual) cpus (--cpus-per-tasks=2).

```bash
> srun -N1 --ntasks-per-node=2 --cpus-per-task=2 --gres=gpu:2 -A <account_name> -p <partition_name> --time=01:00:00 --pty bash
```

SLURM automatically exports the environment variables you defined in the source shell on the login node. If you need a specific environment for your job (i.e. specific library paths or options) you can "prepare" it on the login node before submitting your job.

A more specific description of the options used by salloc/srun to allocate resources in the "Batch" section, because they are the same of the sbatch command described there.

**Batch**

As usual on HPC systems, the production runs are executed in batch mode. This means that the user writes a list of commands into a file (for example script.x) and then submits it to a scheduler (SLURM for DGX) that will search for the required resources in the system. As soon as the resources are available script.x is executed.

This is an example of script file:

```bash
#!/bin/bash
#SBATCH --A <account_name>
#SBATCH --p dgx_usr_prod
#SBATCH --time 00:10:00  # format: HH:MM:SS
#SBATCH --N 1            # 1 node
#SBATCH --ntasks-per-node=8 # 8 tasks
#SBATCH --gres=gpu:1     # 1 gpus per node out of 8
#SBATCH --mem=7100       # memory per node out of 980000 MB
#SBATCH --job-name=my_batch_job
#SBATCH --mail-type=ALL
#SBATCH --mail-user=<user_email>

<your commands>
```
As stated in the previous section, by requesting --ntasks-per-node=8 your job will be assigned 8 logical cpus (hence, the first 4 cpus with their 2 HTs). You can write your script file (for example script.x) using any editor, then you submit it using the command:

```bash
> sbatch script.x
```

The script file must contain both directives to SLURM and commands to be executed, as better described in the section Batch Scheduler SLURM.

Using SLURM directives you indicate the account_number (-A: which project pays for this work), where to run the job (-p: partition), the optional requested QOS, what is the maximum duration of the run (-time: time limit). On DGX there is only one partition (dgx_usr_prod), and it is defined as the default one; hence, you can omit the #SBATCH -p directive and your job will automatically be submitted to the default partition. Moreover you indicate the resources needed, in terms of cores, GPUs and memory.

One of the commands will be probably the launch of a singularity container application (see Section How to use Singularity on DGX).

### Summary

In the following table, you can find all the main features and limits imposed on the SLURM partitions and QOS.

<table>
<thead>
<tr>
<th>SLURM partition</th>
<th>QOS</th>
<th>#cores/GPUs per job</th>
<th>max walltime</th>
<th>max running jobs per user/ max n. of cores/GPUs/nodes per user</th>
<th>Priority</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>dgx_usr_prod</td>
<td>dgx_qos_sprod</td>
<td>max = 32 cores (64 cpus) / 2 GPUs max mem=245000MB</td>
<td>12 h</td>
<td>1 job per user 32 cores (64 cpus) / 2 GPUs</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>normal (noQOS)</td>
<td>max = 128 cores (256 cpus) / 8 GPUs max mem=980000MB</td>
<td>4 h</td>
<td>1 job per user 8 GPUs / 1 node per user</td>
<td>40</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### How to use Singularity on DGX

On each node of DGX cluster singularity is installed in the default path. You don't need to load singularity module in order to use it, but we have created it to provide some examples to users via the command "module help singularity". If you need more info you can type `singularity --help` or visit Singularity documentation web site.

#### Pull a container

To pull a container from a repository you need to execute the command `singularity pull <container location>`. For example if you want to download pytorch container from docker repository you can use this command:

```bash
singularity pull docker://nvcr.io/nvidia/pytorch:20.12-py3
```

While pulling a container form a repository you may encounter some issues:

1. **no space left on device.** This error happens often because singularity uses /tmp directory to store temporary files. Since /tmp size is quite small, when it is full you obtain this error and you cannot perform container pull. We suggest to set the variable SINGULARITY_TMPDIR in order to set a different temporary directory for Singularity, e.g.

   ```bash
   $ export SINGULARITY_TMPDIR=$CINECA_SCRATCH/tmp
   ```

2. **CPU time limit exceeded.** This error happens when the container you want to pull is quite large and the last stage (creating SIF image) needs to much CPU time. On the login node each process has 10 minutes of CPU time limit. In order to avoid this issue we suggest to request an interactive job to pull it, or pull it on your laptop and copy it on DGX cluster.

#### Run a container

Once you have pulled a container, running it is quite easy. Here an interactive example about pytorch container:

```bash
fcola000@dgxslurm:~$ git clone https://github.com/pytorch/examples.git
fcola000@dgxslurm:~$ srun -p dgx_usr_prod -N1 -n32 --gres=gpu:2 -t 01:00:00 --pty /bin/bash
fcola000@dgx01:~$ cd examples/mnist
fcola000@dgx01:~examples/mnist$ singularity exec --nv $HOME/pytorch_20.12-py3 python main.py
```
You can launch the same computation using a batch script:

<table>
<thead>
<tr>
<th>Run a container in batch mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcola000@dgxslurm:~$ cat submit.sh</td>
</tr>
<tr>
<td>#!/bin/bash</td>
</tr>
<tr>
<td>#SBATCH -N1</td>
</tr>
<tr>
<td>#SBATCH -n 32</td>
</tr>
<tr>
<td>#SBATCH -p dgx_usr_prod</td>
</tr>
<tr>
<td>#SBATCH -t 01:00:00</td>
</tr>
<tr>
<td>#SBATCH --gres=gpu:2</td>
</tr>
<tr>
<td>#SBATCH -o job_%J.out</td>
</tr>
<tr>
<td>#SBATCH -e job_%J.err</td>
</tr>
<tr>
<td>git clone <a href="https://github.com/pytorch/examples.git">https://github.com/pytorch/examples.git</a></td>
</tr>
<tr>
<td>cd $HOME/examples/mnist</td>
</tr>
<tr>
<td>singularity exec --nv $HOME/pytorch_20.12-py3.sif python main.py</td>
</tr>
<tr>
<td>fcola000@dgxslurm:~$ sbatch submit.sh</td>
</tr>
</tbody>
</table>

Here you can find more examples about pytorch.

**Build your own container**

Please refer to the relative section of this page in our User Guide.