High Performance Systems User Guide

Supercomputing Group

High Performance Systems Department

CINECA

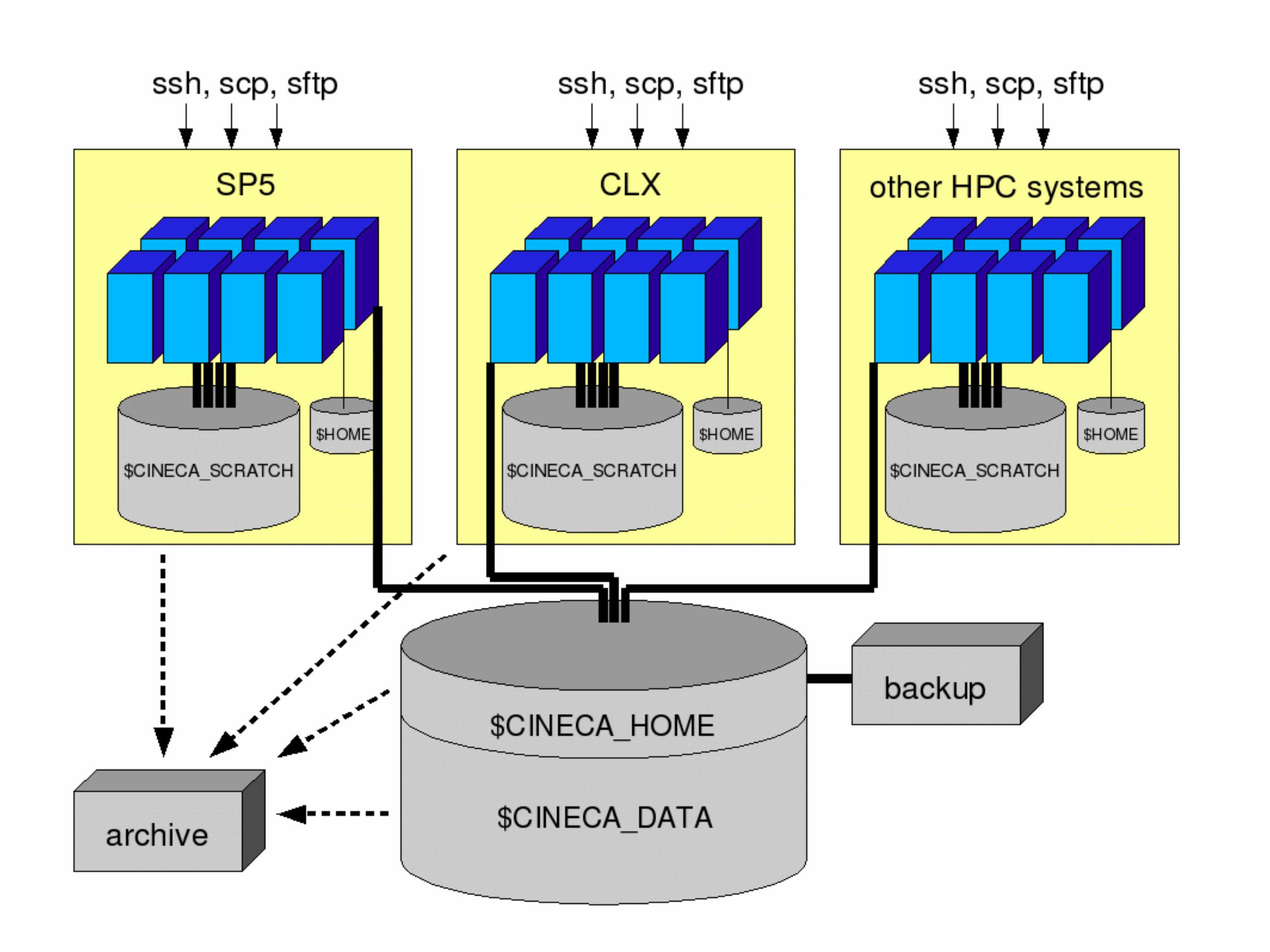
superc@cineca.it

Edited by:

Gerardo Ballabio
Sigismondo Boschi
Cristiano Calonaci
Carlo Cavazzoni
Andrew Emerson
Claudio Gheller
Roberto Gori
Andrea Tarsi

Table of Contents

General guide



Introduction

CINECA is moving in the direction of giving a common infrastructure to all its High Performance Systems and to their access, in order to allow an easy use of our resources and an easy exchangeability among them. The idea is that once a user has access to our systems, he/she will be able to access any other expecting similar behavior and having access to shared resources – despite the different applicative characterizations of the various systems. The basic ingredients for the Infrastructure are then:

– shared resources;

– uniform behavior.

For now the basic shared resources are the shared filesystems with their backup and the archive facility, but we are working on a larger set of shared services, that are independent of the present and future High Performance engines that will enter in the Infrastructure. In the next sections we will present the shared resources and the uniform behavior exhibited by our Systems, as drawn in the image above.

Data storage disk areas

All CINECA High Performance Systems share a uniform set-up in terms of accessible disk areas. In particular every system allows access to:

- **$CINECA_SCRATCH:** temporary work area, local to the system where applications can benefit from the high bandwidth of a parallel filesystem. Data left in $CINECA_SCRATCH and not accessed for more than 20 days will be deleted without notice.
  $CINECA_SCRATCH behaves very well when I/O is performed accessing large blocks of data. Avoid performing frequent and small I/O on this area, because it is not well suited for it. Parallel systems have a lot of distributed memory: use it.
- **$CINECA_HOME**: permanent storage area shared among all our High Performance Systems. This area is subjected to daily incremental backups, so it doesn't matter if data is accidentally deleted and we will keep records of old versions of your data/codes stored here.
- **$CINECA_DATA**: permanent storage area shared among all our High Performance Systems. This area is not backed up, but is potentially much larger than $CINECA_HOME and data will be never deleted automatically. However data are preserved from disk failures with some level of data replication: data will be still accessible even in case of failure of some of the physical disks.
- **$HOME**: **IMPORTANT: Don't use this area for your data**. It is where the login process leaves you and is very small. It is where system and user applications store their dot-files and dot-directories (.nwchemrc, .ssh,...) and where users keeps initialization files local to the systems (.cshrc, .profile,...).

$CINECA_SCRATCH is a shared resource among all the users of a given system; $CINECA_HOME and $CINECA_DATA are segmented in many different filesystems: one for any user group.

$HOME has a very small user quota, $CINECA_SCRATCH, $CINECA_HOME and $CINECA_DATA do not.

The environment variables $HOME, $CINECA_HOME and $CINECA_DATA are defined on all our High Performance Systems, and you can have access to these areas simply with something like “cd $CINECA_HOME”. You are strongly encouraged to use these environment variables instead of full paths to refer to data in your scripts and codes.

Information and Support

If you are a user of the High Performance Systems at CINECA, it is very important to receive all the news, information, scheduled downs, software updates, any problems and so on, about our computing resources. We send any information about our Infrastructure to the HPC-NEWS mailing list.

You can subscribe to HPC-NEWS sending an email, from the email address you want to subscribe, to **listserv@list.cineca.it** with the body (the subject does not matter):

```
subscribe hpc-news
```

You can consult the old messages posted to HPC-NEWS in the archive, from the Listserv web site:

    http://list.cineca.it/archives/hpc-news.html

If you are not using our services any more and want to leave the HPC-NEWS list, send an email, from the email address you want to unsubscribe, to **listserv@list.cineca.it** with the body:

```
unsubscribe hpc-news
```

If you can not find the needed information in the User Guide and you need support, contact us at the email address: **superc@cineca.it**

Please, use this email address and not our personal email addresses for support requests, in order to get a response as soon as possible.

Access to the Systems

All the High Performance Systems can be accessed with:

- **SSH** to have access to the system prompt;
- **SCP, SFTP** to transfer data from/to the systems;
- **FTP** occasionally, if scp/sftp do not allow reasonable bandwidth to the specified system. FTP can be enabled on request, if needed, on any system.

All the systems share the same password and if you change it on one system, it will changed on all of them (the propagation to all the systems can take up to one hour).

Use the “passwd” command at the system prompt of any system to change the general password.

On our systems you will login with one of the two shells: bash or tcsh. Contact us if you need to change your default login shell.

Accounting: repi

All our users have a budget in terms of CPU hours on any High Performance System they have access to. All the systems have a budget, independent of each other.

Every budget has a beginning date and an ending date, within which the budget must be used.

All the production High Performance Systems of CINECA keep track of the CPU time (user time) used by the jobs of the users, and the used budget is updated once a day, at midnight, detracting what has been used in the past 24 hours by the user applications.

Users have access to their accounting record using the “repi” command, available at the command prompt of the related computing system. In the following table, you can find the basic options of “repi”:

Command	***Reported information***
repi -ch	the total budget, the used budget, the start date and the end date
repi -h	the used budget, day by day

Archive your DATA: cart

On our systems you can archive large amount of data off-line by using "cart" procedures. Before using them, you may need to contact us in order to be enabled. You can define as many volumes as you want (imagine them like virtual CDs) and then store data on them. When you do not need them anymore, you can delete single files or a whole virtual volume. Find in the following table the cart commands:

Command	***Action***
`cart_new` *`vol_name`*	create a new volume named *vol_name* (choose whatever you want)
`cart_dir`	show all the defined volumes from the user
`cart_dir` *`vol_name`*	show the files stored on the volume named *vol_name*

Command	***Action***
`cart_put` *`vol_name file`*	save the file named "file" in the volume *vol_name*
`cart_get` *`vol_name file`*	retrive the file named "file" from the volume *vol_name*
`cart_del` *`vol_name file`*	delete the file named "file" from the volume *vol_name*
`cart_del` *`vol_name`*	delete the virtual volume named *vol_name* (must be emptied before)

All the cart commands accept the "-?" flag for a short help.

ATTENTION: it is much more efficient to store a few large files than many small ones: use “tar” to aggregate many small files if needed.

Running applications

Since there are many users in competition for the usage of our High Performance Systems it is mandatory for the users to submit their production runs using queuing systems. This guarantees that the access to our resources is as fair as possible.

Briefly, there are 3 different ways to use our High Performance Systems:

- **interactive**, for minimal testing, interactive operations, data movement, archiving, code development, compilations, basic debugger usage. The CPU time used by a task is limited to usually 10 minutes.
- **debug**, for debugging purposes and to test batch scripts before they are put in production. Usually you can have access almost immediately up to 4 CPUs for 10 minutes.
- **batch**, for the production runs. Users must prepare a shell script containing all the operations that will be executed in batch, once the requested resources have been freed for it.

In batch, users will have access to the temporary disk area **$TMPDIR**, that will be created by the queuing systems just before the start of the execution of the batch script, and deleted just after its end. If you need to access to work data after the end of the job, we strongly suggest to create the directory $CINECA_SCRATCH/$USER and to work there.

The common queuing system that we have chosen for all our computing platforms in our

Infrastructure is LSF (Platform, http://www.platform.com). Only SP5 will stay with its native queuing system, LoadLeveler. See the SP5 documentation for its usage.

Running applications using LSF

Interactive

From the shell session you open on the login node you can not use more than 10 minutes of cpu time per process. This session is useful to edit, compile and other lightweight operations.

The execution of any interactive program, other than compilers, editors and other Unix commands, must be done using LSF (see below), even the interactive runs. To avoid to overload the login node, also serial runs must be executed reserving free resources with LSF:

```
bsub -Is -W 10 program -program-args
```

With the "-Is" flag you specify that this will be an interactive session: input and output will be connected with the current shell session.

With the flag "-W 10" you specify that your job will run at most for 10 minutes (of elapsed real time). Interactive jobs can not take longer than 10 minutes.

To run a small parallel test:

```
bsub -Is -n 2 -W 10 mpirun ./test -test-args
```

"-n 2" means that you need 2 CPUs. If there are many users using debug resources, it can happen that the 2 CPUs are not immediately available. The command will wait some minutes, until the required resources become available.

To obtain a batch-interactive shell on one of the compute nodes use a command like:

```
bsub -Is -n 4 -W 10 tcsh
```

From the shell just created, you can use mpirun to run parallel tests, debug code, edit files... Issue "exit" (or type "CTRL-D") to release the resources granted by LSF before the end of the 10 minutes.

Attention: You can not at all run parallel programs with mpirun directly from the login node.

Debug

On all the platforms a minimal amount of resources is kept free in order to allow to any user to debug its codes and to test the scripts before they are put in production. You can ask at most for 4 CPUs and 10 minutes of time (-n 4 -W 10). Run larger then this are production runs.

Batch

The main commands you should learn, to use this batch system, are:

`bqueues -l`	Display queue information

`bsub < jobscript`
`bsub command`
	Submits the commands contained in "jobscript" or run "command" within LSF.
	ATTENTION: the input jobscript MUST be redirected with "<".
	In both cases the initial working directory of the batch job will be same of where "bsub" was invoked.

`bjobs -a`	Shows submitted jobs
`bkill jobid`	Cancels the job jobid from the queuing system
`bhist jobid`	Display historical information about jobid

The two main resources for user requests are:

- wall clock time: `-W <hh:mm>` (hours:minutes) or `-W <mm>` (minutes only)
- number of processors: `-n <nn>`

You can specify the desired resources both on the command line and inside the script of bsub, with other options like:

-o file.out: appends the output to "file.out"

-e file.err: appends the error to "file.err"

-oo file.out -eo file.err: the same, but overwrites the files.

-u mail_address: send email to the specified address when the job finishes.

On command line:

```
bsub -n 2 -W 6:00 -o job.out -e job.err < jobscript
```

Inside the script:

any option specified ad the beginning of the jobscript, after a comment line beginning with "#BSUB" will be interpreted by bsub, as the options where specified on the command line. However options specified on the command line, take the precedence over the ones in the job-script. For example:

```
#!/bin/tcsh
#BSUB -n 2 -W 2:00
#BSUB -oo %J.out -eo %J.err
... your batch commands ...
```

The first line specify the shell used to interpret the script (sh if not specified), in the following line you specify the other options. %J is replaced by LSF with the JOB ID.

If the job is successfully submitted, `bsub` displays the job ID and the queue to which the job has been submitted.

For more information, view:

```
    man bsub
```

Examples

Suppose you want to run the parallel program program.x on 16 processor for 1 hour. If you want to specify the requests on the bsub command line, you can submit it with the command:

```
    bsub -n 16 -W 1:00 -oo %J.out -eo %J.err mpirun ./program.x
```

If you prefer to include the requests in a job_script, then the file “job.lsf” will contain:

```
    #BSUB -n 16
    #BSUB -W 1:00
    #BSUB -oo %J.out
    #BSUB -eo %J.err
    mpirun ./program.x
```

and you can submit it to the queuing system with the command:

```
    bsub < job.lsf
```

LSF and modules

Differently from other queuing systems, LSF inherits the environment from the shell session from which the you have submitted the job. However it is better to specify all the needed modules, not loaded by default, at the beginning of the script. If there are possibile conflicts (like if you need to use the PGI compiler on CLX, while the standard one is INTEL), start the script with:

```
    module purge
```

then load all the needed modules.

Using installed applications: module

A basic default environment is already set up by mean of system login configuration files, but not the applications environment.

Applications need to be initialized, within the scope of the current shell session, by means of the **module** command.

With the modules approach, users simply "load" and "unload" modules to control the environment needed by applications.

The following table contains the basic sub-commands of “module”:

Command	***Action***
`module avail`	show the available application modules on the machine
`module list`	show the modules currently loaded on that shell session
`module load` ***`application`***	load the module named *application* in the current shell session, preparing the environment for the related application.
`module help application`	show specific information and basic help on the application
`module purge`	unload all the loaded modules

ATTENTION: Remember to load the needed modules in batch scripts too, before using the related applications.

High Performance Tools

On all High Performance Systems of CINECA you will find high performance tools for the effective deployment of the computing resources. In particular, on all the systems you will have access to:

- Fortran77, Fortran90, C, C++ compilers;
- the MPI library, standard for the development of parallel computing codes;
- symbolic parallel debugger;
- an optimized version of the BLAS library (in particular DGEMM – linear algebra matrix-matrix multiply – can benefit a lot with respect to a straightforward Fortran implementation).

Look at the system specific guides to find instructions on the use of such and other optimized tools available on each system.

System Specific Guides

SP5 User Guide

hostname: sp.sp5.cineca.it

early availability: July 1st, 2005

start of production: July 22nd, 2005

System architecture

The SP5 is an “IBM SP Cluster 1600”, made of 64 nodes p5-575 interconnected with a pair of connections to the Federation HPS (High Performance Switch).

Globally the machine has 512 IBM Power5 processors, capable of 4 double precision floating point operations per clock cycle, and 1.2 TBs of memory. The peak performance of SP5 is 3.89 TFlops.

A p5-575 node contains 8 SMP processors Power5 at 1.9GHz. 60 nodes have 16GBs of memory each, 4 nodes have 64GBs each.

The HPS interconnect is capable of a bandwidth of up to 2GB/s unidirectional.

More information about the details of the system at the IBM site:

http://www.ibm.com/servers/eserver/pseries/library/sp_books/

Main differences upgrading from SP4 to SP5

Performance

The Power5 processor has higher clock speed with respect to Power4 and – especially – larger caches, more registers, higher memory bandwidth (the bottleneck for most of the scientific applications), lower memory latency. We expect a gain of about +80% for many applications.

Performance reliability

SP4 was made of 32-way nodes, each made of 4 MCM (Multi Chip Module), giving a system where memory accesses were not very uniform within the node. On SP5 there are only 2 MCM per node, more deeply connected, and on the Power5 chip only one of the 2 CPU cores is active. In this way the memory accesses should be more uniform. Furthermore there is not any need (nor possibility) to partition the nodes, except for system requirements, that was another source of variation in memory latency from one node to another on SP4.

Loadleveler

We have put in place a mechanism to avoid the user having to choose and specify the LoadLeveler class: it will be chosen automatically by the LoadLeveler and any user class specification will be ignored. This will allow to the users to ignore this annoying detail: you only need to specify just what you need in term of resources. On our side this will allow to optimize the usage of resources and keep the waiting time proportional to the required resources. Users that will try to overtake this mechanism will have their accounts temporarily disabled.

Software

AIX is still version 5.2 as on SP4 and the compiled executable should still work. However the Power5 is very different architecturally from Power4, so to obtain the best performance it can be very useful to re-compile applications with "-qarch=pwr5 -qtune=pwr5" (included in -O4).

64 bit development environment

We judge the 64 bit environment now mature: we have put it by default, setting the environment variable OBJECT_MODE=64. This will allow to

- have no limit nor problems with memory;

– have better performance with MPI collective communications.

To revert to the old behavior unset the environment variable or use “-q32” in you compilation options.

Disk space

SP5 adheres completely to the CINECA Infrastructure.

$CINECA_HOME and $CINECA_DATA are IBM San FileSystem, with the client running on the login node, but all the rest of the nodes access SFS via NFS.

$CINECA_SCRATCH is an IBM GPFS filesystem, with high performance, but only if properly used: avoid reading/writing small blocks of data from GPFS.

$HOME is a NFS filesystems, with low performance and a limited user quota.

Compilers

Fortran77: xlf

Fortran90: xlf90, xlf95

C: cc (extended compatibility), xlc, c89 (ANSI C89), c99 (ISO C99)

C++: xlC

Examples and useful options

Basic Fortran77 compilation & linking:
```
    xlf program.f -o program
```

Compile & link Fortran90 sources with “.f90” suffix (the standard one for xlf90 is “.F”):
```
    xlf90 -qsuffix=f=f90 program.f90 -o program
```

Some debugging options. -g=include debug symbols, -C=check bounds of arrays and strings at run-time:
```
    xlf -g -C program.f -o program
```

Aggressive optimization. Can lead to modification in the order of the execution of instructions. Check the results of your code:
```
    xlf -O4 -qmaxmem=-1 program.f -o program
```

Compile only & than link. Aggressive optimization and inlining, also when the code is split in many source files. Remember: it is very difficult to debug inlined codes – do it only for production:
```
    xlf -O4 -Q -qipa=inline -qmaxmem=-1 -c program.f
    xlf -O4 -Q -qipa=inline -qmaxmem=-1 -c subroutine1.f
```

```
xlf -O4 -Q -qipa=inline -qmaxmem=-1 -c subroutine2.f
xlf -O4 -qipa=inline program.o subroutine1.o subroutine2.o -o program
```

Inlining is worthwhile only if the core of your code is made of many calls to small and fast procedures.

Documentation

Fortran77, Fortran90:
 http://www.ibm.com/software/awdtools/fortran/xlfortran/library/

C, C++:
 http://www.ibm.com/software/awdtools/vacpp/library/

Software intrinsic divide

The Power5 assembler contains hardware primitives for the operation x/y. Nevertheless these are not pipelined, and some applications may benefit from the usage of software alternatives in the main computing core. You have to compile, with the XL Fortran compiler, with the option "-qarch=pwr5" (included in -O4) and you will have at disposal these intrinsic functions doing the work, among the others: SWDIV, SWDIV_NOCHK.

Look at the documentation for more details.

Parallel programming

IBM provides a set of scripts that will add the needed options to allow the generated executable aware of the POE (Parallel Operating Environment).

MPI

To use the IBM MPI library and runtime just compile with the appropriate parallel compiler:

mpxlf, mpxlf90, mpcc, mpCC

OpenMP

Use the serial compiler with the option: -qsmp=omp

Mixed mode: MPI + OpenMP

Use the MPI compiler and -qsmp=omp as well.

Documentation

All the documentation related to parallel programming is relative to IBM PE (Parallel Environment):
 http://www.ibm.com/servers/eserver/pseries/library/sp_books/

Running applications

Interactive

You can run interactively a serial program in the standard UNIX way:

```
./program
```

However you can run interactively a parallel application with a command like:

```
poe ./program <program options> <POE options>
```

This will run the application running a LoadLeveler interactive job.

A typical interactive parallel submission with 4 processors:

```
poe ./program -procs 4 -nodes 1 -labelio yes
```

If you get repetitively the message:

```
ERROR: 0031-123  Retrying allocation .... press control-C to terminate
```

it means there are not enough resources at this time: another user is allocating the resources reserved for interactive usage. Just wait and assuming your request is reasonable, when the interactive resources are freed by the other user, your program will be run.

More information with "man poe" at the command prompt and in the Parallel Environment guides at:

http://www.ibm.com/servers/eserver/pseries/library/sp_books/

Batch: LoadLeveler

Batch jobs are managed by the LoadLeveler. Batch jobs must be submitted using the "llsubmit" command with a LoadLeveler script file .

The basic LoadLeveler commands:

`llsubmit script.cmd`	submit in batch queue the job described in the file "script.cmd". See below for the LoadLeveler scripts syntax
`llq`	Returns information about all the jobs waiting and running in the queues. In order to see the information about only your jobs use the command "llq -u $USER"
`llcancel joblist`	Cancels one or more jobs from the queuing system, either they are IDLE or RUNNING
`llstatus`	return information about the status of the machines

`llclass`	return information about the status of the queues (called "classes" by the LoadLeveler)

Users do not specify classes anymore: LoadLeveler will choose the appropriate class for the user. Any class selection performed by the user will be ignored. On SP5 it will be possible to submit jobs of the following types:

job type	***CPUs***	***max wall time***	***max mem/CPU***	***wait time***
parallel(*)	2-256	24:00:00	1.7 GB	days
parallel, high memory	2-16	24:00:00	7 GB	days
serial	1	15 days	10 GB	days

Furthermore, the following job types will be particularly favored from the scheduling point of view:

job type	***CPUs***	***max wall time***	***max mem/CPU***	***wait time***
test/debug	1-4	00:10:00	1.7 GB	minutes
serial/parallel, short	1-32	6:00:00	1.7 GB	hours
serial, short	1	3 days	4 GB	many hours

(*) Jobs larger than 32 CPUs will be run on dedicated nodes.

LoadLeveler script syntax

The LoadLeveler script tells LoadLeveler what resources are needed by your application at run time, in particular:

- maximum elapsed time:
  `#@ wall_clock_limit = hh:mm:ss`
- number of tasks (MPI):
  `#@ total_tasks = N`
- maximum memory per task:
  `#@ resources = ConsumableMemory(N mb)`
- number of CPUS/threads per task (OpenMP):
  `#@ resources = ConsumableCpus(N)`

ATTENTION: If you need more than 650MBs of memory for a serial job, or more of 650MBs per task for a parallel job, you absolutely MUST explicitly require the memory to the LoadLeveler with the `ConsumableMemory` keyword

The last LoadLeveler specification is `#@ queue`. After that you have to write the script, that will be executed when the submitted job is run.

Example 1: script for a SERIAL job

```
# @ wall_clock_limit = 72:00:00
# @ resources = ConsumableCpus(1) ConsumableMemory(1000 mb)
# @ job_type = serial
# @ output = job.log
# @ error = job.err
# @ shell = /bin/tcsh
# @ queue
... your commands here ...
```

Example 2: script for a PARALLEL job

```
# @ wall_clock_limit = 6:00:00
# @ resources = ConsumableCpus(1) ConsumableMemory(1000 mb)
# @ total_tasks = 64
# @ job_type = parallel
# @ network.MPI = csss,shared,US
# @ output = job.log
# @ error = job.err
# @ shell = /bin/tcsh
# @ queue
... your commands here ...
```

The network can be any of: network.MPI, network.LAPI or network.MPI_LAPI. The network specification will be overwritten with the best choice by the LoadLeveler.

Example3: script for an OpenMP PARALLEL job

```
# @ wall_clock_limit = 6:00:00
# @ resources = ConsumableCpus(8) ConsumableMemory(4 gb)
# @ job_type = serial
# @ output = job.log
# @ error = job.err
# @ shell = /bin/tcsh
# @ queue
... your commands here ...
```

Note that job_type = serial: you must not ask for a parallel job type, since OpenMP does not use POE.

For more information on LoadLeveler, consult the on-line guide from here:

http://www.ibm.com/servers/eserver/pseries/library/sp_books/

but keep in mind that many customizations have been performed on CINECA SP5 and some of the keywords are not allowed.

Debuggers

dbx

dbx is the standard AIX symbolic debugger. To include symbolic debugger information in your executables you must compile with the "-g" option.

To run an interactive executable under the control of the debugger:

```
    dbx ./program
```

To make a "post-mortem" analysis of a core file:

```
    dbx program core
```

pdbx

pdbx is the symbolic, textual, parallel debugger included with the IBM Parallel Environment. Remember to compile with "-g" to debug also your parallel codes. pdbx accepts the same options as poe. e.g.:

```
    pdbx ./program -procs 4 -nodes 1
```

Totalview

Totalview is the graphic symbolic debugger from ETNUS. To use it you must however comile with "-g". Since it is a graphic tool, you need a X11 server or emulator.

To debug a parallel code you must run a command line like this:

```
    totalview poe -a ./program -procs 4 -nodes 1
```

You will not see immediately your code, because at the very beginning you are debugging the "poe" executable. press "Go" and wait until poe spawns the copies of your code.

You can find Totalview documentation on the WEB:
    http://www.etnus.com/TotalView/index.html

Static debuggers

Two static debuggers are available on IBM SP4. Static debuggers are program which analyze source code to find problems such as uninitialized variables, argument data type mismatch and other semantic (not syntax) errors.

When your code is ready and the syntax error free (the compiler compiles it) you can check your code using these debuggers.

Language	**Command**
C / C++	`lint`

Fortran 77	`module load ftnchek`
	`ftnchek`

View "man lint" or "man ftnchek" (after module load ftnchek) for information about these tools.

Profilers

Profiling is the first activity needed to optimize your code. Profiling give you information of where (in terms of subroutines or code-lines) your code spends the most of its time.

gprof

Basic textual AIX profiler.

Compile your code with the options “-g -pg”:

```
    xlf -g -pg program.f -o program
```

When you run the code, it will save a sampling file, “gmon.out”. Because of that the performance of the code is worse, and you can not use profiler data for performance, but the information is still very useful.

After the end of the execution run:

```
    gprof program gmon.out
```

Xprofiler

to be confermed and completed

HPM

to be confirmed and completed

MPI_Trace

With MPI_trace you do not profile the executed code, but the MPI library calls.

to be completed

Vampir

With Vampir you have access to very detailed data about the flow of the communications in MPI codes.

to be completed

Scientific Libraries

ESSL

Highly optimized blas (in particular, dgemm) libraries are inside the essl. The essl contains also a “sort of” LaPACK: the routines are the same, the name also, but the arguments differ from the “netlib lapack”. We suggest you use the netlib lapack for highest code portability, but link using the essl to use the optimized blas subroutines, loading the relative module (that initialize the environment variables) and adding at link time the following libraries specifications (remember: THE ORDER IS IMPORTANT):

```
module load lapack
-L$LAPACK_LIB -llapack -lessl
```

The same applies to the pessl, that contains a “sort of” ScaLaPACK:

```
module load scalapack
-L$SCALAPACK_LIB -lscalapack -lpblas -lblacs -lblacsF77init -lessl
```

ESSL has also an SMP parallel version, that allows you to use the shared memory architecture of the single nodes to perform linear algebra operations. This can be a very straightforward way to parallelize your code: just link with “-lesslsmp”.

By default this will use the same number of threads of the CPUs of the node (8 on sp5). To reduce the number of threads per task, set the environment variable “OMP_NUM_THREADS” to the desired number of threads per task, before than running the application.

For documentation on IBM ESSL and PESSL find information here:
http://www.ibm.com/servers/eserver/pseries/library/sp_books/

MASS

MASS (Mathematical Accelerations SubSystem) library is available. It allows the speedup of codes that make heavy usage of mathematical functions (sin, cos, exp, sqrt, ...), especially if used repeatedly, for which a vector version exists. The computations performed with this library are precise, but do not strictly adhere to the IEEE standard.
To use this library load the relative module and link adding the following library specifications:

```
module load mass
-L$MASS_LIB -lmass -lmassv
```

Find documentation about the MASS library here:
http://www.ibm.com/software/awdtools/mass/

NAG

The NAG Fortran Library is a comprehensive collection of Fortran 77 and Fortran 90 routines for the solution of numerical and statistical problems. Presently only the sequential library is available. To use this library, load the relative module and link adding one of the following library specifications:

```
module load nag
-L$NAG_LIB -lnag
-q64 -L$NAG_LIB -lnag64
-L$NAG_LIB -lnagsmp
-q64 -L$NAG_LIB -lnag_use_essl64 -lessl
-L$NAG_LIB -lnag
-L$NAG_LIB -lnag
```

Find the documentation of the NAG library below the directory $NAG_DOC or on the web:

http://www.nag.co.uk/numeric/fl/manual/html/FLlibrarymanual.asp

CLX USER GUIDE

hostname: cl.clx.cineca.it

early availability: December 2003

start of production: June 1st, 2004

System architecture

CLX is an IBM Linux Cluster 1350, made (mostly) of 512 2-way IBM X335 nodes. Each computing node contains 2 Xeon Pentium IV processors. All the compute nodes have 2GB of memory (1GB per processor).

768 processors of CLX are Xeon Pentium IV at 3.06 GHz with 512MB of L2 cache and the Front Side Bus (FSB) at 533MHz.

256 processors, bought at the beginning of 2005, are Xeon Pentium IV EM64T (Nocona) at 3.00GHz with 1024MB of L2 cache, the FSB at 800MHz and support Hyper-Threading Technology.

All the CLX processors are capable of 2 double precision floating point operations per cycle, using the INTEL SSE2 extensions.

Login and service node processors are at 2.8GHz and have more memory.

All the nodes are interconnected to each other through a Myrinet network (http://www.myricom.com), capable of a maximum bandwidth of 256MB/s between each pair of nodes. The core component of the network is a pair of M3-CLOS Myrinet-2000 switches in CLOS256+256 configuration.

The global peak performance of CLX is of 6.1 TFlops.

The queuing system of CLX is LSF (it was OpenPBS up to Agoust 2005).

Disk space

CLX from Agoust 2005 conforms totally to the CINECA Infrastructure.

Using installed applications: module

Attention: if you use a given set of modules to compile an application, very probably you will need the same modules to run it, because by default linking is dynamic on linux systems, and the application will need at runtime the shared libraries of compiler and libraries. To minimize the number of needed modules at runtime, use static linking to compile the applications.

Compilers

Available compilers are standard GNU gcc and g77, GNU g95, INTEL and Portland Group (PGI) compilers. After loading the appropriate module, the command:

```
man compiler_command
```

give you the complete list of the flags supported by the compiler.

INTEL

Initialize the environment with one of the module commands:

```
module load compiler/intel
module load compiler/intel7
module load compiler/intel8
```

Fortran77 and Fortran90 compiler:

```
ifort
```

C and C++ compiler:

```
icc
```

Find the documentation of the two compilers respectively in the directories:

```
$IFORT_DOC
$ICC_DOC
```

Some optimizations we do suggest:

- to align data with cache-line boundaries, tune to Pentium Xeon processor and -O3 optimizations:

```
-align -tpp7 -O3
```

- to add loop vectorization (SSE/SSE2 instructions) to improve loops performance:

```
-align -tpp7 -O3 -xN -ftz
```

PGI

Initialize the environment with one of the module commands:

```
module load compier/pgi
```

The Fortran77, Fortran90, C and C++ compilers are respectively:

```
pgf90
pgf77
pgcc
pgCC
```

Find the documentation of the PGI compilers in the directory:

```
$PGI_DOC
```

Some optimizations we do suggest:

- to align data with cache-line boundaries, tune to Pentium Xeon processor and -O3 optimizations:

```
-Mcache_align -tp p7 -O3
```

best PGI suggested combination:

```
-Mcache_align -tp p7 -fast
```

best PGI suggested combination to use loop vectorization (SSE/SSE2 instructions) to improve loops performance:

```
-Mcache_align -tp p7 -fastsse
```

GNU

g77 and gcc are always available but are not the best optimizing compilers.

Optimize with -O2 or -O3.

To try the GNU g95 compiler you have to load the relative module before:

```
module load compiler/g95
```

Parallel programming

The parallel programming is mainly based on the MPICH-GM version of MPI (myrinet enabled MPI). The main four parallel-MPI compilers available are:

```
mpif90
mpif77
mpicc
mpiCC
```

for Fortran90, Fortran77, C and C++ respectively. These command names are the same for all suites of compilers, but they behave differently depending on the module you have loaded.

In all cases you will run the applications compiled with the parallel compiler with the command:

```
mpirun <executable>
```

Remember: you can use mpirun only within LSF scripts or LSF interactive sessions.

To choose the desired underlying compiler, select the appropriate environment with one of the commands:

```
module load mpich/intel
module load mpich/pgi
module load mpich/gnu
```

Use

```
module avail
```

to know what versions are available.

A version of MPICH, especially useful for debugging and for third party codes, is the one that does not rely on Myrinet protocol, but uses standard TCP/IP instead. The name of the modules for that version of MPICH are:

```
mpich-p4/gnu
mpich-p4/intel
mpich-p4/pgi
```

Default environment

The default environment of CLX has the following loaded modules:

```
mpich/intel
compiler/intel
```

In fact the compiler Intel is the best compiler for the HPC. If you need to use another compiler and MPI version, unload the intel modules, or clean totally the environment with:

```
module purge
```

Debugging

Enabling compiler runtime checks

Pay attention: some flags are available only for the Fortran compiler.

INTEL

Compile and link using

```
-O0 -g -traceback -fpstkchk -check bounds -fpe0
```

no optimizations, debug info, check array for addressing into correct bounds, floating point exceptions trap

If at runtime your code dies, then there is a problem. You can run your code using the debugger or analyze the core (core not available with PGI compilers).

PGI

Compile and link using

```
 -O0 -g -C -Ktrap=ovf,divz,inv
```

no optimizations, debug info, check array for addressing into correct bounds, floating point exceptions trap

GNU

Compile and link using

```
-O0 -g -Wall -fbounds-check
```

no optimizations, debug info, high level warning, check array for addressing into correct bounds

Serial debuggers

Intel

```
idb –gdb ./executable
```

see gdb and idb documentation

PGI

```
pgdbg  ./executable
```

see pgdbg documentation

GNU

```
gdb ./executable
```

Core file analysis

Create core ONLY in the /scratch area, to do not exceed your home quota!

First, enable core dumping

```
bash:       ulimit -c unlimited
csh/tcsh:   limit coredumpsize unlimited
```

If you are using Intel compiler, set the following environment variable:

```
bash:       export decfort_dump_flag=TRUE
csh/tcsh:   setenv decfort_dump_flag TRUE
```

Run your code and create the core file. To analyze it:

INTEL

```
idb -gdb ./executable core
```

GNU

```
gdb ./executable core
```

Parallel debugger

totalview is available for debugging parallel codes. Contact us to have permission to use it and see documentation here

http://www.etnus.com/Documentation/

Performance optimization

Software profilers: gprof

In order to check where your code spends most of its time, you can use gprof.
It uses data collected by the -pg compiling option to construct a text display of the functions within your application (call tree and CPU time spent in every subroutine).

gprof provides quick access to the profiled data, which lets 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:

```
compiler_name -pg <optimization flags> -o filename filename.f
```

run the program filename and get the output profiling file gmon.out Finally perform profiling

```
gprof filename 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:

```
compiler_name -g -pg <optimization flags> -o filename filename.f
gprof -annotated-source filename gmon.out
```

Scientific libraries

MKL

MKL is the Intel Math Kernel Library. It contains a very highly optimized BLAS library, LaPACK and more Intel highly optimized routines

To compile and link with MKL you need to add the following flags:

compiling:

```
-I$MKL_INCLUDE
```

linking:

```
-L$MKL_LIB -lmkl_lapack -lmkl_ia32 -lguide
```

Find the MKL documentation on the CLX in `$MKL_DOC`. You can view the MKL manual on line with the command:

```
acroread $MKL_DOC/mklman.pdf
```

or download and view it locally.

XD1 USER GUIDE

hostname: xd.xd1.cineca.it

early availability: June 20th, 2005

start of production: to be determined

System architecture

XD1 is a "Cray XD1" cabinet, fully populated with 72 2-way nodes, totally 144 AMD 64-bit Opteron processors at 2.4GHz. Each node has 4GBs of memory (2GBs per processor).

The peak performance of XD1 is of 690 GFlops.

The CPUs of XD1 are interconnected with the Cray RapidArray network, directly attached to the Opteron processors using the AMD HyperTransport technology. On our system RapidArrays allow the transfer of data between each pair of nodes at 3.2 GB/s (except for nodes hosting the FPGAs, where each processor has its own RapidArray connection giving up to 6.4GB/s) and the key characteristic of this network is the extremely low latency: 1.7μs measured with MPI.

Another key feature of XD1 is the Linux Synchronized Scheduler (LSS): each node runs its own Linux Kernel, but all the kernels of a set of nodes can be synchronized, in order to keep the parallel execution of user applications coherent and to improve the scalability. To use such feature, you must use the command **`xd1launcher`** (see "Parallel Programming" below).

The last feature of XD1 is that 6 of the nodes are equipped with an additional Rapid Array connection (giving 6.4GB/s maximum interconnection bandwidth to these nodes) and a

"Xilinx Virtex II Pro" FPGA (Field Programmable Gate Array), that can be used as co-processor to speedup significantly some specific applications. We are investigating the possibilities offered by such specific hardware with HPC applications.

You can find more details about XD1 here:

http://www.cray.com/products/xd1/index.html

and also on the documentation local to the machine, in the directory:

```
/opt/XD1/documents
```

Use "acroread" at the command prompt to read it, or download it at your site, but, ATTENTION, this documentation is covered by Cray Copyright and only users of XD1 can consult it. Do not redistribute.

Disk space

XD1 at the moment access $HOME, small, and /scratch, that is larger, but is mounted with the slow NFS protocol. Do not launch simulations that create large amount of data and that need an high I/O bandwidth because at now it is not available.

$CINECA_SCRATCH will be based upon LUSTRE as soon as Cray makes it available.

Using installed applications: module

Attention: if you use a given set of modules to compile an application, very probably you will need the same modules to run it, because by default linking is dynamic on linux systems, and the application will need at runtime the shared libraries of compiler and libraries. To minimize the number of needed modules at runtime, use static linking to compile the applications.

Sequential programming

The best performance compiler is the PGI, that is available by default. The module "pgi" should be loaded by default (check it with the command "module list").

The "default" PGI compiler version is set to the most stable one.

The PGI compiler commands:

```
pgf90
pgf77
pgcc
pgCC
```

However the GNU compiler is available without loading any module (gcc, g++, g77), but the PGI compiler optimize much better.

Parallel programming

The parallel programming is mainly based on the MPICH version of MPI customized by Cray in order to support the Rapid Array interconnect.

To use mpich type:

```
module load mpich
```

that defaults to the best stable mpich/PGI available. To use different versions, choose among the available modules (view them with “module avail”).

Available parallel compilers are:

```
mpif90
mpif77
mpicc
mpicxx
```

For Fortran90, Fortran77, C and C++ respectively. To run the application you have compiled with one of them use:

```
mpirun ./executable
```

Remember: you can use mpirun only within LSF scripts or LSF interactive sessions.

Using the Linux Synchronous Scheduler

To use the LSS, you must use the command “xd1launcher” to run the single tasks of an MPI execution, in this way:

```
mpirun $XD1LAUNCHER ./executable
```

Find more information about xd1launcher with:

```
man xd1launcher
```

Scientific libraries

ACML

The AMD Core Math Library (ACML) is a set of numerical routines tuned specifically for AMD64 platform processors (including Opteron(TM) and Athlon(TM) 64). The routines, which are available via both FORTRAN77 and C interfaces, include:

- BLAS - Basic Linear Algebra Subprograms (including Sparse Level 1 BLAS);
- LAPACK - A comprehensive package of higher level linear algebra routines;
- FFT - a set of Fast Fourier Transform routines for real and complex data.

To use ACML, load the acml module. Find the needed information typing:

```
    module help acml
```

After you have loaded the module, you can find on-line ACML documentation in `$ACML_DOC`

Using the FPGAs

Only 6 nodes of the XD1 are equipped with FPGA. On these nodes you can both:

- run special applications, interacting with this customizable hardware;
- experiment applications communicating with very high bandwidth among nodes.

To address these nodes both interactively and in batch, you must add the option “-R fpga” to the bsub requests.

EXAMPLE:

To obtain an interactive session on one host containing the FPGA:

```
    bsub -Is -R fpga -W 10 /bin/tcsh
```

FAQ: Frequently Asked Questions

Questions

How do I connect to CINECA's systems?

You must use ssh (secure shell). All current UNIX systems have the ssh client installed by default. To connect to our system you have to type, at your UNIX prompt:

```
ssh username@hostname
```

Once connected you should be able also to use the X11 applications remotely, even if you are behind a firewall, because ssh intalled on our systems performs the X11-forwarding.

From a Windows system you have to install a Windows ssh client. You can get an Open client from here:

http://www.openssh.com/windows.html

If you need to export the graphical DISPLAY on a Windows system, you will need an X11 emulator, like Xmanager, Exceed, Xvision and similars.

Why doesn't the “module” command work?

We should have fixed it on all the systems, for both bash and tcsh.

If your login shell is not one of these two, ask us to change it to one of them.

You can check what is your login shell with the command:

```
echo $SHELL
```

If it still does not work, avoid using the module initialization script in your job scripts, and contact us.

How do I measure how much time has elapsed for a part of my program?

Maybe you need to *profile* your code – see the next question.

If not, and you really need to time a part of your code, remember to use real time clocks, don't use the “user time” (the cpu time), because it might not take in account for I/O, swapping, MPI communications, all events that however contribute to the time elapsed by your routine. In the following there are some solutions:

Fortran90:

SYSTEM_CLOCK is standard Fortran90, very efficient, since it is a language intrinsic.

```
integer counti, countf, count_rate
real dt
call system_clock(counti,count_rate)
... work ....
call system_clock(countf)
dt=REAL(countf-counti)/REAL(count_rate)
```

MPI (both Fortran and C):

MPI_WCLOCK is very portable and general if you are writing an MPI code. However it is an MPI routine and it has some overhead:

```
double precision t1,t2,dt
...
t1 = MPI_WCLOCK()
... work ...
t2 = MPI_WCLOCK()
deltat = t2-t1
```

or

```
double t1,t2,dt;
...
t1 = MPI_Wclock();
... work ...
t2 = MPI_Wclock();
deltat = t2-t1;
```

C (UNIX):

If you need timing in C and you are not using MPI, you must avoid the typical routines time, times, clock, because all of them return the user time, not the real one. To get the real-time use gettimeofday(), standard UNIX, that is precise up to microseconds:

```
#include <stddef.h>
#include <sys/time.h>
...
  double t1,t2,elapsed;
  struct timeval tp;
  int rtn;
```

```
rtn=gettimeofday(&tp, NULL);
t1=(double)tp.tv_sec+(1.e-6)*tp.tv_usec;
... work ...
rtn=gettimeofday(&tp, NULL);
t2=(double)tp.tv_sec+(1.e-6)*tp.tv_usec;
elapsed=t2-t1;
```

What does it mean “profiling”?

When you need to optimize your code, the first information you need is to know in what part it spends most of its time. The profiler is a tool that comes with all the development systems. You need to compile the code with a special option (typically -pg), to run it and than to analyze the profiling data (typically in the file gmon.out) using the profiler (usually gprof). However profiling usually alters the time elapsed by your code, but the relative timings are still very useful.

Look at the system specific guides for specific information about specific tools.

Why did my code die? What is the “core” file?

The core file is the snapshot of your application, taken by the Operating System when it died. The first thing to do is to discover where the application has died:

```
dbx program.exe core
dbx> where
```

“dbx” is one of the typical debuggers. You can do it with any (gdb, ddd, pdbx, totalview...).

The “where” instruction tells you exactly in which subroutine stack the execution was running when it died. If you compiled the application with “-g” you will be able to see the code lines and to inspect the content of the variables at the time of the dump, with the “p” debugger command. Furthermore, you will be able to inspect the variables at all the levels of the stack with the “up” and “down” commands.

Why did my code die with this message: “Warning: no access to tty (Bad file descriptor)”?

The message is related to the fact that in batch you do not have a terminal, it is harmless, and it is not the cause of the failure of your job.

Why did my code die with this message: “error while loading shared libraries...”?

The message means that the executable uses some shared library. When you compile on Linux systems, the libraries are linked dynamically by default. In fact the executable do not know where to find libraries that are not in standard places (/lib, /usr/lib,...).

The solution is to properly configure the environment in the job, initializing the modules and loading any module you needed to compile (of both compiler and libraries) also at run-time, before invoking the executable in the script.

How can I run many batch jobs, one after the other?

Every queuing system has the possibility to chain many jobs. For LoadLeveler what you need is a “multi-step” job and for LSF a so called “job chain”. In the following, the details related to the different queuing systems.

LSF

Submit the first job

```
    bsub < script_1
    Job <1845> is submitted to queue <queue_name>.
```

Read the JOB_ID (`1845`) that the command produce, and use it to submit the second job of the sequence with the option -w “done(previous)”:

```
    bsub -w ”done(1845)”< script_2
    Job <1846> is submitted to queue <queue_name>.
```

The second job will start only when the previous one is “done”. To add more jobs, use the JOB_ID of the second job, to submit the third one:

```
    bsub -q parallel -w ”done(1846)”< script_3
    Job <1847> is submitted to queue <queue_name>.
```

then you can continue in this way for all the other jobs of the sequence.

SP5: LoadLeveler

In a typical multi-step job, the various steps are terminated by the #@queue statement. However you need to tell to the LoadLeveler not to execute all the steps (this is the standard behavior), but to wait for the completion of the previous one (except for the first) with the #@dependency keyword. For example:

```
    #@ input = step00.inp
    #@ output = step00.out
    #@ executable = program00.exe
    #@ step_name = step00
    #@ queue
    ###repeat all the options on every step
    #@ input = step01.inp
    #@ output = step01.out
    #@ executable = program01.exe
    #@ step_name = step01
    ### with the following dependency you state that the "step00"
    ### have been completed correctly
    #@ dependency = step00 == 0
    #@ queue
```

BEWARE: if you have a shell script following the latest #@queue statement, or anywhere in the middle, keep in mind that ALL of it will be executed at each step! If you run in this way, you need to differentiate the various steps by using the appropriate environment variable: $LOADL_STEP_NAME. Example:

```
### your loadleveler command here
# @ shell = /bin/tcsh
# @ step_name = step00
# @ queue
### your loadleveler commands here again
# @ shell = /bin/tcsh
# @ step_name = step01
# @ dependency = step00 == 0
# @ queue

### lo script eseguito e' sempre lo stesso. L'esecuzione
### va quindi differenziata utilizzando la variabile di
### ambiente $LOADL_STEP_NAME (uso uno switch di csh)

echo "performing step *** $LOADL_STEP_NAME ***"

switch ($LOADL_STEP_NAME)
  case "step00":
    #
    # step00 commands here
    #
  breaksw
  case "step01":
    #
    # step01 commands here
    #
  breaksw
endsw

echo "END of step *** $LOADL_STEP_NAME"
```

LSF: The job I have submitted with bsub exits immediately without output. Why?

Very probably you have not redirected your script with “<”: bsub tries to run it as a command but can not, because the script should not be executable, and even if it was, eventual #BSUB lines are ignored.

REMEMBER:

```
bsub < jobscript
```

NOT:

```
bsub jobscript
```

LoadLeveler: What is the exact usage of the keyword “job_type”?

“job_type” valid values are “parallel” or “serial”. You will specify:

```
#@ job_type = parallel
```

when your job script contains one or more commands that rely on POE, that is, MPI or LAPI codes that have been complied with mpxlc, mpxlc... This must be done even if you will run on only one processor. job_type = parallel tells to the LoadLeveler to interact with POE.

Otherwise:

```
    #@ job_type = serial
```

when you do not use POE at all. In this case you will compile your code with xlf, xlc... Note that you can use this specification also for some kind of parallel jobs, like OpenMP, fork based,... However intra-node jobs. In this case you will allocate more than one CPU on one node with the ConsumableCpus specification. You can not use the “total_tasks” keyword if job_type is serial.