

UG3.2: MARCONI100 UserGuide

- [System Architecture](#)
- [Access](#)
- [Accounting](#)
- [Disks and Filesystems](#)
- [Modules environment](#)
- [GPU and intra/inter connection environment](#)
- [Production environment](#)
- [Graphic session](#)
- [Programming environment](#)

hostname: login.m100.cineca.it

early availability: April 20, 2020

start of production: May 4, 2020

This system is an upgrade of the "not conventional" partition of the Marconi Tier-0 system. It is an accelerated cluster based on Power9 chips and Volta NVIDIA GPUs, acquired by Cineca within the [PPI4HPC](#) European initiative.

System Architecture

Architecture: IBM Power 9 AC922

Internal Network: Mellanox Infiniband EDR DragonFly+ 100 Gb/s

Storage: 8 PB (raw) GPFS of local storage

Login nodes: 8 Login IBM Power9 LC922 (similar to the compute nodes)

Model: IBM Power AC922 (Whisperspoon)

Racks: 55 total (49 compute)

Nodes: 980

Processors: 2x16 cores IBM POWER9 AC922 at 2.6(3.1) GHz

Accelerators: 4 x NVIDIA Volta V100 GPUs/node, Nvlink 2.0, 16GB

Cores: 32 cores/node, Hyperthreading x4

RAM: 256 GB/node (242 usable)

Peak Performance: about 32 Pflop/s, 32 TFlops per node

Internal Network: Mellanox IB EDR DragonFly++ 100Gb/s

Disk Space: 8PB raw GPFS storage



More technical details on this architecture can be found on the IBM RedBook series: you can find some of them in our [repository](#) or at the [IBM site for redBooks](#) (search for AC922).

Peak performance details

The peak performance of M100 (980 compute nodes + 8 login nodes) is about 32PFlops. The performance of the single node is 32 TFlops due to 0.8 for the CPU part and 7.8x4 for the four GPUs on the node.

The theoretical peak performance is $988 \times (0.8 + 4 \times 7.8) = 31,6$ PFlop/s

In the below table you can find the single node theoretical peak performance of the AC922 CN node with two Power9 processors and four NVidia V100 GPUs.

Node Performance		
Theoretical Peak Performance	CPU (nominal/peak freq.)	691/791 GFlops
	GPU	31.2 TFlops
	Total	32 TFlops

Memory Bandwidth (nominal/peak freq.)	220/300 GB/s
---------------------------------------	--------------

Access

All the login nodes have an identical environment and can be reached with **SSH (Secure Shell)** protocol using the "collective" hostname:

```
> login.m100.cineca.it
```

which establishes a connection to one of the available login nodes. To connect to Marconi100 you can also indicate explicitly the login nodes:

```
> login01-ext.m100.cineca.it
> login02-ext.m100.cineca.it
```

For information about data transfer from other computers please follow the instructions and caveats on the dedicated section [Data storage](#) or the document [Data Management](#).

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"

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

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

```
> saldo -b (reports projects defined on M100 )
```

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](#).

Budget Linearization policy

On M100, as on the other HPC clusters in Cineca, a linearization policy for the usage of project budgets has been defined and implemented. The goal is to improve the response time, giving users the opportunity of using the cpu hours assigned to their project in relation to their actual size (total amount of core-hours).

Disks and Filesystems

The storage organization conforms to the CINECA infrastructure (see Section [Data Storage and Filesystems](#)).

In addition to the home directory [\\$HOME](#), for each user is defined a scratch area [\\$CINECA_SCRATCH](#), a large disk for the storage of run time data and files.

A [\\$WORK](#) area is defined for each active project on the system, reserved for all the collaborators of the project. This is a safe storage area to keep run time data for the whole life of the project.

	Total Dimension (TB)	Quota (GB)	Notes
\$HOME	200	50	permanent/backed up, user specific, local
\$CINECA_SCRATCH	2.000	no quota	- temporary, user specific, local - no backup - automatic cleaning procedure of data older than 40 days (time interval can be reduced in case of critical usage ratio of the area. In this case, users will be notified via HPC-News)
\$WORK	4.000	1.024	- permanent, project specific, local - no backup - extensions can be considered if needed (mailto: superc@cineca.it)

It is also available a temporary storage local on compute nodes generated when the job starts and accessible via environment variable **\$TMPDIR**. For more details please see the dedicated section of [UG2.5: Data storage and FileSystems](#). On Marconi100 the \$TMPDIR local area has 1 TB of available space.

\$DRES environment variable points to the shared repository where **Data RESources** are maintained. This is a data archive area available only on-request, shared with all CINECA HPC systems and among different projects. \$DRES is not mounted on the compute nodes of the production partitions and can be accessed only from login nodes and from the nodes of the serial partition. This means that you cannot access it within a standard batch job: all data needed during the batch execution has to be moved to \$WORK or \$CINECA_SCRATCH before the run starts, either from the login nodes or via a job submitted to the serial partition.

Since all the filesystems are based on IBM Spectrum Scale™ file system (formerly GPFS), the usual unix command "quota" is not working. Use the local command **cindata** to query for disk usage and quota ("cindata -h" for help):

```
> cindata
```

Dedicated node for Data transfer

A time limit of 10 cpu-minutes for processes running on login nodes has been set.

For Data transfer that may require more time, we set up a dedicated "data" VM accessible with a dedicated alias.

Login via ssh to this VM is not allowed. Environment variables as \$HOME or \$WORK are not defined, so you always have to explicitate the complete path to the files you need to copy.

For example to copy data to M100 using rsync you can run the following command:

```
rsync -PravzHS </data_path_from/file> <your_username>@data.m100.cineca.it:<complete_data_path_to>
```

You can also use the "data" VM onto login nodes to move data from Marconi100 to another location with public IP:

```
ssh -xt data.m100.cineca.it rsync -PravzHS <complete_data_path_from/file> </data_path_to>
```

this command will open a session on the VM that will not be closed until the rsync command is completed.

In similar ways you can use also scp and sftp commands if you prefer them.

Modules environment

The software modules are collected in different profiles and organized by functional categories (compilers, libraries, tools, applications,...). The profiles are of two types: "programming" type (base and advanced) for compilation, debugging and profiling activities, and "domain" type (chem-phys, lifesc,...) for the production activity. They can be loaded together.

"Base" profile is the default. It is automatically loaded after login and it contains basic modules for the programming activities (ibm, gnu, pgi, cuda compilers, math libraries, profiling and debugging tools,...).

If you want to use a module placed under other profiles, for example an application module, you will have to load preventively the corresponding profile:

```
>module load profile/<profile name>
>module load autoload <module name>
```

For listing all profiles you have loaded you can use the following command:

```
>module list
```

In order to detect all profiles, categories and modules available on MARCONI100 the command "modmap" is available. With modmap you can see if the desired module is available and which profile you have to load to use it.

```
>modmap -m <module_name>
```

Spack environment

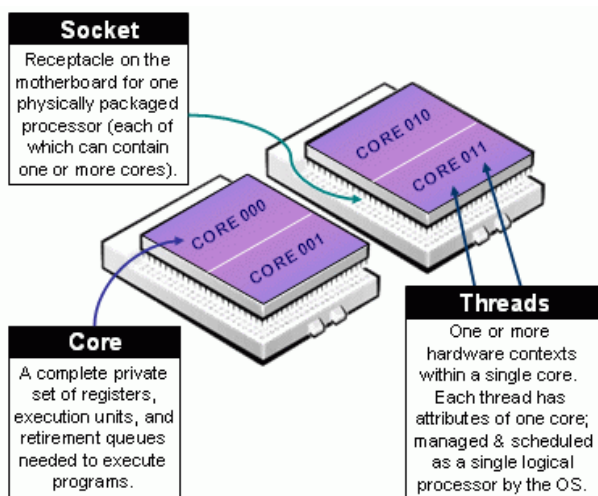
In case you don't find a software you are interested in, you can install it by yourself.

In this case, on Marconi100 we also offer the possibility to use the "spack" environment by loading the corresponding module. Please refer to the [dedicated section](#) in UG2.6: Production Environment

GPU and intra/inter connection environment

Marconi100 consists of 980 compute nodes and 8 login nodes, connected with a Mellanox Infiniband EDR network arranged into an architecture called DragonFly ++.

The login nodes and the compute nodes are exactly the same. Each node consists in 2 Power9 **sockets**, each of them with 16 **cores** and 2 Volta **GPUs** (32 cores and 4 GPUs per node). The **multi-threading** is active with 4 threads per physical core (128 total threads – or logical cpus – per node).



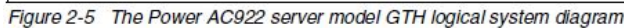
Due to how the hardware is detected on a Power9 architecture, the numbering of (logical) cpus follows the order of threading. You can get it with the `ppc64_cpu` command:

```
$ ppc64_cpu --info
Core 0: 0* 1* 2* 3*
Core 1: 4* 5* 6* 7*
Core 2: 8* 9* 10* 11*
Core 3: 12* 13* 14* 15*
..... (Cores from 4 to 28).....
Core 29: 116* 117* 118* 119*
Core 30: 120* 121* 122* 123*
Core 31: 124* 125* 126* 127*
```

The topology of the node can be visualized by running the command `nvidia-smi` as follows:

```
$ nvidia-smi topo -m
GPU0    GPU1    GPU2    GPU3    CPU Affinity
GPU0     X      NV3     SYS     SYS     0-63
GPU1    NV3     X      SYS     SYS     0-63
GPU2     SYS     SYS     X      NV3     64-127
GPU3     SYS     SYS     NV3     X      64-127
```

From the output of the command it is possible to see that GPU0 and GPU1 are connected with the NVLink (NV3), as well as the couple GPU2 & GPU3. The first couple is connected to (virtual) cpus 0-63 (on the first socket), the second to (virtual) cpus 64-127 (on the second socket). The cpus are numbered from 0 to 127 because of a hyperthreading. The two Power9 sockets are connected by a 64 GBps X bus. Each of them is connected with 2 GPUs via NVLink 2.0.



The internode communications is based on a Mellanox Infiniband EDR network 100 Gb/s, and the OpenMPI and IBM MPI Spectrum libraries are configured so to exploit the Mellanox Fabric Collective Accelerators (also on CUDA memories) and Messaging Accelerators.

Production environment

Marconi100 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 GPUs. 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.

Warning: When you compile your code using the XL compiler with **Spectrum-MPI** parallel library (our [recommended software stack](#)) you **have to use** **mpirun** (not **srun**) to execute your program.

```
> ./program
```

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 four GPUs) for one hour, launch the command:

```
>
> salloc -N1 --exclusive --gres=gpu:4 -A <account_name> -p <partition_name> --time=01:00:00
salloc: Granted job allocation 1174
...
Run here your applications using srun or mpirun for example:
- srun -n 1 serial.exe (serial or OpenMP application)
- srun -n 4 parallel.exe (MPI or mixed application)
Stop the session with exit or ^D
exit
salloc: Relinquishing job allocation 1174
>
```

In the following is reported an interactive job on 2 cores and two GPUs. Within the job a parallel (MPI) program using 2 MPI tasks and two GPUs is executed. Since the request of tasks per node (--ntasks-per-node) refers to the 128 (virtual) cpus, if you want 2 physical cores you also have to specify that each task is made of 4 (virtual) cpus (--cpus-per-task=4).

```
> salloc -N1 --ntasks-per-node=2 --cpus-per-task=4 --gres=gpu:2 -A <account_name> -p <partition_name> --time=01:00:00
salloc: Granted job allocation 1175

> srun ./myprogram
...
> exit
```

SLURM automatically exports the environment variables you defined in the source shell, so that if you need to run your program "myprogram" in a controlled environment (i.e. specific library paths or options), you can prepare the environment in the origin shell being sure to find it in the interactive shell.

A more specific description of the options used by salloc/srun to allocate resources or to give direction to SLURM on how to place tasks and threads on the resources (pinning) is reported later in the "Batch" section, because they are the same of the sbatch command described there.

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 Marconi100) that will search for the required resources in the system. As soon as the resources are available script.x is executed and the results are sent back to the user.

This is an example of script file:

```
#!/bin/bash
#SBATCH -A <account_name>
#SBATCH -p m100_usr_prod
#SBATCH --time 00:10:00      # format: HH:MM:SS
#SBATCH -N 1                # 1 node
#SBATCH --ntasks-per-node=8 # 8 tasks out of 128
#SBATCH --gres=gpu:1        # 1 gpus per node out of 4
#SBATCH --mem=7100          # memory per node out of 246000MB
#SBATCH --job-name=my_batch_job
#SBATCH --mail-type=ALL
#SBATCH --mail-user=<user_email>
mpirun ./myexecutable      #in case you compiled with spectrum-mpi
OR
srun ./myexecutable        #in all the other cases
```

Please note that by requesting --ntasks-per-node=8 your job will be assigned 8 logical cpus (hence, the first 2 cpus with their 4 HTs). 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_number (-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.

One of the commands will be probably the launch of a parallel MPI application. In this case the right command is srun, as an alternative to the usual mpirun command. In this way you will get full support for process tracking, accounting, task affinity, suspend/resume and other features.

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 M100.

Note: **core** refers to a physical cpu, with its 4 HTs; **cpu** refers to a logical cpu (1 HT). Each node has 32 cores/128 cpus.

SLURM partition	Job QOS	# cores/# GPU per job	max walltime	max running jobs per user/ max n. of cores/nodes/GPUs per user	priority	notes
m100_all_serial (default)	<i>normal</i>	max = 1 core, 1 GPU max mem= 7600MB	04:00:00	4 cpus/1 GPU	40	
	<i>qos_install</i>	max = 16 cores	04:00:00	max = 16 cores 1 job per user	40	request to superc@cineca.it
m100_usr_prod	<i>normal</i>	max = 16 nodes	24:00:00		40	runs on 880 nodes
	m100_qos_dbg	max = 2 nodes	02:00:00	2 nodes/64cores/8GPUs	80	runs on 12 nodes
	m100_qos_bprod	min = 17 nodes max =256 nodes	24:00:00	256 nodes	60	runs on 512 nodes min is 17 FULL nodes (544 cores, 2176 cpus)
m100_usr_preempt	<i>normal</i>	max = 16 nodes	24:00:00		1	runs on 99 nodes
m100_fua_prod (EUROFUSION)	<i>normal</i>	max = 16 nodes	24:00:00		40	runs on 87 nodes
	m100_qos_fuadbq	max = 2 nodes	02:00:00		45	runs on 12 nodes
	m100_qos_fuabprod	min = 17 nodes max = 32 nodes	24:00:00		40	run on 64 nodes at same time
all partitions	qos_special	> 32 nodes	> 24:00:00		40	request to superc@cineca.it
all partitions (NO EUROFUSION)	qos_lowprio	max = 16 nodes	24:00:00		0	active projects with exhausted budget request to superc@cineca.it

The **partition m100_usr_preempt** allows users to access the additional nodes of m100_fua_prod partition in preemptable modality (if available and not used by Eurofusion community). The jobs submitted to the m100_usr_preempt partition may be killed if the assigned resources are requested by jobs submitted to higher priority partition (m100_fua_prod); hence we recommend its use only with restartable applications.

Users with **exhausted but still active projects** are allowed to keep using the cluster resources, even if at a very low priority, by adding the "**qos_lowprio**" flag to their job:

```
#SBATCH --qos=qos_lowprio
```

This qos is **not active for EUROFusion projects** for which a **different dedicated QOS (qos_fualowprio)** is automatically associated to Eurofusion users once their projects exhaust the budget before their expiry date.

For all the other users, please ask superc@cineca.it the QOS association.

Eurofusion users can also use the computing resources at low priority before their budget gets exhausted, in case they wish to run non urgent jobs without consuming the budget of the granted project. Please ask superc@cineca.it to be added to the Account **FUAC6_LOWPRIO**, and specify this account and the **qos_fualowprio** QOS in your submission script.

M100 specific information

In the following we report information specific to M100, as well as examples suited for this kind of system.

Each node exposes itself to SLURM as having 128 (virtual) cpus, 4 GPUs and 246.000 MB memory. SLURM assigns a node in shared way, assigning to the job only the resources required and allowing multiple jobs to run on the same node/nodes. If you want to have the node/s in exclusive mode, use the SLURM option "--exclusive" together with "--gres=gpu:4".

The maximum memory which can be requested is 246.000 MB (average memory per physical core ~ 7GB) and this value guarantees that no memory swapping will occur.

Even if the nodes are shared among users, **exclusivity is guaranteed** for the single physical core and the single GPU. When you ask for "tasks" (--ntasks-per-node), SLURM gives you the requested number of (virtual) cpus rounded on multiple of four. For example

```
#SBATCH --ntasks-per-node = 1 (or 2, 3, 4)      1 core
#SBATCH --ntasks-per-node = 13 (or 14, 15, 16)  4 cores
```

By default the number of (virtual) cpus per task is one, but you can change it.

```
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=4
```

In this way each tasks will correspond to one (physical) core.

M100 GPU use report

A statistics of the GPU utilization during a job, provided by nvidia dcgmi tool, can be obtained at the end of the job by explicitly requesting the constraining "gpureport":

```
#SBATCH --constraint=gpureport (or -C gpureport)
```

This option will result, at the end of the job, in producing a file for each of the nodes assigned to the job with the relevant information on the employed GPUs (performance statistics, such as Energy Consumed, Power Usage, Max GPU Memory Used, GPU and Memory Used etc.; Event Stats, as ECC Errors etc.; Slowdown Stats; Overall Health). The files are named "dcgmi_stats_<node_name>_<jobid>.out".

Submitting serial batch jobs

The m100_all_serial partition is available with a maximum walltime of 4 hours, 1 core and 7600 MB per job. It runs on two dedicated nodes (equipped with 4 Volta GPUs), and it is designed for pre/post-processing serial analysis (using or not the GPUs), for moving your data (via rsync, scp etc.), and for programming tools.

```
#SBATCH -p m100_all_serial
```

This is the default partition, the use of this partition is free of charge and available to all users on the cluster.

Submitting batch jobs for production

Not all of the partitions are open to access by the academic community as some are reserved to dedicated classes of users (for example m100_fua_* partitions are for EUROfusion users):

- m100_fua_* partitions are reserved to EUROfusion users
- m100_usr_* partitions are open to academic production.

In these partitions you can use also the QOS directives, in order to modulate your request:

```
#SBATCH -p m100_usr_prod
#SBATCH --qos=m100_qos_dbg
```

(debug queue for academic users)

```
#SBATCH -p m100_usr_prod
```

(production queue for academic users)

Examples

```
> salloc -N1 --ntasks-per-node=2 --cpus-per-task=4 --gres=gpu:2 --partition=...
export OMP_NUM_THREADS=4
mpirun ./myprogram
```

Two full cores on one node are requested, as well as 2 GPUs. A hybrid code is executed with 2 MPI tasks and 4 OMP threads, exploiting the HT capability of M100. Since 2 GPUs are used, 16 cores will be accounted to this job.

With Spectrum MPI you need to launch your parallel program with "mpirun". For the OpenMPI environment, both srun and mpirun can be used.

```
> salloc -N1 --ntasks-per-node=4 --cpus-per-task=16 --gres=gpu:2 --partition=...
```

With Spectrum MPI:

```
export OMP_NUM_THREADS=16
mpirun -n 4 --map-by socket:PE=4 ./myprogram
```

With OpenMPI

```
export OMP_NUM_THREADS=16
export OMP_PLACES=threads

srun ./myprogram
```

16 full cores are requested and 2 GPUs. The 16x4 (virtual) cpus are used for 4 MPI tasks and 16 OMP threads per task. The -m flag in the srun command specifies the desired process distribution between nodes/socket/cores (the default is block:cyclic). Please refer to srun manual for more details on the processes distribution and binding. The --map-by socket:PE=4 will assign and bind 4 physical consecutive cores to each process (see process mapping and binding on the [official IBM Spectrum MPI manual](#)).

```
> salloc -N1 --ntasks-per-node=32 --cpus-per-task=4 --gres=gpu:2 --partition=...
```

```
export OMP_NUM_THREADS=4
mpirun ./myprogram
```

32 full cores are requested and 2 GPUs. The 32x4 (virtual) cpus are used for 32 MPI tasks and 4 OMP threads. In this way you are asking for entire node and you can ask for 2 or 3 or 4 GPUs, because you can obtain only cores related to the requested GPUs.

Here you can find [Other batch job examples on M100](#). You can find more information on process mapping and binding on the [official IBM Spectrum MPI manual](#).

Graphic session

If a graphic session is desired we recommend to use the tool [RCM \(Remote Connection Manager\)](#). For additional information visit [Remote Visualization](#) section on our User Guide.

Programming environment

Marconi100 login and compute nodes host four Tesla Volta (V100) GPUs per node (CUDA compute capability 7.0). The most recent versions of nVIDIA CUDA toolkit and of the Community Edition PGI compilers (supporting CUDA Fortran) is available in the module environment, together with a set of GPU-enabled libraries, applications and tools.

The programming environment of the M100 cluster consists of a choice of compilers for the main scientific languages (Fortran, C and C++), debuggers to help users in finding bugs and errors in the codes, profilers to help in code optimisation.

In general, you must "load" the correct environment also for using programming tools like compilers, since "native" compilers are not available.

If you use a given set of compilers and libraries to create your executable, very probably you have to define the same "environment" when you want to run it. This is because, since by default linking is dynamic on Linux systems, at runtime the application will need the compiler shared libraries as well as other proprietary libraries. This means that you have to specify "module load" for compilers and libraries, both at compile time and at run time. To minimize the number of needed modules at runtime, use static linking to compile the applications.

Compilers

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

```
> module available
```

and checking the "compilers" section. The available compilers are:

- XL
- NVIDIA HPC-SDK (ex PGI)
- GNU
- CUDA

XL

The XL compiler family offers C, C++, and Fortran compilers designed for optimization and improvement of code generation, exploiting the inherent opportunities in Power Architecture. This is the **recommended software stack** on M100, together with Spectrum-MPI parallel library and ESSL scientific library.

The xl module provides:

- IBM XL C/C++ and Fortran compilers
- IBM XL Shared-memory parallelism (SMP) runtime library/environment
- Mathematical Acceleration Subsystem (MASS) Libraries

The name of the XL C/C++ and Fortran compilers are:

Invocations	Usage (supported standards)
xlc	Compile C source files. (ANSI C89, ISO C99, IBM language extensions)
xlc++	Compile C++ source files.
xlF	Compile FORTRAN 77 source files
xlF90	Compile FORTRAN 90 source files
xlF95	Compile FORTRAN 95 source files
xlCuf	Compile CUDA FORTRAN source files

All invocations with a suffix of `_r` (eg. `xlc_r`, `xlF_r`,...) allow for thread-safe compilation. Use these commands to create threaded applications or to link programs that use multi-threading.

The OpenMP parallelization is enabled by the `-qsmp` compiler option. If `-qsmp=omp` is specified, strict OpenMP compliance is applied on the compiling programs. Please refer to the official [OpenMP support in IBM XL compilers](#) documentation.

To learn more about the XL C/C++ and Fortran compilers, access the online product documentation in IBM Knowledge Center for the [XL C/C++ compiler](#) and the [XL Fortran compiler](#).

NVIDIA HPC SDK (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 HPC SDK suite also offers the PGI C, C++, and Fortran compilers with their original names:

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

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

GNU compilers

The gnu compilers are always available but they are not the best optimizing compilers, it ensures the maximum portability. You do not need to load the module for using them.

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
```

CUDA

Compute Unified Device Architecture 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.

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:

XL Fortran compiler

```
-O0      Lower level of optimization, you can use also -O2
-g       Produce debugging information
```

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

Totalview

The TotalView debugger is a programmable tool that lets you debug, analyze, and tune the performance of complex serial, multiprocessor, and multithreaded programs.

TotalView has many features and it gives you a great number of tools for finding your program's problems.

Details on how to use totalview are in

<https://docs.roguewave.com/en/totalview/current/html/>

Scalasca

Scalasca is a tool for profiling parallel scientific and engineering applications that make use of MPI and OpenMP.

Details how to use scalasca in

<http://www.scalasca.org/software/scalasca-2.x/documentation.html>

PGI: pgdbg (serial/parallel debugger)

pgdbg is the Portland Group Inc. symbolic source-level debugger for F77, F90, C, C++ and assembly language programs. It is capable of debugging applications that exhibit various levels of parallelism.

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 (gprof)

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.

GPU profilers (Nvidia Nsight System)

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](#).

On Marconi100 **only the Command Line Interface (CLI) is available** since the GUI does not support Power9 nodes. 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 modules hpc-sdk, cuda/11.0 and later versions.

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 /scratch_local/slurm_job.\$SLURM_JOB_ID where you have 1 TB 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
```

Nsight Systems can also collect kernel IP samples and backtraces, however, this is prevented by the perf event paranoid level being set to 2 on Marconi100. It is possible to bypass this restriction by adding the SLURM directive:

```
#SBATCH --gres=sysfs
```

Along with the *exclusive* one.

MPI environment

We offer two options for MPI environment on Marconi100:

```
> Spectrum-MPI
> OpenMPI
```

Here you can find some useful details on how to use them on Marconi100.

Warning: When you compile your code using the XL compiler with **Spectrum-MPI** parallel library (our [recommended software stack](#)) you **have to use mpirun** (not srun) to execute your program.

Spectrum-MPI

It is an IBM implementation of MPI. Together with XL compiler it is the **recommended enviroment** to be used on Marconi100.

In addition to OpenMPI it adds unique features optimized for IBM systems such as CPU affinity features, dynamic selection of interface libraries, workload manager integrations and better performance.

Spectrum-MPI supports both CUDA-aware and GPUDirect technologies.

The spectrum_mpi module provides the following compiler wrappers:

compiler	wrapper	usage
xlC_r	mpixlc	Compile C source files with XL
xlC_r	mpixlC	Compile C++ source files with XL
xlF_r	mpixlf	Compile FORTRAN source files with XL
pgcc	mpipgcc	Compile C source files with PGI
pgc++	mpipgic++	Compile C++ source files with PGI
pgfortran	mpipgifort	Compile FORTRAN source files with PGI
gcc	mpicc	Compile C source files with GNU

g++	mpic++ mpiCC mpicxx	Compile C++ source files with GNU
gfortran	mpif77 mpif90 mpifort	Compile FORTRAN source files with GNU

Detailed documentation can be found in the [original IBM website](#).

Important flags

- In order to enable the GPU awareness you need to run **mpirun** with the flag **-gpu**. This is required for applications that pass pointers to GPU buffers to MPI API calls.

OpenMPI

This most common MPI implementation is installed inside the GNU environment. It is configured to support both CUDA-aware and GPUDirect.

The openmpi module provides the following wrappers:

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

Scientific libraries

ESSL: Engineering and Scientific Subroutine Library by IBM

Scientific libraries designed for Power architecture included in the XL compiler package,

```
> module load essl
```

ESSL includes blas lapack and fftw (but not scalapack). Unfortunately the lapacks included are not complete, so if you need lapack or scalapack you need to add the flag:

-lessl: single-threaded routines

-lesslsmp: multi-threaded routines

The correct way is to link essl before lapacks, for example:

```
-L $ {ESSL_LIB} -lesslsmp -L {LIB_LAPACK} -llapack
```

Other option is to use the openBLAS (instead of lapack and BLAS) and in some case they are more efficient.

Documentation: https://www.ibm.com/support/knowledgecenter/SSFHY8/essl_content.html