UG3.4.x: Old GALILEO UserGuide

(updated: Jan 2018)

Attention

This system is out of production since Nov 20, 2017

In this page:

---

hostname: login.galileo.cineca.it

early availability: 26/01/2015

start of production: 02/02/2015

end of production: 20 Nov 2017

```
Model: IBM NeXtScale
Architecture: Linux Infiniband Cluster
Nodes: 516
Processors: 8-cores Intel Haswell 2.40 GHz (2 per node)
Cores: 16 cores/node, 8256 cores in total
Accelerators: 2 Intel Phi 7120p per node on 344
nodes  (688 in total)
RAM: 128 GB/node, 8 GB/core
Internal Network: Infiniband with 4x QDR switches
Disk Space:2,500 TB of local storage
Peak Performance: 1.1 PFlop/s
```

The GALILEO supercomputer has been introduced in January 2015 and it is availaible to Italian public and industrial researchers. It is the national Tier-1 system for scientific research.

GALILEO has been ranked in the TOP500 list, as July2015, at position105 with a sustained performance of 684.3 TFlops.

This supercomputer is used to optimize and develop applications targeted at hybrid architectures, leveraging software applications in the fields of computational fluid dynamics, material and life science, and geophysics. The computing system is also available to European researchers as a Tier-1 system of the PRACE (www.prace-project.eu) infrastructure.

System Architecture

Compute Nodes: There are 516 IBM NX360 M5 16-core compute nodes. Each one contains 2 Intel(R) Xeon(R) Hashwell 8-core E5-2630 v3 processors, with a clock of 2.40GHz. All the compute nodes have 128 GB of memory. 384 compute nodes are equipped with 2 accelerators (Intel Xeon Phi 7120P) for a total of 768 accelerators in the whole system.

Login and Service nodes: 8 Login & Viz node NX360M5 are available, equipped with 2 nVidia K40 GPU each. 8 service nodes NX360M5 for I/O and cluster management.

All the nodes are interconnected through a Infiniband network, with OFED v1.5.3, capable of a maximum bandwidth of 40Gbit/s between each pair of nodes.

Accounting

For accounting information 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 of 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.

$DRES points to the shared repository where **D**ata **RES**ources are maintained. This is a data archive area **availble 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
```

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 the 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 and Tools".

Detailed information on how to use the accelerators on this system will be shortly availble in a separeted document.

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 has a 10 minutes time limit: for longer runs please use the "batch" mode.

A **parallel program** can be executed interactively only within an "Interactive" PBS batch job, using the "**-I**" (capital i) option: the job is queued and scheduled as any PBS batch job, but when executed, the standard input, output, and error streams are connected to the terminal session from which qsub was submitted.

For example, for starting an interactive session with the MPI program *myprogram*, using 2 processors, enter the commands::

```
> qsub -A <account_no> -I -l select=1:ncpus=2:mpiprocs=2  -- /bin/bash
    qsub: waiting for job ... to start
    qsub: job ... ready
> mpirun ./myprogram
> ^D
```

If you want to export variables to the interactive session, use the *-v* option. For example if *myprograms* is not compiled statically, you have to define and export the LD_LIBRARY_PATH variables:

```
> export LD_LIBRARY_PATH= ...
> qsub -I -v LD_LIBRARY_PATH ...
```

Batch

Batch jobs are managed by the **PBS** batch scheduler, that is described in section "Batch Scheduler PBS".

On GALILEO, it is possible to submit jobs of different types, using only one "routing" queue: just declare how many resources you need and your job will be directed into the right queue with a right priority.

The maximum number of nodes that you can request is 128 with a **maximum walltime of 24 hours**. If you do not specify the walltime, a default value of 30 minutes will be assumed. **The maximum amount of memory for each node is 120 GB** (to be specified as the value of the "mem" resource in the #PBS -l select directive).

For moving your data the "archive" queue is available with one cpu and a maximum walltime of 4 hours. In order to use this queue you have to specify the PBS flag "-q":

#PBS -q archive

You submit a script with the command:

```
> qsub script
```

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

```
> qstat -Q
```

At present, some general and preliminary queues are defined, for serving both the general scientific users and the industrial users on the system. A complete description will be included as soon as a stable situation will be reached.

For more information and examples of job scripts, see section "Batch Scheduler PBS".

Graphic session

If a graphic session is desired we recommend to use the RCM tool or the EnginFrame environment. Both of them are under construction, we will report here all required information asap.

Programming environment

The programming environment of the GALILEO machine 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.

For those interested to start programming in a mixed environment using Intel accelerators, we will soon publish a separated document in the docs /documents repository.

Compilers

You can check the complete list of available compiler on GALILEO with the command:

```
> module available
```

and 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
- 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 specifics. 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 optimisation by choosing a higher optimisation level

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

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

INTEL Compiler

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 miscellanous 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 miscellanous 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. 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:

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

Some miscellanous 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
```

Hybrid programming (Intel PHI)

Some of the nodes of the system are equipped with 2 accelerators per node. They can be addressed within C or Fortran programs by means of the Intel compilers suite.

You can find a **brief guide on the production and programming environment for the Intel Xeon Phi (MIC)** nodes in a separate document **here**.

Debuggers 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 **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 debuggin 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 usefull (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 usefull (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 usefull (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, Scalasca, TAU)

We plan to make available on GALILeO the three applications riported above in a short time. Detailed information will be published asap.

In the following we report information about other ways to debug your codes:

IDB (serial debugger for INTEL compilers)

The Intel Debugger (idb) is a source-level, symbolic debugger that lets you:

- Control the execution of individual source lines in a program.
- Set stops (breakpoints) at specific source lines or under various conditions.
- Change the value of variables in your program.
- Refer to program locations by their symbolic names
- Print the values of variables and set tracepoints
- Perform other functions (examining core files, examining the call stack, displaying registers)

The idb debugger has two modes: **dbx** (default mode) or **gdb** (optional mode) For complete information about idb, see the online Intel Debugger (IDB) Manual in:

```
$INTEL_HOME/Documentation/en_US/idb
```

To use this debugger, you should specify the ifort command and the debugging command-line options described above, then you run your executable inside the "idb" environment:

```
> module load intel
> ifort -O0 -g -traceback -fp-stack-check -check bounds -fpe0 -o myexec myprog.f90
> idb ./myexec
```

On PLX the debugger runs in GUI mode by default. You can also start the debugger in command line mode on these systems by specifying **idbc** instead of idb in the command line.

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 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 though 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 on line 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:

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

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

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

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

Core file analysis

In order to understand what problem was affecting you 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 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

```
> 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
```

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 possilbe 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 in:

```
${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 inormation 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.

The main four parallel-MPI commands for compilation are:

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

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

IntelMPI
To load IntelMPI on Galileo check the names with the "module avail" command, then load the relevant module:

```
> module avail intelmpi
intelmpi/5.0.2—binary

> module load intel intelmpi/5.0.2--binary
> man mpif90
> mpif90 -o myexec myprof.f90 (uses the ifort compiler)
```

OpenMPI
At present on Galileo "intel", "gnu" and “pgi” versions of OpenMPI are available. To load one of them, check the names with the "module avail" command, then load the relevant module:

```
> module load profile/advanced
> module avail openmpi
openmpi/1.8.4--gnu--4.9.2
openmpi/1.8.4--intel--cs-xe-2015--binary
openmpi/1.8.5--pgi--15.3

> module load gnu openmpi/1.8.4--gnu--4.9.2
> man mpif90
> mpif90 -o myexec myprof.f90 (uses the gfortran compiler)

> module purge
> module load intel openmpi/1.8.4--intel--cs-xe-2015--binary
> man mpif90
> mpif90 -o myexec myprof.f90 (uses the ifort compiler)

> module purge
> module load pgi openmpi/1.8.5--pgi--15.3
> man mpif90
> mpif90 -o myexec myprof.f90 (uses the pgf90 compiler)
```

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

```
> mpirun ./myexec
```

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

Outgoing links: Unknown macro: 'outgoing-links'