UG3.3: GALILEO UserGuide

In this page:

---

hostname: login.galileo.cineca.it

early availability: 05/03/2018

start of production: 12/03/2018

system upgrade in production: 19/08/2019

```
Model: Lenovo NeXtScale
Architecture: Linux Infiniband Cluster
Nodes: 1022
Processors: 2 x 18-cores Intel Xeon E5-2697 v4 (Broadwell) at 2.30 GHz

Cores: 36 cores/node
RAM: 128 GB/node, 3.5 GB/core
Internal Network: Intel OmniPath, 100 Gb/s
Peak performance single node: 1.3 TFlop/s
Peak Performance: 1.5 PFlop/s

+

Accelerators: 60 nodes equipped with 1 nVidia K80 GPU
               2 nodes equipped with 1 nVidia V100 GPU
```

Starting from 15th March 2021 Galileo will be turned off to make space for the new more performant infrastructure Galileo100.

The GALILEO supercomputer has been introduced first in January 2015 and in its first configuration, it has been available to the Italian public and industrial researchers until January 2018. It has been the national Tier-1 system for scientific research.

Starting from January 2018 GALILEO has been reconfigured with Intel Xeon E5-2697 v4 (Broadwell) nodes, inherited from MARCONI system. The cluster with this reconfiguration has opened for production in March of the same year.

Starting from August 2019 a new reconfiguration phase led to a new, upgraded environment, equipped with Intel Omni-Path internal network and increasing the number of compute nodes available to 1022 Intel Xeon E5-2697 v4 (Broadwell).

System Architecture

Compute Nodes: There are currently 1022 36-core compute nodes. Each one contains 2 18-cores Intel Xeon E5-2697 v4 (Broadwell) at 2.30 GHz. All the compute nodes have 128 GB of memory. Of these compute nodes, 60 are equipped with nVidia K80 GPU and two with nVidia V100.

Login and Service nodes: There are 8 Login & Viz nodes available (5 are academic nodes and 3 are for industrial users). There are 8 service nodes for I /O and cluster management.

All the nodes are interconnected through an Infiniband network, with OPA v10.6, capable of a maximum bandwidth of 100Gbit/s between each pair of nodes.

Accounting

For more information about accounting, please consult our **dedicated section**.

Budget Linearization policy

On GALILEO a linearization policy for the usage of project budgets has been defined and implemented. For each account, a monthly quota is defined as:

```
monthTotal = (total_budget / total_no_of_months)
```

Starting from the first day of each month, the collaborators of any account are allowed to use the quota at full priority. As long as the budget is consumed, the jobs submitted from the account will gradually lose priority, until the monthly budget (monthTotal) is fully consumed. At that moment, their jobs will still be considered for execution, but with a lower priority than the jobs from accounts that still have some monthly quota left.

This policy is similar to those already applied by other important HPC centers in Europe and worldwide. 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 organisation 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 storing run time data and files. **$WORK** 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	300	no quota	• temporary, user specific, local • automatic cleaning procedure of data older than 50 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	300	1.000	• permanent, project specific, local • 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 Galileo the $TMPDIR local area has 49 GB of available space.

$DRES points to the shared repository where **D**ata **RES**ources 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. This means that you **can't access it within a batch job**: all data needed during the batch execution has to be moved on $WORK or $CINECA_SCRATCH before the run starts.

Since all the filesystems are based on gpfs (General Parallel FIle System), 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 category (compilers, libraries, tools, applications,..).

On GALILEO 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:

```
>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 GALILEO the command “modmap” is available:

```
>modmap
```

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>
```

Production environment

Since GALILEO 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*. This guarantees that access to the resources is as fair as possible.

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.

A parallel program can be executed interactively only by submitting an "Interactive" SLURM batch job, using the "srun" command: the job is queued and scheduled as any other job, but when executed, the standard input, output, and error streams are connected to the terminal session from which srun was launched.

For example, to start an interactive session with the MPI program "myprogram", using one node and two processors, you can launch the command:

```
> srun -N 1 --ntasks-per-node=2 -A <account_name> --pty /bin/bash
```

SLURM will then schedule your job to start, and your shell will be unresponsive until free resources are allocated for you. If not specified, the default time limit for this kind of jobs is one hour.

When the shell returns a prompt inside the compute node, you can execute your program by typing:

```
> srun ./myprogram
```

or

```
> mpirun ./myprogram
```

The default SLURM MPI type has been set equal to PMI2.

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. with specific library paths or options), you can prepare the environment in the login shell and be sure to find it again in the interactive shell o the compute node.

Batch

As usual, on systems using SLURM, you can submit a script *script.x* using the command:

```
> sbatch script.x
```

You can get a list of defined partitions with the command:

```
> sinfo
```

For more information and **examples of job scripts**, see section **Batch Scheduler SLURM**.

Submitting serial Batch jobs

The **gll_all_serial** partition is available with one core and a maximum walltime of 4 hours. It runs on two nodes and it is designed for pre/post-processing serial analysis, and for moving your data (via rsync, scp etc.) in case more than 10 minutes are required to complete the data transfer. In order to use this partition you have to specify the **SLURM flag "-p"**:

```
#SBATCH -p gll_all_serial
```

The gll_all_serial partition has a limit of 1 task per job and 4GB of memory per job. If you wish to ask for more than a core on a single job, remember to add on your jobscript the specific about the memory limit, since the default per core is 3.5GB and therefore your job won't enter because the required memory exceeds the partition limit.

IMPORTANT FOR USERS OF THE PRE-UPGRADED VERSION OF GALILEO: the old filesystems (such as $CINECA_SCRATCH) haven't been migrated to the upgraded cluster, but will remain available for a certain period of time (until the end of 2019) and visible only on specific nodes. You can access such nodes for transferring your data from the old environment to the new, by using the partition:

```
#SBATCH -p gll_all_transfer
```

with the limitations of one core and 24 hours per job. Inside the gll_all_transfer nodes, you will find your old scratch area at the mount point /gpfs /scratch_old .

Graphic session

If a graphic session is desired we recommend to use the tool "RCM". See the corresponding session to know more about how to download and use RCM.

A complete reconfiguration of the RCM environment is in progress. This guide will be completed as soon as a final configuration will be implemented.

Submitting parallel Batch jobs

To run parallel batch jobs on GALILEO you need to specify the partition gll_usr_prod, or any other partition described in this user guide.

Users who need to run on GPU-equipped nodes need to specify the partition gll_usr_gpuprod.

If you do not specify the partition, your jobs will try to run on the default partition bdw_all_serial, meant for serial jobs, eventually failing if specific partition limits (maximum four tasks per job and maximum walltime of 4 hours) are violated.

The minimum number of cores you can request for a batch job is 1. The **maximum number of cores** that you can request is 2304 (64 nodes). It is also possible to request a **maximum walltime of 24 hours**. Defaults are as follows:

- If you do not specify the walltime (by means of the #SBATCH --time directive), a default value of 30 minutes will be assumed.
- If you do not specify the number of cores (by means of the "SBATCH -n" directive) a default value of 36 will be assumed.
- If you do not specify the amount of memory (as the value of the "SBATCH --mem" DIRECTIVE), a default value of 3000MB will be assumed.
- The maximum memory per node is **118000MB**

A special QOS (qos_special) is also available for not-ordinary types of jobs, such as a walltime larger than 24 hours. Since it violates our standard policy, there are restrictions in its usage, and users who want to use it need to be enabled by the User Support staff. Please write to superc@cineca.it in case you think you need to use it. Your request will be evaluated and, if approved, you will be allowed to use the special QOS for a limited period of time.

Use of GPUs on GALILEO

The gll_usr_gpuprod partition is defined on 42 Broadwell nodes (18-cores Intel Xeon E5-2697 v4 @ 2.40GHz), each equipped with 1 nVidia K80 GPUs (seen as two K40 gpus). All users using an account with available budget can submit jobs on this partition and use GPU nodes on GALILEO.

The maximum number of nodes that can be required on gll_usr_gpuprod is **4** for a maximum walltime of **08:00:00 hours**. The maximum memory is **118000 MB**.

In regards of writing a SLURM jobscript, you need to request the GPU as "gres":

```
#SBATCH --partition=gll_usr_gpuprod
#SBATCH --gres=gpu:kepler:N     (N=1,2)
```

GALILEO is also equipped with two nodes with one nVIDIA Volta (V100) GPUs each, accessible for tests for a limited period of time. Please write to superc @cineca.it if you are interested to test the Volta GPUs, with a brief motivation for your request. Once your request is approved and you are enabled to use these resources (via the association to a special QOS "gll_qos_gpudev"), you can submit jobs to the Volta node with the following options:

```
#SBATCH --partition=gll_usr_gpudev

#SBATCH --qos=gll_qos_gpudev

#SBATCH --gres=gpu:volta:1
```

Users with reserved resources

Users of projects that require reserved resources (such as industrial users or users associated to an agreement that involves dedicated resources) will be associated to a QOS.

Using the specific QOS (i.e. specifying the QOS in the submission script) , and specifying the partition gll_spc_prod, users associated to the allowed project will run their jobs on reserved nodes in the gll_spc_prod partition:

```
>#SBATCH --partition=gll_spc_prod

>#SBATCH --qos=<specific_qos>
```

Summary

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

SLURM partition	**QOS**	**# cores per job**	**max walltime**	**max running jobs per user/ max n. of cpus/nodes per user**	**max memory per node (MB)**	**priority**	**notes**
gll_all_serial (default partition)	*noQOS*	1	04:00:00	4 jobs / 4 cpus	3000 (per core)	40	
gll_all_transfer	*noQOS*	1	24:00:00	/	3000 (per core)	40	Temporary partition for accessing the filesystem related to the old version of GALILEO (UNTIL 29/02 /2020)
gll_usr_prod	*noQOS* *gll_qos_dbg* *gll_qos_bprod*	min = 1 max = 2304 (64 nodes) min = 1 max = 144 min = 65 nodes max = 128 nodes	24:00:00 02:00:00 24:00:00	20 jobs 144 cpus /4nodes 128 nodes per account 256 nodes in total per qos	118000 118000 118000	40 95 85	 higher priority than default qos --qos=gll_qos_dbg --qos=gll_qos_bprod
gll_usr_gpuprod	noQOS	min = 1 max = 144	08:00:00	4 nodes	118000	40	--gres=gpu:kepler:N (N=1,2)
gll_spc_prod	Every account needs to have a valid QOS to access this partition	Depending on the QOS used by the particular account	Depending on the QOS used by the particular account	/	118000	40	Partition dedicated to specific kind of users.
gll_meteo_prod							Partition reserved to meteo services, NOT opened to production

Programming environment

The programming environment of GALILEO 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 their codes, **profilers** to help with code optimization. In general, you must "load" also the correct environment 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, you will likely 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. If you prefer 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 GALILEO with the command:

```
> module available
```

checking the "compilers" section.

In general, the available compilers are:

- INTEL (ifort, icc, icpc) : module load intel
- PGI - Portland Group (pgf77,pgf90,pgf95,pghpf, pgcc, pgCC): module load pgi (profile/advanced)
- GNU (gcc, g77, g95): module load gnu

After loading the appropriate module, use the "man" command to get the complete list of the flags supported by the compiler, for example:

```
> module load intel
> man ifort
```

There are some flags that are common for all these compilers. Others are more specific. The most common are reported later for each compiler.

1. If you want to use a specific library or a particular include file, you have to give their paths, using the following options

```
-I/path_include_files        specify the path of the include files
-L/path_lib_files -l<xxx>    specify a library lib<xxx>.a in /path_lib_files
```

1. If you want to debug your code you have to turn off optimisation and turn on run time checkings: these flags are described in the following section.
2. If you want to compile your code for normal production you have to turn on optimization by choosing a higher optimization level

```
-O2 or -O3      Higher optimisation levels
```

Other flags are available for specific compilers and are reported later.

INTEL Compiler

Intel family compiler suite is recommended on GALILEO, 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. Initialize the environment with the module command:

```
> module load intel
```

The names of the Intel compilers are:

- **ifort**: Fortran77 and Fortran90 compiler
- **icc**: C compiler
- **icpc**: C++ compiler

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

```
> man ifort
> man icc
```

Some miscellaneous flags are described in the following:

```
-extend_source    Extend over the 77 column F77's limit
-free / -fixed    Free/Fixed form for Fortran
-ip               Enables interprocedural optimization for single-file compilation
-ipo              Enables interprocedural optimization between files - whole program optimisation
```

PORTLAND Group (PGI)

Initialize the environment with the module command:

```
> module load profile/advanced
> module load pgi
```

The name of the PGI compilers are:

- **pgf77**: Fortran77 compiler
- **pgf90**: Fortran90 compiler
- **pgf95**: Fortran95 compiler
- **pghpf**: High Performance Fortran compiler
- **pgcc**: C compiler

- **pgCC**: C++ compiler

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

```
> man pgf95
> man pgcc
```

Some miscellaneous flags are described in the following:

```
-Mextend            To extend over the 77 column F77's limit
-Mfree / -Mfixed    Free/Fixed form for Fortran
-fast               Chooses generally optimal flags for the target platform
-fastsse            Chooses generally optimal flags for a processor that supports SSE instructions
```

GNU compilers

The gnu compilers are always available but they are not the best optimizing compilers, especially for an Intel-based cluster like GALILEO. The default version is 4.8.5, you do not need to load the module for using it.

For a more recent version of the compiler, initialize the environment with the module command:

> module load gnu

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:

```
> man gfortan
> man gcc
```

Some miscellaneous flags are described in the following:

```
-ffixed-line-length-132       To extend over the 77 column F77's limit
-ffree-form / -ffixed-form    Free/Fixed form for Fortran
```

Debuggers and Profilers

If your code aborts at runtime, there may be a problem with it. In order to solve it, you can decide to analyze the core file (feature not available if the code is compiled with PGI) or to run your code using a debugger.

Compiler flags

In both cases, 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 keep in mind that 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 optimisation
-g      Produce debugging information
```

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

INTEL Fortran compiler

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

```
-traceback        generate extra information to provide source file traceback at run time
-fp-stack-check   generate extra code to ensure that the floating-point stack is in the expected state
-check bounds     enables checking for array subscript expressions
-fpe0             allows some control over floating-point exception handling at run-time
```

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.
```

In the following we report informations about some ways to debug your codes:

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, including:

- Single-thread, serial applications
- Multi-threaded applications
- Distributed MPI applications
- Any combination of the above

There are two forms of the command used to invoke pgdbg. The first is used when debugging non-MPI applications, the second form, using mpirun, is used when debugging MPI applications:

```
> pgdbg [options] ./myexec [args]
> mpirun [options] -dbg=pgdbg ./myexec [args]
```

More details in the on line documentation, using the "man pgdbg" command after loading the module.

To use this debugger, you should compile your code with one of the pgi compilers and the debugging command-line options described above, then you run your executable inside the "pgdbg" environment:

```
> module load profile/advanced
> module load pgi
> pgf90 -O0 -g -C -Ktrap=ovf,divz,inv  -o myexec myprog.f90
> pgdbg ./myexec
```

By default, pgdbg presents a graphical user interface (GUI). A command-line interface is also provided through the "-text" option.

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.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

More details in the online documentation, using the "man gdb" command.

To use this debugger, you should compile your code with one of the gnu compilers and the debugging command-line options described above, then you run your executable inside the "gdb" environment:

```
> module load gnu
> gfortran -O0 -g -Wall -fbounds-check -o myexec myprog.f90
> gdb ./myexec
```

Core file analysis

In order to understand what problem was affecting your code, you can also try a "Core file" analysis. Since core files are usually quite large, be sure to work in the /scratch area.

There are several steps to follow:

1. Increase the limit for possible core dumping

```
> ulimit -c unlimited (bash)
> limit coredumpsize unlimited (csh/tcsh)
```

1. If you are using Intel compilers, set to TRUE the **decfort_dump_flag** environment variable

```
> export decfort_dump_flag=TRUE  (bash)
> setenv decfort_dump_flag TRUE  (csh/tcsh)
```

1. Compile your code with the debug flags described above.

2. Run your code and create the core file.
3. Analyze the core file using different tools depending on the original compiler.

INTEL compilers

```
> module load intel
> ifort -O0 -g -traceback -fp-stack-check -check bounds -fpe0 -o myexec prog.f90
> ulimit -c unlimited
> export decfort_dump_flag=TRUE
> ./myexec
> ls -lrt
  -rwxr-xr-x 1 aer0 cineca-staff   9652 Apr  6 14:34 myexec
  -rw------- 1 aer0 cineca-staff 319488 Apr  6 14:35 core.25629
> idbc ./myexec core.25629
```

PGI compilers

```
> module load profile/advenced
> module load pgi
> pgf90 -O0 -g -C -Ktrap=ovf,divz,inv  -o myexec myprog.f90
> ulimit -c unlimited
> ./myexec
> ls -lrt
  -rwxr-xr-x 1 aer0 cineca-staff   9652 Apr  6 14:34 myexec
  -rw------- 1 aer0 cineca-staff 319488 Apr  6 14:35 core.25666
> pgdbg -text -core core.25666 ./myexec
```

GNU Compilers

```
> module load gnu
> gfortran -O0 -g -Wall -fbounds-check -o myexec prog.f90
> ulimit -c unlimited
> ./myexec
> ls -lrt
  -rwxr-xr-x 1 aer0 cineca-staff   9652 Apr  6 14:34 myexec
  -rw------- 1 aer0 cineca-staff 319488 Apr  6 14:35 core.25555
> gdb ./myexec core.2555
```

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.

To analyse a serial application:

1. Load Valgrind module --> module load valgrind
2. Load module for the compiler and compile your code with the compiler you prefer (Use -O0 -g flags)
3. Run the executable under Valgrind.

   If you normally run your program like this:

   ```
   myprog arg1 arg2
   ```

   Use this command line:

   ```
   valgrind  (valgrind-options) myprog arg1 arg2
   ```

   Memcheck is the default tool. You can add the --leak-ceck option that turns on the detailed memory leak detector. Your program will run much slower than normal, and use a lot more memory. Memcheck will issue messages about memory errors and leaks that it detects.

To analyse a parallel application:

1. Load Valgrind module --> module load valgrind
2. Load modules for compiler and openmpi libraries (at present only available for intel and gnu)
3. Compile your code with the "-O0 -g" flags both at compiling and linking time
4. Run the executable under Valgrind (both in interactive than in batch mode)

```
mpirun -np 4 valgrind (valgrind-options) myprog arg1 arg2
```

Totalview

Totalview is a parallel debugger with a practical GUI that assist users to debug their parallel code. It has functionalities like stopping and reprising a code mid-run, setting breakpoints, checking the value of variables anytime, browse between the different tasks and threads to see the different behaviours, memory check functions and so on. For information about how to run the debugger (by connecting the compute nodes to your display via RCM), type the command:

```
> module help totalview
```

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

Profilers (gprof)

In software engineering, **profiling** is the investigation of a program's behaviour 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.

Scientific libraries (MKL)

MKL

The Intel Math Kernel Library (Intel MKL) enables improving performance of scientific, engineering, and financial software that solves large computational problems. Intel MKL provides a set of linear algebra routines, fast Fourier transforms, as well as vectorized math and random number generation functions, all optimized for the latest Intel processors, including processors with multiple cores.

Intel MKL is thread-safe and extensively threaded using the OpenMP technology.

documentation can be found by loading the mkl module and searching in the directory:

```
${MKLROOT}/../Documentation/en_US/mkl
```

To use the MKL in your code you to load the module, then to define includes and libraries at compile and linking time:

```
> module load mkl
> icc -I$MKL_INC -L$MKL_LIB  -lmkl_intel_lp64 -lmkl_core -lmkl_sequential
```

For more informations please refer to the documentation.

Parallel programming

The parallel programming on Galileo is based on IntelMPI and OpenMPI versions of MPI. The libraries and special wrappers to compile and link the personal programs are contained in several modules, one for each supported suite of compilers.

These command names refer to wrappers around the actual compilers, they behave differently depending on the module you have loaded.

IntelMPI
IntelMPI in GALILEO is recommended since the architecture is based on Intel processors.
To load IntelMPI on GALILEO, check the module versions available with the "module avail" command, then load the relevant module. After you have loaded the Intelmpi module

```
> mpiifort -o myexec myprof.f90 (uses the ifort compiler)
```

For more option of the compiler, please see

```
> man mpiifort
```

The three main parallel-MPI commands for compilation with OpenMPI are:

- mpiifort (Fortran90/77)
- mpiicc (C)
- mpiicpc (C++)

OpenMPI
On GALILEO, "gnu" versions of OpenMPI are available. After you have load the openmpi module (check the available version with the "module avail" command):

```
> mpif90 -o myexec myprof.f90 (uses the gfortran compiler)
```

The four main parallel-MPI commands for compilation with OpenMPI are:

- mpif90 (Fortran90)
- mpif77 (Fortran77)
- mpicc (C)
- mpiCC (C++)

In all cases the parallel applications have to be executed with the recommended command:

```
> srun ./myexec
```

There are limitations to running parallel programs in the login shell. You should use the "Interactive SLURM" mode, as described in the "Interactive" section, previously on this page.