

UG3.4.x: PLX UserGuide



ATTENTION: PLX is out of production by Jan 21, 2015

(updated:)

In this page:

- [System Architecture](#)
- [Disks and Filesystems](#)
- [Production environment](#)
 - [Interactive](#)
 - [Batch](#)
 - [Graphic session](#)
- [Programming environment](#)
 - [Compilers](#)
 - [GPU programming](#)
 - [Debuggers](#)
 - [Compiler flags](#)
 - [Debuggers available on PLX \(ddt, idb, pgdbg, gdb, valgrind\)](#)
 - [Core file analysis](#)
 - [Profilers \(gprof\)](#)
 - [Scientific libraries \(MKL\)](#)
 - [Parallel programming](#)

hostname: login.plx.cineca.it

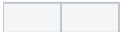
First PLX system:

- **early availability:** 23/11/2009
- **start of production:** 02/12/2009

Second PLX system (GPU enabled):

- **early availability:** 02/05/2011
- **start of production:** 01/06/2011

(a separate document about GPU usage is available in the [Other documents](#) repository)



The fastest Italian cluster based on GPU for industrial and public research

The PLX-GPU supercomputer has been introduced in June 2011 by SCS S.r.l - SuperComputing Solutions and CINECA, and it is available to Italian industrial and public researchers. At the time of the release, It was ranked at the 54th position in the TOP500 ([July 2011](#)) and 5th position in the Green500 ([June 2011](#)), which list, respectively, the most powerful and the most energy-efficient supercomputers in the world.

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

Model: IBM iDataPlex DX360M3
Architecture: Linux Infiniband Cluster
Processors Type:

- Intel Xeon (Esa-Core Westmere) E5645 2.4 GHz (Compute)
- Intel Xeon (Quad-Core Nehalem) E5530 2.66 GHz (Service and Login)

Number of nodes: 274 Compute + 1 Login + 1 Service + 8 Fat + 6 RVN + 2 Big mem +
+ 8 Storage + 2 Management
Number of cores: 3288 (Compute)
Number of GPUs: 528 nVIDIA Tesla M2070 + 20 nVIDIA Tesla M2070Q
RAM: 14 TB (48 GB/Compute node + 128GB/Fat node Registered ECC Double Data Rate 3-DDR3 1333MHz)

PLX is an IBM iDataPlex DX360M3 Cluster with RedHat Enterprise Linux 5.6, made of 300 nodes of different types:

Compute Nodes: There are 274 IBM X360M2 12-way compute nodes. Each one contains 2 [Intel\(R\) Xeon\(R\) Westmere six-core E5645 processors](#), with a clock of 2.40GHz. All the compute nodes have 48GB of memory, with an allocatable memory on the node of 47 GB (see [PBS resources:memory allocation](#)). All the PLX cores are capable of 6 floating point operations per cycle (hyper-threading disabled). All compute nodes have two [nVIDIA M2070 GPU cards](#), among these 10 nodes are equipped with the "[Quadro" M2070Q](#) card model.

Login and Service nodes: 1 Login and 1 service (back-up login) [Intel\(R\) Xeon\(R\) Nehalem quad-core E5530 processor](#) at 2.66GHz with 24GB of memory.

Fat Nodes: 8 IBM X3650M2 nodes containing 2 Nehalem quad-core processors, with a clock of 2.66GHz and 128GB of memory. *The fat nodes are currently reserved and are not available.*

RVN Nodes : 6 RVN nodes for pre and post-processing activities, supporting DCV-RVN remote visualization software of IBM. 4 RNV nodes are equipped with IBM X5570 Intel(R) Xeon(R) CPU at 2.93GHz, and 2 with IBM E5540 Intel(R) Xeon(R) CPU at 2.53GHz. *Only two of these RVN nodes can be accessed upon request.*

Big memory Nodes: two nodes, big1 and big2, equipped respectively with 0.5 T and 1 T of RAM are available for specific activities, which require a remarkable quantity of memory. Both are HP DL980 servers. 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. 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. *These nodes can be accessed upon request.*

Storage Nodes: 8 IBM X3550M2 dual socket Intel(R) Xeon(R) E5530 at 2.40GHz and 24 GB of memory dedicated to I/O activity.

Management Nodes: 2 IBM X3550M2 dual socket Intel(R) Xeon(R) E5530 at 2.40GHz and 24 GB of memory.

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.

Disks and Filesystems

PLX 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**: large disk for storing run time data and files
- a data archive area **\$CINECA_DATA**: shared with other HPC systems.

\$CINECA_DATA is not mounted on the compute nodes. This means that you **can't access \$CINECA_DATA within a batch job**: all data needed during the batch execution has to be moved on \$CINECA_SCRATCH before the run starts.

Since all the filesystems, specifically those with a quota defined on them (\$HOME and \$CINECA_DATA) 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 PLX 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 our 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 GPUs on this system are available in a separated document (in the [HPC systems related documents](#) repository)

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 -q debug -- /bin/bash
qsub: waiting for job 4611.node351.plx.cineca.it to start
qsub: job 4611.node351.plx.cineca.it 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 PLX it is possible to submit jobs on different queues, each identified by different resource allocation. You submit a script with the command:

```
> qsub script
```

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

```
> qstat -Q
```

At present the following queues are defined for general scientific users (please note that the parallel queue is a routing queue: parallel jobs will be served by the longpar queue together with the longpar jobs):

job type	Max nodes	Max CPUs/GPUs	max wall time
debug	2	24/ 4	0:30:00
parallel	22	264/44	24:00:00
longpar	22	264/44	24:00:00
archive	1	1	04:00:00
big1	1	32/1	72:00:00
big2	1	40/0	72:00:00

The **archive** queue is not meant for computational jobs; it is a service queue for accessing data on \$CINECA_DATA and archiving data on the cartridge medium (see ["Batch Scheduler PBS"](#) - archive your data: cart).

The big1 and big2 queues are reserved to use the big memory nodes.

Moreover, several *"private"* queues are available. They are dedicated to industrial users of the machine.

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

Graphic session

If a graphic session is desired we recommend to establish a VNC client/server connection with PLX. To do so, the following procedure has to be followed:

- 1 launch the VNC server on PLX
- 2 establish a ssh tunnelling from your local workstation (i.e. "client" side)
- 3 launch a VNC viewer on your local workstation (i.e. VNC client)

To launch the VNC server on the PLX machine, you can use a specific script:

```
> module load tightvnc
> vncserver_wrapper
```

please, make sure you carefully read what is printed to screen by this script! The following steps depend on the nature of your local workstation. In case it is Linux, it should be enough to carefully follow the instruction printed on screen by the vncserver_wrapper after its execution. In case you are working on a Windows workstation, please refer to the [relevant document](#) in the *"Other documents"* folder of this portal.

Programming environment

The programming environment of the IBM-PLX 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.

If you want to start programming in a mixed environment using GPUs, please refer to a [separated document](#) in the docs/documents repository.

Compilers

You can check the complete list of available compiler on PLX 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,pgl, 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 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 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 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 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. 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 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
```

GPU programming

The new PLX system is equipped with 2 GPUs per node. They can be addressed within C or Fortran programs by means of directives for a "pre-processor" made available by the CUDA toolkit. Detailed information about GPU programming are contained in a specific document (see [GPGPU \(General Purpose Graphics Processing Unit\)](#))

Debuggers

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 on PLX (ddt, idb, pgdbg, gdb, valgrind)

DDT (Distributed Debugging Tool - Allinea)

DDT is the most advanced debugging tool available for scalar, multi-threaded and large-scale parallel applications. C, C++, Fortran and Fortran 90/95 /2003 are all supported by DDT, along with a large number of platforms, compilers and all known MPI libraries. You may also debug GPU programs by enabling CUDA support.

A short tutorial is available on the hpc portal: [DDT \(Distributed Debugging Tool - Allinea\) tutorial](#)

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

```
> valgrind ./test (or, more detailed, ...)
> valgrind --leak-check=full --show-reachable=yes --leak-resolution=high ./test
```

To analyse a parallel application, you need a script (mpirun-dbg) that is contained in the "cineca-tools" module :

1. Load Valgrind and cineca-tools modules

```
> module load valgrind
> module load cineca-tools
```

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 using an Interactive qsub shell

```
> qsub -q debug -l select=1:mpiprocs=4 -I -v LD_LIBRARY_PATH mpirun-dbg -openmpi-ib -tool valg -valgrind-args .
/prova
```

The same flags used for the serial code must be added after the arguments "-tool valg -valgrind-args".

By default the output is written on a number of files equal to the number of requested processors.

A typical output name file is **Valgrind.report.123445** where 123445 is the PID of the pbs job. For that use the flag:

```
--log-file=Valgrind.report
```

To get more details for the parallel execution of valgrind:

```
> module load cineca-tools
> mprun-dbg --help
```

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 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 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 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 information please refer to the documentation.

Parallel programming

The parallel programming is based on the OpenMPI version of MPI. The library and special wrappers to compile and link the personal programs are contained in several "openmpi" 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. At present on PLX only the "intel" and "gnu" versions of OpenMPI are available. To load one of them, check the names with the "module avail" command then load the relevant module:

```
> module avail openmpi
  openmpi/1.3.3--gnu--4.1.2          openmpi/1.3.3--intel--11.1--binary
> module load gnu openmpi/1.3.3--gnu--4.1.2
> man mpif90
> mpif90 -o myexec myprof.f90 (uses the gfortran compiler)

> module purge
> module load intel openmpi/1.3.3--intel--11.1--binary
> man mpif90
> mpif90 -o myexec myprof.f90 (uses the ifort 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.

-
- [DDT \(Distributed Debugging Tool - Allinea\) tutorial](#)
-

Outgoing links:

 Unknown macro: 'outgoing-links'