

# UG3.4.x: PICO UserGuide

(updated: Jan 2018 )



## Attention

This system is out of production since Jan 24, 2018

In this page:

- [System Architecture](#)
- [Disks and Filesystems](#)
- [Production Environment](#)
- [Programming environment](#)
  - [Debuggers](#)
    - [Compiler flags](#)
  - [Profilers \(gprof\)](#)
  - [Scientific libraries \(MKL\)](#)
  - [Parallel programming](#)

**hostname:** login.pico.cineca.it

**Production:**

**End of production: Jan 24, 2018**

```
Model: IBM NeXtScale server
Architecture: Linux Infiniband Cluster
Processors Type: Intel Xeon (Ten-Core) E5-2670v2
2.50 GHz (Compute)
Number of nodes: 54 Compute + 4 visualization + 2
Login + 14 other
Number of cores: 1080 (compute)
Number of accelerators: 4 + 2 (only on viz nodes)
RAM: 128 GB/Compute node (2 viz nodes with 512GB)
```

## System Architecture

PICO is a Intel Cluster made of 74 nodes of different types, devoted to different purposes, with the common task of data analytics and visualization on large amount of data.

**Login nodes:** 2 x 20-core nodes, 128 GB-mem. Both of this two nodes are reachable with the IP: login.pico.cineca.it

**Compute/Cloud Nodes:** 64 x 20-core nodes, 128 GB-mem. Part of these nodes contains a standard scientific computational environment. Pre-installed applications are in the visualization domain, as well as data analysis, post-processing and bioinformatics. You can access this environment via ssh and submit your large analysis in a PBS batch environment. Another part is used for internal activities in the domain of Cloud computing, large Scientific Databases, hadoop for science. Special projects can be activated on this partition, selected by special calls.

**Big memory Nodes:** one node, big1, equipped with 32 cores - 0.5 T is available for specific activities, which require a remarkable quantity of memory. It is a HP DL980 server.

- The big1 node is equipped with 8 [Quad-Core ARK Intel\(R\) Xeon\(R\) E7520](#) processors with a clock of 1.87GHz and 512 GB of RAM, and a [NVidia Quadro 6000](#) graphics card.
- **(REMOVED on June 2016)** The big2 node is equipped with 4 Ten-Core Intel(R) Xeon(R) E7-2860 with a clock of 2.26GHz and 1024 GB of RAM.

*This node can be accessed upon request (to [superc@cinca.it](mailto:superc@cinca.it)) and is served by the "big1" queue of PBS.*

**Viz nodes:** 2 x (20-core, 128 GB-mem, 2 x GPU Nvidia K40) + 2 x (20-core, 512 GB-mem, 1 x GPU Nvidia K20). A remote visualization environment is defined on this partition, taking advantage from the large memory and the GPU acceleration. The 512 GB nodes can be accessed using the "bigmem" queue of PBS.

**BigInsights nodes:** 4 x 16-core nodes, 64 GB-mem/node, 32TB local disks/node + 1 x 20-core nodes, 128 GB-mem.). On these nodes an IBM solution for Hadoop applications is available. [InfoSphere BigInsights](#) is available for special project to be agreed upon with our staff.

The compute nodes contain 2 Intel(R) Xeon(R) ten-core E5-2670 v2 processors, with a clock rate of about 2.5 GHz. The PICO cores are capable of 8 floating point operations per cycle (hyper-threading disabled). All the nodes are interconnected through a custom Infiniband network, allowing for a low latency/high bandwidth interconnection.

## Disks and Filesystems

PICO is completely consistent with the general CINECA infrastructure (see Sect "[Data Storage and File Systems](#)").

	Tot Dimension (GB)	Quota (GB)
<b>\$HOME</b>	50 TB	50 GB
<b>\$CINECA_SCRATCH</b>	300 TB	no quota
<b>\$WORK(*)</b>	1.500 TB	1 TB (or as required by the project)

Only the \$CINECA\_SCRATCH and \$WORK areas should be used for running programs.

Since these filesystems are all based on GPFS (General Parallel File System), the usual unix command "quota" will not work. Instead use the local command "cindata" to query for disk usage and quota ("cindata -h" for help):

```
cindata
```

In addition to these filesystems, a long term archive area on tape, **\$TAPE**, is available upon request. Finally, a new storage resource, **\$DRES**, was added on the 1st October 2014. It is intended as a medium/long term repository and as a shared area within the project tema and across HPC platforms. Also this resource is created upon request, please, refer to the [relevant page](#) in the General Information section.

(\*) the \$WORK space has not been defined on PICO system yet.

## Production Environment

PICO is a special purpose system for data analytics and processing. It is normally used by several users at the same time, so production jobs must be submitted using a *queuing system*. This guarantees that the access to our resources is as fair as possible.

There are two different modes to use this system: **Interactive** and **Batch**. In order to request the computational resources, either in interactive or in batch mode, you need to have a "budget" of hours to spend, organized in projects (called *account\_numbers*). **Please, refer to the [Billing Policy](#) features in the [Accounting Section](#).** The command "saldo -b" displays the list of projects you have at your disposal, together with their validity period, the initial and the consumed amounts of hours:

```
> saldo -b
```

### Interactive

A **serial program** can be executed on Pico login node in the standard UNIX way:

```
> ./program
```

This is allowed only for very short runs, since the interactive environment on the login node 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. You can then specify the resources you need (number of nodes, number of processors per node, memory etc.) by using the flag -l (lower-case l of lima).

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 -q parallel -- /bin/bash
qsub: waiting for job 389.node001 to start
qsub: job 389.node001 ready
> mpirun ./myprogram
```

The default memory assigned to a job, if not explicitly requested, is **7GB per node**. Please, ensure to request the memory you need for your processes to run via the -l flag of qsub:

```
> qsub -A <account_no> -I -l select=4:ncpus=20:mem=20GB -q parallel
```

The default walltime limit defined on the available queues (see next section for their description) is 6 hours. You can define the walltime you need via the `-l` flag:

```
> qsub -A <account_no> -I -l select=4:ncpus=16:mem=14GB,walltime=04:00:00 -q parallel
```

If you want to export variables to the interactive session, use the `-v` option. You can export *all* the variables defined in your environment on the login node with 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](#)". Changes could occur to better meet the user requests.

You can submit a script with the command:

```
> qsub script
```

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

```
> qstat -Q
```

At present, on PICO it is possible to submit jobs on the following queues: parallel, serial, bigmem and big1

Queue	Max nodes	Max CPUs/	max wall time
parallel		400	100:00:00
serial		1	4:00:00
bigmem		40 (+2 gpu)	100:00:00
big1		36	100:00:00 (only on request)

For more information and examples of job scripts, see the section for "[Batch Scheduler PBS](#)".

## Programming environment

The programming environment 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 compiler 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 (pgfortran, pgHPF, pgCC): module load pgf
- GNU (gfortran, gcc, g++): 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
```

2. 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.

3. 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
```

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 miscellaneous flags are described in the following:

```
-extend_source      Extend over the 77 column F77's limit  
-free / -fixed      Free/Fixed form for Fortran  
-openmp             Enables the parallelizer to generate multi-threaded code based on OpenMP directives
```

## PORTLAND Group (PGI)

Initialize the environment with the module command:

```
> 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  
-mp               Enables the parallelizer to generate multi-threaded code based on OpenMP directives
```

## GNU compilers

The name of the GNU compilers are:

- **gfortran**: Fortran compiler
- **gcc**: C compiler
- **g++**: C++ compiler

The documentation can be obtained with the **man** command:

```
> man gfortran  
> man gcc
```

Please notice that "man gfortran" only provides additional flags working for the Fortran compiler with respect to gcc. Refer to the gcc manual for the common flags. 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
-fopenmp	Enables the parallelizer to generate multi-threaded code based on OpenMP directives

## Debuggers

If your code dies at runtime, you can try to understand and solve the problem by analyzing the core files (core not available with PGI compilers) or running 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 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
-traceback      Generate extra information to provide source file traceback at run time
-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
-ffpe-trap=zero,overflow,invalid,underflow Specify the list of IEEE exceptions when
                                                a Floating Point Exception should be raised
```

## 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
```

## 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
```

With the Intel Compilers you can simply link the mkl library via the flag `-mkl=sequential/parallel`, for example:

```
> ifort -mkl=sequential mycode.f90
> icc -mkl=parallel mycode.c
```

For more information please refer to the documentation.

## Parallel programming

The parallel programming is based on the IntelMPI version of MPI or on OpenMPI (for GNU compilers). The library and special wrappers to compile and link the personal programs are contained in the "intelmpi" (or "openmpi") modules.

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. On PICO, in addition to the IntelMPI package, only the "gnu" version of OpenMPI is available at present. To load one of them, check the names with the "module avail" command then load the relevant module:

```
> module avail openmpi
openmpi/1.8.2--gnu--4.8.3


> module load gnu openmpi/1.8.2--gnu--4.8.3
> man mpif90
> mpif90 -o myexec myprof.f90 (uses the gfortran 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'