UG3.2.1: LEONARDO Booster UserGuide

Production environment

Since LEONARDO is a general purpose system and is used by several users at the same time, long production jobs must be submitted using a queuing system (scheduler). The scheduler guarantees that the access to the resources is as fair as possible. The production environment on LEONARDO Booster partition is based on the SLURM scheduler.

LEONARDO is based on a policy of node sharing among different jobs, i.e. a job can ask for resources and these can also be a part of a node, for example few cores and 1 GPU. 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 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:

```
$ ./program
```

This is allowed only for very short runs, since the interactive environment set on the login nodes has a **10 minutes time limit**: for longer runs please use the "batch" mode.

Please do not execute parallel applications on the login nodes!

Batch

As usual on HPC systems, the large 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 LEONARDO) that will search for the required resources in the system. As soon as the resources are available script.x is executed and the results and sent back to the user.

This is an example of script file:

```
#!/bin/bash
#SBATCH -A <account_name>
#SBATCH -p boost_usr_prod
#SBATCH --time 00:10:00     # format: HH:MM:SS
#SBATCH -N 1                # 1 node
#SBATCH --ntasks-per-node=4 # 4 tasks out of 32
#SBATCH --gres=gpu:4        # 4 gpus per node out of 4
#SBATCH --mem=123000          # memory per node out of 494000MB (481GB)
#SBATCH --job-name=my_batch_job

srun ./myexecutable
```

- Please refer to the general online guide to SLURM and on task/thread bindings, and please pay attention to the setting of the SRUN_CPUS_PER_TASK for hybrid applications dispatched with "srun".

You can write your script file (for example script.x) using any editor, then you submit it using the command:

```
$ 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_name (-A: which project pays for this work), where to run the job (-p: partition), what is the maximum duration of the run (--time: time limit). Moreover you indicate the resources needed, in terms of cores, GPUs and memory.

Please note that the recommended way to launch parallel MPI applications in slurm jobs is with srun. By using srun instead of mpirun you will get full support for process tracking, accounting, task affinity, suspend/resume and other features.

Please note: the "mail" directive #SBATCH --mail-user is not effective yet.

SLURM partitions

A list of partitions defined on the cluster, with access rights and resources definition, can be displayed with the command sinfo:

$ sinfo -o "%10D %20F %P"

The command returns a more readable output which shows, for each partition, the total number of nodes and the number of nodes by state in the format "Allocated/Idle/Other/Total".

In the following table you can find the main features and limits imposed on the partitions of LEONARDO Booster.

SLURM partition	**Job QOS**	**# cores/# GPU per job**	**max walltime**	**max running jobs per user/ max n. of nodes/cores/GPUs per user**	**priority**	**notes**
lrd_all_serial (default)	*normal*	max = 4 physical cores (8 logical cpus) max mem = 30800 MB	04:00:00	1 node / 4 cores / 30800 MB	40	No GPUs Hyperthreading x2
boost_usr_prod	*normal*	max = 64 nodes	24:00:00		40	
	boost_qos_dbg	max = 2 nodes	00:30:00	2 nodes / 64 cores / 8 GPUs	80	
	boost_qos_bprod	min = 65 nodes max =256 nodes	24:00:00	256 nodes	60	runs on 1536 nodes min is 65 FULL nodes
	boost_qos_lprod	max = 3 nodes	4-00:00:00	3 nodes /12 GPUs	40	

- For EUROFusion users and their dedicated queues please refer to the dedicated document.

Programming environment

LEONARDO Booster compute nodes host four A100 GPUs per node (CUDA compute capability 8.0). The most recent versions of NVIDIA CUDA toolkit and of the NVIDIA nvhpc compilers (ex PGI, supporting CUDA Fortran) are available in the module environment.

Compilers

You can check the complete list of available compilers on LEONARDO with the command

```
$ modmap -c compilers
```

The available CUDA-aware compilers are:

- GNU Compilers Collection (GCC)
- NVIDIA nvhpc (ex PGI)
- CUDA

Intel compilers are also available, but do not support CUDA, thus they are described in the page dedicated to LEONARDO Data Centric partition.

GNU Compiler Collection (GCC)

The GNU compilers are always available. GCC version 8.5.0 is available without the need to load any module. In the module environment you can find more recent version though.

The name of the GNU compilers are:

- **g77**: Fortran77 compiler
- **gfortran**: Fortran95 compiler
- **gcc**: C compiler
- **g++**: C++ compiler

The documentation can be obtained with the "man" command after loading the GNU module:

```
$ man gfortan
$ man gcc
```

NVIDIA nvhpc (ex PORTLAND PGI + NVIDIA CUDA)

As of August 5, 2020, the "PGI Compilers and Tools" technology is a part of the NVIDIA HPC SDK product, available as a free download from NVIDIA.

Invocations	**Usage**
nvc	Compile C source files (C11 compiler. It supports GPU programming with OpenACC, and supports multicore CPU programming with OpenACC and OpenMP)
nvc++	Compile C++ source files (C++17 compiler. It supports GPU programming with C++17 parallel algorithms (pSTL) and OpenACC, and supports multicore CPU programming with OpenACC and OpenMP)

nvfortran	Compile FORTRAN source files (supports ISO Fortran 2003 and many features of ISO Fortran 2008. It supports GPU programming with CUDA Fortran and OpenACC, and supports multicore CPU programming with OpenACC and OpenMP)
nvcc	CUDA C and CUDA C++ compiler driver for NVIDIA GPUs

For legacy reasons, the NVIDIA nvhpc suite also offers the PGI C, C++, and Fortran compilers with their original names, as follows.

Invocations	Usage
pgcc	Compile C source files.
pgc++	Compile C++ source files.
pgf77	Compile FORTRAN 77 source files
pgf90	Compile FORTRAN 90 source files
pgf95	Compile FORTRAN 95 source files

To enable CUDA C++ or CUDA Fortran, and link with the CUDA runtime libraries, use the **-cuda** option (-Mcuda is deprecated). Use the **-gpu** option to tailor the compilation of target accelerator regions.

The OpenACC parallelization is enabled by the **-acc** flag. GPU targeting and code generation can be controlled by adding the **-gpu** flag to the compiler command line.

The OpenMP parallelization is enabled by the **-mp** compiler option. The GPU offload via OpenMP is enabled by the **-mp=gpu** option.

CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by **NVIDIA** for general computing on **graphical processing units (GPUs).** With CUDA, developers are able to **dramatically speed up** computing applications by harnessing the **power of GPUs.**

In GPU-accelerated applications, the **sequential part** of the workload runs on the **CPU** – which is optimized for single-threaded performance – while the **compute intensive portion** of the application runs on thousands of **GPU cores** in parallel. When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords. We refer to the NVIDIA CUDA Parallel Computing Platform documentation.

CUDA compilers are available inside the nvhpc module, as well as in a stand-alone module.

Debugger and Profilers

If at runtime your code dies, then there is a problem. In order to solve it, you can decide to analyze the core file (core not available with PGI compilers) or to run your code using the debugger.

Compiler flags

Whatever your decision, in any case, you need to **enable compiler runtime checks**, by putting specific flags during the compilation phase. In the following we describe those flags for the different Fortran compilers: if you are using the C or C++ compiler, please check before because the flags may differ.

The following flags are generally available for all compilers and are mandatory for an easier debugging session:

```
-O0     Lower level of optimization
-g      Produce debugging information
```

Other flags are compiler specific and are described in the following.

PORTLAND Group (PGI) Compilers

The following flags are useful (in addition to "-O0 -g") for debugging your code:

```
-C                    Add array bounds checking
-Ktrap=ovf,divz,inv   Controls the behaviour of the processor when exceptions occur:
                      FP overflow, divide by zero, invalid operands
```

GNU Fortran compilers

The following flags are useful (in addition to "-O0 -g")for debugging your code:

```
-Wall              Enables warnings pertaining to usage that should be avoided
-fbounds-check     Checks for array subscripts.
```

Debuggers available

GNU: gdb (serial debugger)

GDB is the GNU Project debugger and allows you to see what is going on 'inside' your program while it executes -- or what the program was doing at the moment it crashed.

VALGRIND

Valgrind is a framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. The Valgrind distribution currently includes six production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache profiler, and a heap profiler.

Valgrind is Open Source / Free Software, and is freely available under the GNU General Public License, version 2.

Profilers

In software engineering, **profiling** is the investigation of a program's behavior using information gathered as the program executes. The usual purpose of this analysis is to determine which sections of a program to optimize - to increase its overall speed, decrease its memory requirement or sometimes both.

A (code) *profiler* is a performance analysis tool that, most commonly, measures only the frequency and duration of function calls, but there are other specific types of profilers (e.g. memory profilers) in addition to more comprehensive profilers, capable of gathering extensive performance data.

gprof

The GNU profiler **gprof** is a useful tool for measuring the performance of a program. It records the number of calls to each function and the amount of time spent there, on a per-function basis. Functions which consume a large fraction of the run-time can be identified easily from the output of gprof. Efforts to speed up a program should concentrate first on those functions which dominate the total run-time.

gprof uses data collected by the -pg compiler flag to construct a text display of the functions within your application (call tree and CPU time spent in every subroutine). It also provides quick access to the profiled data, which let you identify the functions that are the most CPU-intensive. The text display also lets you manipulate the display in order to focus on the application's critical areas.

Usage:

```
$ gfortran -pg -O3 -o myexec myprog.f90
$ ./myexec
$ ls -ltr
   .......
   -rw-r--r-- 1 aer0 cineca-staff    506 Apr  6 15:33 gmon.out
$ gprof myexec gmon.out
```

It is also possible to profile at code line-level (see "man gprof" for other options). In this case, you must use also the “-g” flag at compilation time:

```
$ gfortran -pg -g -O3 -o myexec myprog.f90
$ ./myexec
$ ls -ltr
   .......
   -rw-r--r-- 1 aer0 cineca-staff    506 Apr  6 15:33 gmon.out
$ gprof -annotated-source myexec gmon.out
```

It is possible to profile MPI programs. In this case, the environment variable GMON_OUT_PREFIX must be defined in order to allow to each task to write a different statistical file. Setting

```
$ export GMON_OUT_PREFIX=<name>
```

once the run is finished each task will create a file with its process ID (PID) extension

```
<name>.$PID
```

If the environmental variable is not set every task will write the same gmon.out file.

Nvidia Nsight System (GPU profiler)

Nvidia Nsight System is a system-wide performance analysis tool designed to visualize an application’s algorithms, help you identify the largest opportunities to optimize, and tune to scale efficiently across any quantity or size of CPUs and GPUs; from large server to our smallest SoC.
You can find general info on how to use it in the dedicated Nvidia User Guide pages.

Our suggestion is to run the CLI inside your job script in order to generate the qdrep files. Then you can download the qdrep files on your local PC and visualize them with the Nsight System GUI available on your workstation.

The profiler is available under the module nvhpc.

Standard usage of an MPI job running on GPU is

```
$ mpirun <options> nsys profile -o ${PWD}/output_%q{OMPI_COMM_WORLD_RANK} -f true --stats=true --cuda-memory-
usage=true <your_code> <input> <output>
```

On the single node you can also run the profiler as "nsys profile mpirun", but keep in mind that with this syntax nsys will put everything in a single report.

Unfortunately nsys usually generates several files in /tmp dir of the compute node even if a TMPDIR environment variable is set. These files may be big causing the **filling of the /tmp folder** and, as a consequence, the **crash of the compute** node and the **failure of the job**.
In order to avoid such a problem we strongly suggest to include in your sbatch script the following lines around your mpirun call as a **workaround**:

```
$ rm -rf /tmp/nvidia
$ ln -s $TMPDIR /tmp/nvidia
$ mpirun ... nsys profile ...
$ rm -rf /tmp/nvidia
```

This will place the temporary outputs of the nsys code in your TMPDIR folder that by default is /dev/shm/slurm_job.$SLURM_JOB_ID where you have about 250 GB of free space.
This workaround may cause conflicts between multiple jobs running this profiler on a compute node at the same time, so we **strongly suggest** also to request the compute node **exclusively**:

```
#SBATCH --exclusive
```

MPI environment

OpenMPI is the most common MPI implementation. It is installed inside the GNU environment, and it is configured to support CUDA. Here you can find some useful details on how to use OpenMPI on LEONARDO Booster partition.

The MPI implementation of Intel, i.e. Intel-OneAPI-MPI, even if available, doesn't support CUDA, thus you can find details on the Data Centric partition section.

Compiling

OpenMPI

To install MPI applications using OpenMPI, you have to load openMPI module (use "modmap -m openmpi" command to see the available OpenMPI versions) and select the MPI compiler wrapper for Fortran, C or C++ codes.

The openmpi module provides the following wrappers:

Compiler	Wrapper	Usage
g++	mpic++ mpiCC mpicxx	Compile C++ source files with GNU
gcc	mpicc	Compile C source files with GNU
gfortran	mpif77 mpif90 mpifort	Compile FORTRAN source files with GNU

e.g. Compiling C code:

```
$ module load openmpi/<version>
$ mpicc -o myexec  myprog.c (uses the gcc compiler)
```

You can add all options available for the backend compiler (you can show it by "-show" flag, e.g. "mpicc -show"). In order to list them type the "man" command:

```
$ man mpicc
```

Running

To run MPI applications there are two ways:

- using mpirun launcher
- using srun launcher

mpirun launcher

To use mpirun launcher on LEONARDO Booster partition, the **openmpi** module needs to be loaded:

```
$ module load openmpi/<VERSION>
```

After loading the module, MPI applications can be directly launched as

```
$ mpirun ./mpi_exec
```

or via salloc

```
$ salloc -N 2 (allocate a job of 2 nodes)
$ mpirun ./mpi_exec
```

or via sbatch

```
$ sbatch -N 2 my_batch_script.sh (allocate a job of 2 nodes)
$ cat my_batch_script.sh
#!/bin/sh
mpirun ./mpi_exec
```

srun launcher

MPI applications can also be launched directly with the SLURM launcher srun

```
$ srun -N 2  ./mpi_exec
```

or via salloc

$ salloc -N 2 (allocate a job of 2 nodes)
$ srun ./mpi_exec

or via sbatch

```
$ sbatch -N 2 my_batch_script.sh (allocate a job of 2 nodes)
$ vi my_batch_script.sh
#!/bin/sh
srun -N 2 ./mpi_exec
```

Scientific libraries

Libraries listed in this section are GPU-accelerated and support CUDA (see LEONARDO Data Centric section for not CUDA-aware libraries).

The **nvidia** math libraries are available by loading "nvhpc" module (use "modmap -m nvhpc" command to see the available versions of nvhpc).

Non-nvidia math libraries installed with cuda support are available by loading the corresponding module, e.g "module load magma/<vers>". Notice that when you load the module of any of these libraries, the CUDA module is not automatically loaded.

Linear Algebra

- BLAS: nvidia cublas, magma
- LAPACK: nvidia cusolver, magma
- SCALAPACK: slate
- EIGENVALUE SOLVERS: nvidia cusolver, magma (single-node), slate, elpa and slepC (multi-node)
- SPARCE MATRICES : nvidia cuSPARSE, PetSc (multi-node), SuperLU-dist (multi-node)
- Hypre (multi-node)

Fast Fourier Transform

- nvidia cuFFT/cuFFTW (single-node)

GPU and intra/inter connection environment

It will be described soon.