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 PP4HPC European initiative.

### System Architecture

**Architecture:** IBM Power 9 AC922  
**Internal Network:** Mellanox Infiniband EDR DragonFly+  
**Storage:** 8 PB (raw) GPFS of local storage  
**Login nodes:** 8 Login IBM Power9 LC922 (similar to the compute nodes)
Model: IBM Power AC922 (Whiterspoon)

- Racks: 55 total (49 compute)
- Nodes: 980
- Processors: 2x16 cores IBM POWER9 AC922 at 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)
- Internal Network: Mellanox IB EDR DragonFly++
- Disk Space: 8PB raw GPFS storage

More technical details on this architecture can be found on the IBM RedBook series: you can find same 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*(0.8+4*7.8) = 31.6 PFlop/s.

In the below table you can find the single node theoretical peak performance of the AC992 CN node with two Power9 processors and four NVIdia V100 GPUs.

<table>
<thead>
<tr>
<th>Node Performance</th>
<th>Theoretical Peak Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (nominal/peak freq.)</td>
<td>691/791 GFlops</td>
</tr>
<tr>
<td>GPU</td>
<td>31.2 TFlops</td>
</tr>
<tr>
<td>Total</td>
<td>32 TFlops</td>
</tr>
<tr>
<td>Memory Bandwidth (nominal/peak freq.)</td>
<td>220/300 GB/s</td>
</tr>
</tbody>
</table>

**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
> login03-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.

<table>
<thead>
<tr>
<th></th>
<th>Total Dimension (TB)</th>
<th>Quota (GB)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$HOME</td>
<td>200</td>
<td>50</td>
<td>• permanent/ backed up, user specific, local</td>
</tr>
<tr>
<td>$CINECA_SCRATCH</td>
<td>2.000</td>
<td>no quota</td>
<td>• 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)</td>
</tr>
<tr>
<td>$WORK</td>
<td>4.000</td>
<td>1.024</td>
<td>• permanent, project specific, local • no backup • extensions can be considered if needed (mailto: <a href="mailto:superc@cineca.it">superc@cineca.it</a>)</td>
</tr>
</tbody>
</table>

$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

**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, "domain" type (chem, phys, lifesc...) for the production activity and "programming" type (base and advanced) for compilation, debugging and profiling activities and that 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 (intel e gnu 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:

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

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

```bash
> 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.

```bash
> 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 UG2.8: 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).

```bash
$ 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

<table>
<thead>
<tr>
<th></th>
<th>GPU0</th>
<th>GPU1</th>
<th>GPU2</th>
<th>GPU3</th>
<th>CPU Affinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU0</td>
<td>X</td>
<td>NV3</td>
<td>SYS</td>
<td>SYS</td>
<td>0-63</td>
</tr>
<tr>
<td>GPU1</td>
<td>NV3</td>
<td>X</td>
<td>SYS</td>
<td>SYS</td>
<td>0-63</td>
</tr>
<tr>
<td>GPU2</td>
<td>SYS</td>
<td>SYS</td>
<td>X</td>
<td>NV3</td>
<td>64-127</td>
</tr>
<tr>
<td>GPU3</td>
<td>SYS</td>
<td>SYS</td>
<td>NV3</td>
<td>X</td>
<td>64-127</td>
</tr>
</tbody>
</table>
```

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 knowledge of the topology of the node is important for correctly distribute the parallel threads of your applications in order to get the best performances.

The internode communications is based on a Mellanox Infiniband EDR network, 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.

nVIDIA GPUDirect technology is fully supported (shared memory, peer-to-peer, RDMA, async), enabling the use of CUDA-aware MPI.

### Production environment

Since M100 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 Marconi100 the available scheduler is SLURM.

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.

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

**Interactive**

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

```
> ./program
```

This is allowed only for very short runs on the login nodes, since the *interactive environment has a 10 minutes cpu-time limit*. Please do not execute parallel applications on the login nodes!

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 the `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 are 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 and 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
```
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.

<table>
<thead>
<tr>
<th>SLURM partition</th>
<th>Job QOS</th>
<th># cores/# GPU per job</th>
<th>max walltime</th>
<th>max n. of cpus/nodes/GPUs per user</th>
<th>priority</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>m100_all_serial (def. partition)</td>
<td>normal</td>
<td>max = 1 core, 1 GPU max mem= 7600MB</td>
<td>04:00:00</td>
<td>4 cpus/1 GPU</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>m100_usr_prod</td>
<td>m100_qos_dbg</td>
<td>max = 2 nodes</td>
<td>02:00:00</td>
<td>2 nodes/64 cpus/8 GPUs</td>
<td>45</td>
<td>runs on 12 nodes</td>
</tr>
<tr>
<td>m100_usr_prod</td>
<td>normal</td>
<td>max = 16 nodes</td>
<td>24:00:00</td>
<td>10 jobs</td>
<td>40</td>
<td>runs on 880 nodes</td>
</tr>
<tr>
<td>m100_qos_bprod</td>
<td>min = 17 nodes max =256 nodes</td>
<td>24:00:00</td>
<td>256 nodes</td>
<td>85</td>
<td>runs on 256 nodes</td>
<td></td>
</tr>
<tr>
<td>m100_fua_prod</td>
<td>m100_qos_fuadbg</td>
<td>max = 2 nodes</td>
<td>02:00:00</td>
<td>45</td>
<td>runs on 12 nodes</td>
<td></td>
</tr>
<tr>
<td>m100_fua_prod</td>
<td>normal</td>
<td>max = 16 nodes</td>
<td>24:00:00</td>
<td>40</td>
<td>runs on 68 nodes</td>
<td></td>
</tr>
<tr>
<td>qos_special</td>
<td>&gt;16 nodes</td>
<td>&gt;24:00:00</td>
<td>40</td>
<td>request to <a href="mailto:superc@cineca.it">superc@cineca.it</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qos_lowprio</td>
<td>max = 16 nodes</td>
<td>24:00:00</td>
<td>0</td>
<td>active projects with exhausted budget</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

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 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.

### 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_fu a_* 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

```
> s alloc -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.

```bash
> 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.

**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:

```bash
> module available
```

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

- XL
- 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:

<table>
<thead>
<tr>
<th>Invocations</th>
<th>Usage (supported standards)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlc</td>
<td>Compile C source files.</td>
</tr>
<tr>
<td></td>
<td>(ANSI C89, ISO C99, IBM language extensions)</td>
</tr>
<tr>
<td>xlc++</td>
<td>Compile C++ source files.</td>
</tr>
<tr>
<td>xlf</td>
<td>Compile FORTRAN 77 source files</td>
</tr>
<tr>
<td>xlf90</td>
<td>Compile FORTRAN 90 source files</td>
</tr>
<tr>
<td>xlf95</td>
<td>Compile FORTRAN 95 source files</td>
</tr>
<tr>
<td>xlcuf</td>
<td>Compile CUDA FORTRAN source files</td>
</tr>
</tbody>
</table>

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 Fortran for Linux compiler, access the online product documentation in IBM Knowledge Center for the XL C/C++ compiler and the XL Fortran compiler.

PORTLAND Group (PGI)

The name of the PGI compilers are:

- pgf77: Fortran77 compiler
- pgf90: Fortran90 compiler
- pgf95: Fortran95 compiler
- pgphf: High Performance Fortran compiler
- pgcc: C compiler
- pgCC: C++ compiler

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

> man pgf95
> man pgcc

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 gfortran
> 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


**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:

```bash
> 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
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.

**Spectrum-MPI**

It is an IBM implementation of MPI. Together with XL compiler it is the **recommended environment** 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:

<table>
<thead>
<tr>
<th>compiler</th>
<th>wrapper</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlc_r</td>
<td>mpixlc</td>
<td>Compile C source files with XL</td>
</tr>
<tr>
<td>xIC_r</td>
<td>mpixIC</td>
<td>Compile C++ source files with XL</td>
</tr>
<tr>
<td>xlf_r</td>
<td>mpixlf</td>
<td>Compile FORTRAN source files with XL</td>
</tr>
<tr>
<td>pgcc</td>
<td>mpipgicc</td>
<td>Compile C source files with PGI</td>
</tr>
<tr>
<td>pgc++</td>
<td>mpipgic++</td>
<td>Compile C++ source files with PGI</td>
</tr>
<tr>
<td>pgfortran</td>
<td>mpigfort</td>
<td>Compile FORTRAN source files with PGI</td>
</tr>
<tr>
<td>gcc</td>
<td>mpicc</td>
<td>Compile C source files with GNU</td>
</tr>
<tr>
<td>g++</td>
<td>mpiCC  mpiCCxx</td>
<td>Compile C++ source files with GNU</td>
</tr>
<tr>
<td>gfortran</td>
<td>mpif77 mpif90 mpifort</td>
<td>Compile FORTRAN source files with GNU</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>compiler</th>
<th>wrapper</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>mpicc</td>
<td>Compile C source files with GNU</td>
</tr>
<tr>
<td>g++</td>
<td>mpiCC  mpiCCxx</td>
<td>Compile C++ source files with GNU</td>
</tr>
<tr>
<td>gfortran</td>
<td>mpif77 mpif90 mpifort</td>
<td>Compile FORTRAN source files with GNU</td>
</tr>
</tbody>
</table>
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