UG3.2.2: LEONARDO DCGP 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 Data Centric General Purpose (DCGP) 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. 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 dcgp_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 112
#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 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 Data Centric.

SLURM partition	Job QOS	# cores/ # GPU per job	max walltime	max n. of nodes/cores/mem per user max n. of nodes per account	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
dcgp_usr_prod	*normal*	max = 16 nodes	24:00:00	512 nodes per account	40	
	dcgp_qos_dbg	max = 2 nodes	00:30:00	2 nodes / 224 cores per user 512 nodes per account	80	
	dcgp_qos_bprod	min = 17 nodes max =128 nodes	24:00:00	128 nodes per user 512 nodes per account	60	runs on 1536 nodes min is 17 FULL nodes
	dcgp_qos_lprod	max = 3 nodes	4-00:00:00	3 nodes / 336 cores per user 512 nodes per account	40	

Note: a maximum of 512 nodes per account is also imposed on the dcgp_usr_prod partition, meaning that, for each account, all the jobs associated with it cannot run on more than 512 nodes at the same time (if you submit a job that imply to exceed this limitation, it will stay pending until a

Programming environment

LEONARDO Data Centric compute nodes are not provided with GPUs, thus applications running on GPUs can be used only on the Booster partition. The programming environment include a list of compilers and of debugger and profiler tools, suitable for programming on CPUs.

Compilers

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

```
$ modmap -c compilers
```

The native, and recommended, compilers for LEONARDO Data Centric partition are the Intel ones, since the architecture is based on Intel processors and therefore using the Intel compilers may result in a significant improvement in performance and stability of your code. On the other side, since Intel compilers do not support CUDA, they are not recommended when working on GPUs with LEONARDO Booster partition.

For these reason, CUDA-aware compilers, such as GNU, NVIDIA nvhpc, and CUDA compilers, are suitable and recommended for LEONARDO Booster partition, and they are described in the dedicated page.

Intel OneAPI Compilers

Initialize the environment with the module command:

```
$ module load intel-oneapi-compilers/<VERSION>
```

The suite contains the new Intel oneAPI nextgen compilers (icx, icpx, ifx) and the classic compilers (icc, icpc, ifort):

	Classic	oneAPI	Notes
C /C++ compilers	icc/icpc	icx/icpx	• ICX is the Intel nextgen compiler based on Clang/LLVM technology plus Intel proprietary optimizations and code generation, • ICX enables OpenMP TARGET offload to Intel GPU targets (irrelevant on Galileo100) • ICX and ICC Classic use different compiler drivers. The ICC Classic drivers are icc, icpc, and icl. The ICX drivers are icx and icpx. Use icx to compile and link C programs, and icpx for C++ programs. • Unlike the icc driver, icx does not use the file extension to determine whether to compile as C or C+. Users must invoke icpx to compile C+ files. In addition to providing a core C++ Compiler, ICX is the base compiler for the Intel® oneAPI Data Parallel C++ Compiler and its new driver, dpcpp. • **Intel still recommends ICC/ICPC for standard C/C++ applications**
Fortran compilers	ifort	ifx	• The Intel® Fortran Compiler (Beta) IFX is s a new compiler based on the Intel® Fortran Compiler Classic (ifort) frontend and runtime libraries using LLVM backend technology. ifx is released as a Beta version for users interested in trying offloading to supported Intel GPUs using OpenMP* TARGET directives which ifort does not support (irrelevant on Galileo100) • **Intel recommends IFORT for standard Fortran applications**

NOTE:

- ICX is a new compiler. It has functional and behavioral differences compared to ICC. You can expect some porting will be needed for existing applications using ICC. According to Intel, the transition from ICC Classic to ICX is smooth and effortless. However, you must port and tune any existing applications from ICC Classic to ICX. Please refer to the official Intel Porting Guide for ICC Users to DPCPP or ICX
- IFORT is a completely new compiler. According to Intel, although considerable effort is being made to make the transition from ifort to ifx as smooth and as effortless as possible, customers can expect that some effort may be required to tune their application. IFORT will remain Intel's recommended production compiler until ifx has performance and features superior to ifort. Please refer to the official Intel Porting Guide for ifort Users to ifx
- Please refer to the official Intel C++ Developer Guide and Reference and Fortran Developer Guide and Reference for an exhaustive list of compiler options

After loading the module, the documentation can be obtained with the **man** command:

```
$ man ifort
$ man icc
```

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 behavior 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.

MPI environment

The MPI implementation of Intel, i.e. Intel-OneAPI-MPI, is recommended on the LEONARDO Data Centric partition, and it doesn't support CUDA. Here you can find some useful details on how to use it on this partition.

See the page dedicated to LEONARDO Booster partition for a description of OpenMPI, which instead is installed for supporting CUDA.

Compiling

Intel-OneAPI-MPI

To install MPI applications using IntelMPI you have to load intel-oneapi-mpi module (use "modmap -m intel-oneapi-mpi command to see the available versions).

The intel-oneapi-mpi module provides the following wrappers for classic Intel compilers and OneAPI ("x") compilers.

Compiler	Wrapper	Usage
icpc	mpiicpc	Compile C++ source files with classic Intel
icpx	mpiicpc -cxx=icpx	Compile C++ source files with oneapi
icc	mpiicc	Compile C source files with classic Intel
icx	mpiicc -cc=iccx	Compile C source files with oneapi
ifort	mpiifort (Fortran90/77)	Compile FORTRAN source files with classic Intel
ifx	mpiifort -fc=ifx	Compile FORTRAN source files with oneapi

e.g. Compiling Fortran code:

```
$ module load intel-oneapi-compilers/<VERSION>
$ module load intel-oneapi-mpi/<version>
$ mpiifort -o myexec  myprog.f90 (uses the ifort 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 mpiifort
```

Running

To run MPI applications there are two ways:

- using mpirun launcher
- using srun launcher

mpirun launcher

To use mpirun launcher on LEONARDO Data Centric partition, the **intel-oneapi-mpi** module needs to be loaded:

```
$ module load intel-onepi-mpi/<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 do not support CUDA (see LEONARDO Booster section for GPU-accelerated libraries).

Linear Algebra

- BLAS: openblas, intel-oneapi-mkl
- LAPACK: openblas, intel-oneapi-mkl
- SCALAPACK: netlib-scalapack, intel-oneapi-mkl
- SPARCE MATRICES : PetSc (multi-node), SuperLU-dist (multi-node)

PetSc and SuperLU-dist are GPU-accelerated libraries and are also listed in LEONARDO Booster dedicated page. However, we report them here for the frequent use also in non-accelerated applications.

Fast Fourier Transform

- FFTW (single and multi-node)