bullx scs 4 R4

bullx MPI User's Guide

extreme computing

REFERENCE
86 A2 83FK 03

Printed in France

Trademarks and Acknowledgements

We acknowledge the rights of the proprietors of the trademarks mentioned in this manual.

All brand names and software and hardware product names are subject to trademark and/or patent protection.

Quoting of brand and product names is for information purposes only and does not represent trademark and/or patent misuse.

Software

March 2014

Bull Cedoc
357 avenue Patton
BP 20845
49008 Angers Cedex 01
FRANCE

Table of Contents

Preface

The purpose of this guide is to describe **bullx MPI**, which is the Message Passing Interface parallel library recommended for extreme computing development applications.

The installation of all hardware and software components of the cluster must have been completed. The cluster Administrator must have carried out basic administration tasks (creation of users, definition of the file systems, network configuration, etc.).

Note You are advised to consult the Bull Support Web site for the most up-to-date product information, documentation, firmware updates, software fixes and service offers: http://support.bull.com

Intended Readers

This guide is aimed at MPI Application Developers of **bullx supercomputer suite** clusters.

Highlighting

The following highlighting conventions are used in this guide:

Bold	Identifies the following: • Interface objects such as menu names, labels, buttons and icons. • File, directory and path names. • Keywords to which particular attention must be paid.
Italic	Identifies references such as manuals or URLs.
`monospace`	Identifies portions of program codes, command lines, or messages displayed in command windows.
< >	Identifies parameters to be supplied by the user.

```
Commands entered by the user
```

```
System messages displayed on the screen
```

WARNING

A *Warning* notice indicates an action that could cause damage to a program, device, system, or data.

Related Publications

Important **The Software Release Bulletin (SRB) delivered with your version of bullx supercomputer suite must be read first.**

- *Software Release Bulletin*, 86 A2 91FK
- *Documentation Overview*, 86 A2 90FK
- *Installation and Configuration Guide*, 86 A2 74FK
- *Extreme Pack - Installation and Configuration Guide*, 86 A2 75FK
- *bullx MC Administration Guide*, 86 A2 76FK
- *bullx MC Monitoring Guide*, 86 A2 77FK
- *bullx MC Power Management Guide*, 86 A2 78FK
- *bullx MC Storage Guide*, 86 A2 79FK
- *bullx MC InfiniBand Guide*, 86 A2 80FK
- *bullx MC Ethernet Guide*, 86 A2 82FK
- *bullx MC Security Guide*, 86 A2 81FK
- *bullx EP Administration Guide*, 86 A2 88FK
- *bullx PFS Administration Guide*, 86 A2 86FK
- *bullx MPI User's Guide*, 86 A2 83FK
- *bullx DE User's Guide*, 86 A2 84FK
- *bullx BM User's Guide*, 86 A2 85FK
- *bullx MM Argos User's Guide*, 86 A2 87FK
- *Extended Offer Administration Guide*, 86 A2 89FK
- *bullx scs 4 R4 Documentation Portfolio, 86 AP 23PA*
- *bullx scs 4 R4 Documentation Set, 86 AP 33PA*

This list is not exhaustive. Useful documentation is supplied on the Resource & Documentation CD(s) delivered with your system. You are strongly advised to refer carefully to this documentation before proceeding to configure, use, maintain, or update your system.

Chapter 1. Introduction to bullx MPI Development Environment

1.1 The Program Execution Environment

When a user logs onto the system, the login session is directed to one of several nodes where the user may then develop and execute their applications. Applications can be executed on other cluster nodes apart from the user login system. For development, the environment consists of various components described in this section.

1.1.1 Parallel Processing and MPI libraries

bullx MPI is a set of tools and libraries that provide support to users throughout a project, from design to production.

bullx MPI is based on **OpenMPI**, the open source MPI 2.1 standards-compliant library. **bullx MPI** enables scalability for tens of thousands of cores with functions such as:

- Network-aware collective operations
- Fine grained process affinity thanks to integration with **bullx Batch Manager**
- Zero memory copy for intra-node communication

Other features, such as effective abnormal communication pattern detection and multi-path network failover, have been implemented to enhance the reliability and resilience of **bullx MPI.**

See *Chapter 2* for more details.

1.1.2 Resource Management

The resource manager is responsible for the allocation of resources to jobs. The resources are provided by nodes that are designated as compute resources. Processes of the job are assigned to and executed on these allocated resources.

bullx Batch Manager is based on **SLURM** (Simple Linux Utility for Resource Management) and has the following functions.

- It allocates compute resources, in terms of processing power and Compute Nodes to jobs for specified periods of time. If required the resources may be allocated exclusively with priorities set for jobs.
- It launches and monitors jobs on sets of allocated nodes, and will also resolve any resource conflicts between pending jobs.
- It includes new scheduling policies, such as fair-sharing, pre-emptive and backfilling policies
- It implements topology-aware resource allocation aimed at optimizing job performance, by taking into consideration the topology and interconnect performance.

See The *bullx BM User's Guide* for more information.

1.1.3 Batch Management

The batch manager is responsible for handling batch jobs for extreme computing clusters.

PBS-Professional, a sophisticated, scalable, robust Batch Manager from **Altair Engineering** is supported as a standard, and can be used as batch manager. **PBS Pro** can also be integrated with the **MPI** libraries.

Important **PBS Pro does not work with SLURM and should only be installed on clusters which do not use SLURM.**

See *Extended Offer Administration Guide* for details regarding the installation and configuration of **PBS-Professional**.

1.1.4 Linux Tools

Standard Linux tools such as **GCC** (a collection of free compilers that can compile C/C++ and FORTRAN), **GDB Gnu Debugger**, and other third-party tools including the **Intel** FORTRAN Compiler, the Intel C Compiler, Intel MKL libraries and Intel Debugger **IDB** and the **padb** parallel debugger.

1.1.5 Modules

Modules software provides a means for predefining and changing environments. Each one includes a compiler, a debugger and library releases, which are compatible with each other. It is easy to invoke one given environment in order to perform tests and then compare the results with other environments.

See The *bullx DE User's Guide* for details on Modules.

1.2 Data and Files

bullx Parallel File System is based on the open source parallel file system, **Lustre**. Specific features have been developed and integrated to improve scalability, performance and resilience:

- Support of Lustre servers with twin InfiniBand links
- Support for fault-tolerant cells, e.g. Lustre servers with up to 4 nodes
- Silent deadlock and denied access event detection thanks to integration with **bullx Management Center**
- Integration of Lustre's dedicated **Shine** administration tool, with bullx Management Center

Application file I/O operations may be performed using locally mounted storage devices, or alternatively, on remote storage devices using either **Lustre** or the **NFS** file systems.

See The *bullx PFS Administration Guide* for more information on **Lustre**.

Chapter 2. Using bullx MPI Parallel Library

A common approach to parallel programming is to use a message passing library, where a process uses library calls to exchange messages (information) with another process. This message passing allows processes running on multiple processors to cooperate.

Simply stated, a **MPI** (Message Passing Interface) provides a standard for writing message-passing programs. A **MPI** application is a set of autonomous processes, each one running its own code, and communicating with each other through calls to subroutines of the **MPI** library.

This chapter describes the MPI interface used with bullx Extreme Computing:

Programming with MPI

It is not in the scope of this guide to describe how to program with MPI. Please, refer to the web, where you will find complete information.

2.1 Overview

bullx MPI is based on the Open Source **Open MPI** project. **Open MPI** is an **MPI-2** implementation that is developed and maintained by a consortium of academic, research, and industry partners. **Open MPI** offers advantages for system and software vendors, application developers and computer science researchers.
This library enables dynamic communication with different device libraries, including **InfiniBand (IB)** interconnects, socket Ethernet/IB devices or single machine devices.
bullx MPI conforms to the **MPI-2** standard and supports up to the **MPI_THREAD_SERIALIZED** level.

Note As **bullx MPI** is based on **Open MPI**, most of the documentation available for Open MPI also applies to **bullx MPI**. You can therefore refer to http://open-mpi.org/faq/ for more detailed information.

2.1.1 OpenMPI Version

The OpenMPI version from which bullx MPI derivates, is indicated in the file **/opt/mpi/bullxmpi/<current_version>/share/doc/bullxmpi-<current_version>/NEWS.BULL**.

To display the OpenMPI version, assuming the current bullx MPI version is 1.2.7.1, enter:

```
# head /opt/mpi/bullxmpi/1.2.7.1/share/doc/bullxmpi-1.2.7.1/NEWS.BULL
```

```
============================
bullxmpi 1.2.7 Release Notes
============================

Based on Open MPI 1.6.4.
See NEWS file for more informations


bullx MPI Additional Features
============================
```

2.1.2 Quick Start for bullx MPI

bullx MPI is usually installed in the **/opt/mpi/bullxmpi/<version>** directory.

To use it, you can either:

- Use the **mpivars.{sh,csh}** environment setting file, which may be sourced from the **${bullxmpi_install_path}/bin** directory by a user or added to the profile for all users by the administrator
- Or use module files bundled with **bullx MPI**.

mportant **If you are using Intel compilers, you have to set the compilers environment before setting the bullx MPI environment.**

2.2 Compiling with bullx MPI

MPI applications should be compiled using **bullx MPI** wrappers:

C programs	mpicc your-code.c
C++ programs	mpiCC your-code.cc or mpic++ your-code.cc (for case-insensitive file systems)
F77 programs	mpif77 your-code.f
F90 programs	mpif90 your-code.f90

Wrappers to compilers simply add various command line flags and invoke a back-end compiler; they are not compilers in themselves.

bullx MPI currently uses **Intel C** and **Fortran** compilers to compile MPI applications.

For each wrapper, there is a file named **<wrapper>-data.txt** located in **/opt/mpi/bullxmpi/<version>/share/bullxmpi** which defines the default parameters for the compilation.

Additionally, you can export environment variables to override these settings, for example:

OMPI_MPICC=gcc
OMPI_MPICXX=g++

See The *Compiling MPI applications* **FAQ** available from http://www.open-mpi.org for more information.

2.3 Running with bullx MPI

bullx MPI comes with a launch command: **mpirun**.

mpirun is a unified processes launcher. It is highly integrated with various batch scheduling systems, auto-detecting its environment and acting accordingly.

Running without a Batch Scheduler

mpirun can be used without a batch scheduler. You only need to specify the Compute Nodes list:

```
$ cat hostlist
node1
node2
$ mpirun -hostfile hostlist -np 4 ./a.out
```

Running with SLURM

mpirun is to be run inside a SLURM allocation. It will auto-detect the number of cores and the node list. Hence, **mpirun** needs no arguments.

```
salloc -N 1 -n 2 mpirun ./a.out
```

You can also launch jobs using **srun** with the port reservation mechanism:

```
srun -N 1 -n 2 ./a.out
```

Notes

- From bullx scs 4 R4, the **–resv-ports** option is replaced by **–mpi=pmi2**. This option is not needed if the option **MpiDefault=pmi2** is set in **slurm.conf**. To know if this option is set, enter:
  scontrol show config|grep MpiDefault
- **MPI-2** dynamic processes are not supported by **srun**.
- The port reservation feature must be enabled in **slurm.conf** adding a line of this type: **MpiParams=ports=13000-14000**

Running with PBS Professional

To launch a job in a **PBS** environment, just use **mpirun** with your submission:

MPI-2 dynamic processes are not supported by **srun**.

```
#!/bin/bash
#PBS -l select=2:ncpus=1
mpirun ./a.out
```

2.4 Binding with bullx MPI

Note The examples will be for 2 socket and 8 cores per socket nodes.

2.4.1 Binding a Full MPI Application

Binding with miprun

Note By using the **–report-bindings** option, **mpirun** will display a view of the binding.

By default, **mpirun** binds each process to one core. So, in most cases, **mpirun** needs no arguments.
The following command:

```
salloc -N2 -n32 mpirun ./a.out
```

is equivalent to:

```
salloc -N2 -n32 mpirun --bycore --bind-to-core ./a.out
```

Binding with srun

```
srun -N2 -n32 --distribution=block:block ./a.out
```

2.4.2 Binding Hybrid Applications

Lot of applications use hybrid MPI/OpenMP parallelization. In this case you have to bind MPI process on multiple cores, as described in the following examples.

- To bind to all cores of an entire socket with **mpirun**:

Note The **–exclusive** SLURM option is mandatory when binding hybrid applications with mpirun.

```
salloc -N1 -n2 --exclusive mpirun --bind-to-socket --bysocket ./a.out
```

- To bind to all cores of an entire socket with **srun**:

```
srun -N1 -n2 --distribution=block:cyclic --cpu_bind=sockets ./a.out
```

- To bind to 4 cores in order to have 4 available threads per MPI process with **mpirun**:

```
salloc -N1 -n4 --exclusive mpirun --cpus-per-rank 4 ./a.out
```

- To bind to 4 cores in order to have 4 available threads per MPI process with **srun**:

```
srun -N1 -n4 -c4 --distribution=block:block --cpu_bind=cores ./a.out
```

2.5 Configuring and Tuning bullx MPI

Parameters in **bullx MPI** are set using the **MCA** (Modular Component Architecture) subsystem.

2.5.1 Obtaining Details of the MPI Configuration

The **ompi_info** command is used to obtain the details of your **bullx MPI** installation - components detected, compilers used, and even the features enabled. The **ompi_info -a** command can also be used; this adds the list of the **MCA** subsystem parameters at the end of the output.

Output Example

```
MCA btl: parameter "btl" (current value: <none>, data source: default value)
Default selection set of components for the btl framework (<none> means use all
components that can be found)
```

The parameter descriptions are defined using the following template:

```
MCA <section> : parameter "<param>" (current value: <val>, data source: <source>)
                <Description>
```

2.5.2 Setting the MCA Parameters

MCA parameters can be set in 3 different ways: Command Line, Environment Variables and Files.

Note The parameters are searched in the following order - Command Line, Environment Variables and Files.

Command Line

The Command line is the highest-precedence method for setting **MCA** parameters. For example:

```
shell$ mpirun --mca btl self,sm,openib -np 4 a.out
```

This sets the **MCA** parameter **btl** to the value of **self,sm,openib** before running **a.out** using four processes. In general, the format used for the command line is:
--mca <param_name> <value>

Note When setting multi-word values, you need to use quotes to ensure that the shell and **bullx MPI** understand that they are a single value. For example:

shell$ mpirun –mca param "value with multiple words" ...

Environment Variables

After the command line, environment variables are searched. Any environment variable named **OMPI_MCA_<param_name>** will be used. For example, the following has the same effect as the previous example (for **sh**-flavored shells):

```
shell$ OMPI_MCA_btl=self,sm,openib
shell$ export OMPI_MCA_btl
shell$ mpirun -np 4 a.out
```

Or, for **csh**-flavored shells:

```
shell% setenv OMPI_MCA_btl "self,sm,openib"
shell% mpirun -np 4 a.out
```

Note When setting environment variables to values with multiple words quotes should be used, as below:

```
# sh-flavored shells
shell$ OMPI_MCA_param="value with multiple words"
# csh-flavored shells
shell% setenv OMPI_MCA_param "value with multiple words"
```

Files

Finally, simple text files can be used to set MCA parameter values. Parameters are set one per line (comments are permitted). For example:

```
# This is a comment
# Set the same MCA parameter as in previous examples
mpi_show_handle_leaks = 1
```

Note Quotes are not necessary for setting multi-word values in **MCA** parameter files. Indeed, if you use quotes in the **MCA** parameter file, they will be treated as part of the value itself.

Example

```
# The following two values are different:
param1 = value with multiple words
param2 = "value with multiple words"
```

By default, two files are searched (in order):

1. **$HOME/openmpi/mca-params.conf:** The user-supplied set of values takes the highest precedence.

2. **/opt/mpi/bullxmpi/x.x.x/etc/openmpi-mca-params.conf:** The system-supplied set of values has a lower precedence.

More specifically, the **MCA** parameter **mca_param_files** specifies a colon-delimited path of files to search for **MCA** parameters. Files to the left have lower precedence; files to the right are higher precedence.

Keep in mind that, just like components, these parameter files are only relevant where they are visible. Specifically, **bullx MPI** does not read all the values from these files during start-up and then send them to all nodes for the job. The files are read on each node during the start-up for each process in turn. This is intentional: it allows each node to be customized separately, which is especially relevant in heterogeneous environments.

2.5.3 Parameters Profiles

MCA parameters can also be specified using **profiles**. These are coherent sets of MCA parameters that can be used under certain circumstances, for example for a large-scale application, or for a micro-benchmark.
These parameters should be declared in a file that is then set for the **mpirun** command, using one of the following syntaxes (assuming that the MCA parameters profile is in the **my_profile.conf** file):

```
shell$ mpirun -p my_profile.conf ... a.out
```

or

```
shell$ mpirun --param-profile my_profile.conf ... a.out
```

or

```
shell% export OMPI_MCA_mca_param_profile=my_profile.conf
shell% srun ... a.out
```

2.5.4 Protecting MCA System-wide Parameters

If necessary, the System Administrator is able to prevent users from overwriting the default MCA system-wide parameter settings. These parameters should be declared in the **etc/openmpi-priv-mca-params.conf** privileged file.

Note This file has the same format as the **etc/openmpi-mca-params.conf** file, i.e. <param> = <value>.
Parameters declared in this file are to be removed from the default (non-privileged) **etc/openmpi-mca-params.conf.**

The MCA parameters declared in the **etc/openmpi-priv-mca-params.conf** file are considered as *non overridable*, and if (re)set in one of the following places below then a warning message from **Open MPI** will appear:

- **etc/openmpi-mca-params.conf**
- **$HOME/openmpi/mca-params.conf**
- Environment (by setting the **OMPI_MCA_<param>** environment variable)
- Command line (via the **-mca** option with **mpirun**)

The message will appear as below:

```
WARNING: An MCA parameter file attempted to override the privileged
MCA parameter originally set in /etc/openmpi-priv-mca-params.conf.

  Privileged parameter: btl
  MCA configuration file: /etc/openmpi-mca-params.conf

The overriding value was ignored.
```

The new parameter value is not taken into account: the test is run, using the value set in the MCA system-wide parameters file.

Effect of the mca_param_files Parameter

Even if the **mca_param_files** MCA parameter is used to change the search path for the parameters configuration files, the privileged parameters configuration file is read by default, even if not referenced in the specified path. For example, if the user sets:

```
shell$ mpirun --mca mca_param_file "/home/myself/file1:/home/myself/file2 -np 4
a.out
```

The files that are fetched for MCA parameters are:

- **/opt/mpi/bullxmpi/x.x.x/etc/openmpi-priv-mca-params.conf**
- **/home/myself/file1**
- **/home/myself/file2**

This prevents users from unintentionally bypassing system-wide parameters protection.

Note This protection does not make it impossible to circumvent (for example by rebuilding a library in the home directory). To prevent users from doing this, it is highly recommended to improve the message in **share/bullxmpi/help-mca-param.txt** (tag **[privileged-param-file]**) adding a custom message at the end.

2.6 MPI/IO with bullx MPI

The following sections describe how to use **bullx MPI** MPI/IO for the **NFS** and **Lustre** distributed file systems.

2.6.1 MPI/IO and NFS File Systems

Check the Mount options

To use **bullx MPI** and **NFS** together, the shared **NFS** directory must be mounted with the no attribute caching (**noac**) option added. Run the command below on the **NFS** client machines to check this:

```
mount | grep home_nfs
```

Note **/home_nfs** is the name of the mount point.

The result for **/home_nfs** should appear as below:

```
nfs_server:/home_nfs on /home_nfs type nfs
(rw,noac,lookupcache=positive,addr=xx.xxx.xx.xx)
```

If the **noac** and **lookupcache=positive** flags are not present, ask your System Administrator to add them. If the performance for I/O Operations is impacted, it is possible to improve performance by exporting the **NFS** directory from the NFS server with the **async** option.

Run the command below on the **NFS** server to check this:

```
grep home_nfs /etc/exports
```

The exports entry for **/home_nfs** should appear as below:

```
/home_nfs *(rw,async)
```

If the **async** option is not present, ask your System Administrator to add it.

Note The System Administrator will have to confirm that there is no negative impact when the **async** option is added.

2.6.2 MPI/IO on Lustre

Check the Mount options

To use **bullx MPI** and **Lustre** together, the shared **Lustre** directory must be mounted with the locking support (**flock**) option included. Run the command below on the **Lustre** client machines to check this:

```
mount | grep lustre_fs
```

Note **/lustre_fs** is the name of the mount point.

The result for **/lustre_fs** should appear as below:

```
lustre_srv-ic0@o2ib:/lustre_fs on /mnt/lustre_fs type lustre
(rw,acl,user_xattr,flock)
```

If the **flock** option is not present, ask your System Administrator to add it.

Check the User rights for the Directory

The User must have the read/write rights for the directory that he wants to access.

How to Configure the stripe_count of the File

An important parameter to improve **bullx MPI** performance with a **Lustre** file is the stripe count for the file. It configures the parallelization level for the I/O operations for the **Lustre** file. The best value to set for the stripe count is a multiple of the number of Compute Nodes on which the job runs.
For example, if the job runs on 10 nodes, a value of 10 or 20 for the stripe count means that it is correctly configured. A value of 5 is OK, because 5 processes can perform I/O operations, but a value of 1 is a poor value, because only 1 process will perform the I/O operations.
Run the following command to display the stripe count for a file (**/mnt/lustre/lfs_file** is the path to the **Lustre** file):

```
lfs getstripe -c /mnt/lustre/lfs_file
```

This will give a result, as shown in below, where the stripe count for the file is 10:

```
10
```

There are four different ways to set the stripe count. They are listed below in order of priority: if the first one is used, then the other three are not valid.

1. If the file still exists, it is not possible to change the stripe count, but you can create a new file, copy the original file into the new one, and then rename the new file with the original file name, as shown in the example commands below:

```
lfs setstripe -c 20 /mnt/lustre/lfs_file.new
cp /mnt/lustre/lfs_file /mnt/lustre/lfs_file.new
mv /mnt/lustre/lfs_file.new /mnt/lustre/lfs_file
lfs getstripe -c /mnt/lustre/lfs_file
```

```
20
```

The stripe count for this file is set to 20.

2. It is possible to force the stripe count value for <filename> when it is created within the application, by setting the **striping_factor** hint before the **MPI_File_open()** call in the source file, as shown in the example below.

```
MPI_Info_set( info, «striping_factor », 32);
MPI_File_open(MPI_COMM_WORLD, filename,
MPI_MODE_CREATE | MPI_MODE_RDWR, info, &fh);
```

   This will set the stripe count to 32.

3. It is possible to configure the stripe count using the **hint** file:

```
export ROMIO_HINTS=$HOME/hints
cat $HOME/hints
```

   This will give a result, as shown in the example below, where the stripe count for the file is 32:

```
striping_factor 32
```

4. When the I/O application calls the **MPI_File_open(comm, filename, amode, info, &fh)** function to create a file (**amode** contains the **MPI_MODE_CREATE** flag), **bullx MPI** will set the stripe count using the best option available, using the rules below;
   - If **N** (the number of Compute Nodes of the communicator **comm**) is less than **NO** (the number of OSTs), **bullx MPI** sets the stripe count to the highest value possible for the multiples of **N** that is less than **NO**.
   - Or **bullx MPI** will set the stripe count to number of **OST**s.

Examples

In these examples, N is the Number of Compute Nodes of the **comm** communicator.

- N = 10, Number of **OST**s = 15,
  ==> **bullx MPI** sets the stripe count to 10
- N = 10, Number of OSTs = 32,
  ==> **bullx MPI** sets the stripe count to 30
- N = 100, Number of OSTs = 32,
  ==> **bullx MPI** sets the stripe count to 32

Note You can disable the automatic stripe count calculation by setting the **mca** parameter **io_romio_optimize_stripe_count** to **0**. In this case bullx MPI will use the default directory stripe count.

2.7 Extra Functionalities

This section describes some of the extra functionalities for bullx MPI

2.7.1 Device Failover

Bull has improved Open MPI's default communication layer to add the possibility of switching dynamically from one network to another when there is a failure.

This improvement results from a new PML component called **ob1_df**, which is enabled by using the **maintenance_failover.conf** profile (see *Section 2.5.3 Parameters Profiles*)

Note **ob1_df** incurs an additional latency of around 0.2 µs

The default **ob1 PML** component is replaced by **ob1_df**. If a network generates a fatal error (link down, timeout, etc.) then **ob1_df** will automatically switch to the next available network.

InfiniBand Tuning

In there is an error with an **InfiniBand** network, the **OpenIB** component will try to resend data for a significant period before returning an error, and forcing **obl_df** switch to the next available port. If you would like to accelerate the switching time, reduce the **btl_openib_ib_timeout** parameter. The **InfiniBand** timeout period can be calculated by using the following formula:

Timeout = 4.096 microseconds * (2^btl_openib_ib_timeout)

The **btl_openib_ib_timeout** parameter can be set between 0 and 31.

2.7.2 Deadlock Detection

MPI polling is usually done via the use of busy loops. As most modern interconnects communicate directly with the network card in userspace, blocked processes will keep polling. This can be a problem for two reasons:

- It is hard to distinguish a blocked process from a communicating process.
- **MPI** delays result in higher **CPU** usage and power consumption.

To help resolve this problem, deadlock detection can be enabled. The **opal_progress_wait_count** parameter indicates the number of unsuccessful polling loops that have to pass, before the time count is started. This should be set to a reasonably high value in order to reduce the performance impact of the time count process, e.g. 1000. The default value of **opal_progress_wait_count** is **-1** (disabled).

The **progress_wait_trigger** parameter indicates the period before an action is performed (default 600s = 10 minutes).

Once the number of **progress_wait_trigger** seconds have passed without any activity, then the **opal_progress_wait_action** will be performed. The program may either switch to sleep mode (CPU usage should drop to 0% introducing micro sleeps between polls), display a message, or do both.

The codes for the **opal_progress_wait_action** setting are below:

0x1 : Sleep
0x2 : Warn
0x3 : Sleep and Warn

The micro sleep duration is configured by using the **opal_progress_wait_action_sleep_duration** parameter (default 10 ms).

2.7.3 GHC Component

GHC (Generalized Hierarchical Collective) is a component of the collective framework. It improves the communication performance between the processes of a cluster by taking their placement into account. This component is enabled by default.

It allows you to configure collective operations independently of each other. By default, the configuration file is located at:
$(INSTALLDIR)/etc/bullxmpi-ghc-rules.conf

You can specify another location by using the following parameter:

```
-mca coll_ghc_file_path new_ghc_configuration
```

2.7.4 WDC Framework

Occasionally, *unusual events* can occur when running an MPI application. These events are usually not fatal errors, but just erroneous situations that manifest under unusual circumstances. It then becomes important to notify the administrator or the user about these *unusual events*. The **bullx MPI** runtime ensures that applications run to completion, as long as no fatal errors occur. If the *unusual events* are not fatal, the bullx MPI runtime ignores them. Even though the application successfully completes, these events may result in significant performance degradation. This is not an issue if the *unusual events* are not frequent. However, they could be a real problem if they are frequent and may often be easily avoided.

WDC (**Warning Data Capture**) is a **MPI** framework that helps trace these *unusual events* by providing proper hooks, and has a low impact on the overall performance of the application.

Examples of events that used to be silently handled by **bullx MPI**, and can now be traced via **WDC**:

- During an **RDMA** read operation, if the receive buffer is not contiguous; the protocol is silently changed from **RDMA** to **copy in/out**.
- When the **IBV_EVENT_SRQ_LIMIT_REACHED** event is received, **bullx MPI** silently calls a handler whose job is to resize the SRQs dynamically to be larger.
- When a fragment cannot be immediately sent, it is silently added to a list of pending fragments so the send is retried later.

Activating the WDC basic Component

By default, this **WDC** framework is disabled. It can be activated by activating one of its components (**basic** or **oob**). Use the following **MCA** parameter to activate the basic component:

```
-mca wdc basic
```

When the basic component is activated, the traces are generated locally, on each Compute Node.

By default, the traces are generated in **syslog**. In order to change this behavior, use the following MCA parameter:

```
-mca wdc_basic_log_output output_stream
```

output_stream is a comma separated list of output log streams that are written to one (or more) of the **syslog**, **stdout**, **stderr**, or **file** outputs.

Notes

- If the **file** output stream is chosen, the traces are generated in a file called **output-wdc**, and located under **/tmp/<login>@<host>/<job family>/<local jobid>/<vpid>**
- If the **orte_tmpdir_base** MCA parameter has been changed, **/tmp,** above, should be changed to the new parameter value.
- Only the **root** user can read the **syslog**.

When **WDC** is activated, the traces might be generated frequently. In order to avoid disturbing the MPI application performance, by default, the traces are generated only once, during the finalize phase. Use the following MCA parameter to change this behavior:

```
-mca wdc_log_at_finalize 0
```

In this situation, the collected events are aggregated per MPI process and only traced if a defined threshold (counter threshold) has been reached. Another threshold (time threshold) can be used to condition subsequent traces generation for an event that has already been traced.

Use the following MCA parameters to change the default thresholds values:

```
-mca wdc_basic_cnt_thresh <c_value>
-mca wdc_basic_time_thresh <t_value in secs>
```

Activating the WDC oob Component

Use the following **MCA** parameter to activate the **oob** component:

```
-mca wdc oob
```

When the **oob** component is activated, the traces are relayed to the **hnp**. The **hnp** in turn is the one that actually generates the traces into the output stream.

Note When a job is not launched by **mpirun** (i.e. if it is launched by **srun**), the traces are not relayed to the **hnp**: they are instead generated locally, on each Compute Node. In this case, the behavior is that of the **basic** component.

By default, the traces are generated in **syslog**. In order to change this behavior, use the following MCA parameter:

```
-mca wdc_oob_log_output output_stream
```

output_stream is a comma separated list of output log streams that are written to one (or more) of the **syslog**, **stdout**, **stderr**, or **file** outputs.

Notes

- If the **file** output stream is chosen, the traces are generated in a file called **output-wdc**, and located under **/tmp/<login>@<host>/<job family>/<local jobid>/<vpid** >
- If the **orte_tmpdir_base** MCA parameter has been changed, **/tmp,** above, should be changed to the new parameter value.
- Only the **root** user can read the **syslog**.

When **WDC** is activated, the traces might be generated frequently. In order to avoid disturbing the MPI application performance, by default, the traces are generated only once, during the finalize phase. Use the following MCA parameter to change this behavior:

```
-mca wdc_log_at_finalize 0
```

In this situation, the collected events are aggregated per MPI process and only traced if a defined threshold (counter threshold) has been reached. Another threshold (time threshold) can be used to condition subsequent traces generation for an event that has already been traced.

Use the following **MCA** parameters to change the default thresholds values:

```
-mca wdc_oob_cnt_thresh <c_value>
-mca wdc_oob_time_thresh <t_value in secs>
```

2.7.5 KNEM Module

KNEM is a Linux kernel module enabling high-performance intra-node MPI communication for large messages. KNEM supports asynchronous and vectorial data transfers as well as offloading memory copies on to Intel I/OAT hardware.

Note http://runtime.bordeaux.inria.fr/knem/ for details.

Bullx MPI is compiled to use KNEM as soon as the RPM is loaded. To launch a job without the KNEM optimization feature, start the **mpirun** command with the following option:

```
mpirun --mca btl_sm_use_knem 0 ompi_appli
```

2.7.6 Carto: Managing Multi InfiniBand HCA Systems

The **carto** framework in **Open MPI** enables automatic InfiniBand HCA selection based on the MPI process localisation. It calculates the distance between the MPI process' socket and all the HCAs in the system, and then selects the card(s) with the shortest distance.

Base configuration

To use the **carto** framework, you need to set the following parameters:

```
carto=file
carto_file_path=[my carto file]
btl_openib_warn_default_gid_prefix=0
```

And set **my carto file** to the desired model.

Carto files Syntax

Carto files define graphs representing the hardware topology. To define the topology, two keywords are used exclusively: **EDGE** and **BRANCH_BI_DIR**. These should be sufficient to describe Bull hardware.

EDGE keyword

The **EDGE** keyword is used to describe basic elements such as processors, memory banks and network cards. The syntax is:

```
EDGE <Type> <Name>
```

Where **Type** may be *socket*, *Memory* or *InfiniBand*. **Name** is used for branch definitions. All the sockets, memory banks and InfiniBand cards to be used must have a corresponding **EDGE** definition.

BRANCH_BI_DIR Keyword

A connection in the graph can be defined between two edges with a branch. A branch definition has the following syntax:

```
BRANCH_BI_DIR <Name1> <Name2>:<Distance>
```

To enable **Open MPI** to compute all the distances required, all the edges must be connected.

Note You do not have to describe the actual topology of your hardware. Any graph equivalent to the reality, in terms of distance, should result in the same behavior in Open MPI. This can substantially shorten the graph description.

Example for bullx B505 Accelerator Blade

Here is an example for a **bullx B505 accelerator blade** with 2 InfiniBand cards, one connected to each socket:

```
EDGE socket slot0
EDGE socket slot1
EDGE Memory mem0
EDGE Memory mem1
EDGE Infiniband       mlx4_0
EDGE Infiniband       mlx4_1
BRANCH_BI_DIR   slot0     slot1:10
BRANCH_BI_DIR   slot0     mem0:10
BRANCH_BI_DIR   slot0     mlx4_0:10
BRANCH_BI_DIR   slot1     mem1:10
BRANCH_BI_DIR   slot1     mlx4_1:10
```

The resulting output is:

```
   [mem0]--10--[  slot0 ]--10--[  slot1 ]--10--[mem1]
                    |               |
                   10              10
                    |               |
                [ mlx4_0 ]      [ mlx4_1 ]
```

All socket 0 cores use the **mlx4_0** card, while all socket 1 cores will use **mlx4_1**.

Example for bullx S6030/S6010 Multi-module with Bull Coherent Switch

```
# Module 0 : 4 sockets, 1 HCA
 EDGE socket slot0
 EDGE Memory mem0
 EDGE socket slot1
 EDGE Memory mem1
 EDGE socket slot2
 EDGE Memory mem2
 EDGE socket slot3
 EDGE Memory mem3
 EDGE Infiniband       mlx4_0

 # Module 1 : 4 sockets, 1 HCA
 EDGE socket slot4
 EDGE Memory mem4
 EDGE socket slot5
 EDGE Memory mem5
 EDGE socket slot6
 EDGE Memory mem6
 EDGE socket slot7
 EDGE Memory mem7
 EDGE Infiniband       mlx4_1

 # Module 2 : 4 sockets, 1 HCA
 EDGE socket slot8
 EDGE Memory mem8
 EDGE socket slot9
 EDGE Memory mem9
 EDGE socket slot10
 EDGE Memory mem10
 EDGE socket slot11
 EDGE Memory mem11
 EDGE Infiniband       mlx4_2

 # Module 3 : 4 sockets, 1 HCA
 EDGE socket slot12
 EDGE Memory mem12
 EDGE socket slot13
 EDGE Memory mem13
 EDGE socket slot14
 EDGE Memory mem14
 EDGE socket slot15
 EDGE Memory mem15
 EDGE Infiniband       mlx4_3
```

Note This is not the real connection topology but produces the same results for **InfiniBand**.

```
 # Module 0 : memory
 BRANCH_BI_DIR   slot0     mem0:10
 BRANCH_BI_DIR   slot1     mem1:10
 BRANCH_BI_DIR   slot2     mem2:10
 BRANCH_BI_DIR   slot3     mem3:10

 # Module 0 : inter-socket
 BRANCH_BI_DIR   slot0     slot1:1
 BRANCH_BI_DIR   slot0     slot2:1
 BRANCH_BI_DIR   slot0     slot3:1

 # Module 0 : IO
 BRANCH_BI_DIR   slot0     mlx4_0:10

 # Module 0 : inter-module
 BRANCH_BI_DIR   slot0     slot4:80

 # Module 1 : memories
 BRANCH_BI_DIR   slot4     mem4:10
 BRANCH_BI_DIR   slot5     mem5:10
 BRANCH_BI_DIR   slot6     mem6:10
 BRANCH_BI_DIR   slot7     mem7:10

 # Module 1 : inter-socket
 BRANCH_BI_DIR   slot4     slot5:1
 BRANCH_BI_DIR   slot4     slot6:1
```

```
BRANCH_BI_DIR   slot4     slot7:1

# Module 1 : IO
BRANCH_BI_DIR   slot4     mlx4_1:10

# Module 1 : inter-module
BRANCH_BI_DIR   slot4     slot8:80

# Module 2 : memory
BRANCH_BI_DIR   slot8     mem8:10
BRANCH_BI_DIR   slot9     mem9:10
BRANCH_BI_DIR   slot10    mem10:10
BRANCH_BI_DIR   slot11    mem11:10

# Module 2 : inter-socket
BRANCH_BI_DIR   slot8     slot9:1
BRANCH_BI_DIR   slot8     slot10:1
BRANCH_BI_DIR   slot8     slot11:1

# Module 2 : IO
BRANCH_BI_DIR   slot8     mlx4_2:10

# Module 2 : inter-module
BRANCH_BI_DIR   slot8     slot12:80

# Module 3 : memories
BRANCH_BI_DIR   slot12    mem12:10
BRANCH_BI_DIR   slot13    mem13:10
BRANCH_BI_DIR   slot14    mem14:10
BRANCH_BI_DIR   slot15    mem15:10

# Module 3 : inter-socket
BRANCH_BI_DIR   slot12    slot13:1
BRANCH_BI_DIR   slot12    slot14:1
BRANCH_BI_DIR   slot12    slot15:1

# Module 3 : IO
BRANCH_BI_DIR   slot12    mlx4_3:10

# Module 3 : inter-module
BRANCH_BI_DIR   slot12    slot0:80
```

2.7.7 Mellanox FCA

Mellanox FCA is a component of the collective framework. It improves collectives by offloading them in the InfiniBand fabric. To use FCA users have to:

- Ask administrators to install the FCA product on the cluster and activate the FCA service on Management node.
- Use the bullxmpi **performance_application_fca.conf** profile (see *Section* 2.5.3 *Parameters Profiles*).

2.7.8 bullx MPI Hyperthreading Support

By default Bull clusters are delivered with the hyperthreading feature disabled.

When the users activate this feature on their cluster (for performances reasons), the bullxMPI binding (see *Section 2.4 Binding with bullx MPI*) options have to be modified.

OpenMPI 1.6 series does not support hyperthreading.

BullxMPI has an early support of hyperthreading by considering logical cores (also called hardware threads) as physical cores.

To have the same binding behavior on a hyperthreaded cluster than a not hyperthreaded cluster, you have to:

- add the **--cpus-per-rank 2** option to the mpirun command
- add the **--ntasks-per-node=16** option to the SLURM allocation command for a cluster with 16 cores per node.

2.7.9 bullx MPI GNU Compilers Support

The default version of bullx MPI uses Intel compilers for the **mpicc**, **mpiCC**, **mpic++**, **mpif77** and **mpif90** wrappers.

This use of Intel compilers make the bullx MPI incompatible with the GNU compilers, especially for GNU Fortran.

By installing the optional **bullxmpi_gnu** version, the bullxmpi fully supports the GNU compilers.

To use this version, you have to load the **bullxmpi_gnu** module instead of the **bullxmpi** module and recompile your MPI applications.

Note	**bullxmpi** and **bullxmpi_gnu** modules should not be loaded at the same time.

2.8 Using Accelerators in MPI Programs

bullx MPI supports the Bull accelerator hardware (such as B505).

Two types of accelerator are provided with bullx scs 4:

- **GPUs** from **NVIDIA**
- **Xeon Phi** from **Intel**.

This section describes how these two accelerators can be programmed, and provides an overview of how to use these accelerators on a bullx MPI application.

2.8.1 NVIDIA GPUs

NVIDIA provides a proprietary environment called CUDA.

See To understand the cuda environment, an extensive documentation can be found in http://developer.nvidia.com/category/zone/cuda-zone
Documentation is also available in the CUDA installation directory:
/opt/cuda/cuda_version/doc

The CUDA programming model is based on threads. With CUDA, the programmer is encouraged to create a very large number of threads (several thousand). Threads must be structured into block. The card will thus schedule these threads in free compute cores.

The Khronos Group (a consortium of companies and research groups) has defined a specification called OpenCL. OpenCL defines a programming model close to the CUDA programming model. Because the specification is open, several companies have proposed an implementation of OpenCL for their hardware.

The bullx scs 4 software supports OpenCL for NVIDIA accelerator. Thus, it is possible to write an OpenCL programs and run it on NVIDIA accelerators.

2.8.2 Xeon Phi (MIC)

Note Xeon Phi supports several execution models. In this bullx scs 4 release, we support both the **accelerator mode** (also called **offload** or **co-processor** mode) and **native mode** for the Xeon Phi. On native mode, there are MPI processes on Xeon Phi, where in accelerator mode there are only MPI processes on Xeons, which delegate part of their work to their accelerators.

The Xeon Phi accelerators are automatically configured to be fully usable in native mode: each Xeon Phi holds an IP address. In native mode, the binaries of the application must be compiled for the Xeon Phi architecture: Xeon Phi binaries are not compatible with the hosts processors. Users can compile an application for the Xeon Phi by using the **-mmic** compiler flag. To run a native application, it is not sufficient to compile the application. The application must be available on the Xeon Phi. Different options exist to add software on Xeon Phi (see the *bullx MC Administration Guide, Appendix G.4*).

The most flexible option is to copy the application on the Xeon Phi (via scp for example). The Xeon Phi must be available: see *Section 2.8.3 bullx MPI and Accelerators* for more information about this copy. Xeon Phi jobs can take long time to complete due to EPILOG/PROLOG scripts.

For more information about Xeon Phi and SLURM see *Section 2.9* in *the bullx BM User's Guide*. For more information about using bullx MPI on Xeon Phi see *Section 2.8.3. bullx MPI and Accelerators*

Although the Xeon Phis are configured for the **native mode**, they can be also used in **co-processor mode**. To use a Xeon phi as a co-processor, the programmer should use the new **Intel LEO** directives (**Language Extensions for Offload**). These directives, which exist for both Fortran and C are implemented in the Intel compilers v.13.

With LEO, some directives are designed to launch data transfers, other are designed to compile and offload code to the MIC side.

With LEO, it is possible to offload an OpenMP section: OpenMP threads will be created on the MIC side. Because the MIC can have more than 60 cores, and each core can manage four threads, OpenMP thread number must be adapted to this architecture. Nevertheless, it can be challenged to develop an OpenMP section that scale on a large numbers of cores.

Intel TBB (**Threading Building Blocks**) and **Intel Cilk Plus** can also be used in C programs (not Fortran) to program the Xeon Phi.

The description of these programming models is not in the scope of this document. Programmer must read the Intel compiler documentation.

LEO uses a low-level library called COI. COI can be used directly to offload code to the MIC. Nevertheless, this library is not well documented yet and we discourage the use of this method to offload code on the MIC.

2.8.3 bullx MPI and Accelerators

Using accelerators with MPI means that several MPI processes will use one or more accelerator during the program execution. There are some execution options to know in order to have good performance for both MIC and GPU.

2.8.3.1 GPU

On GPU hardware with several CPU socket and several GPUs (like the B505 blade), best data transfer performance is achieved when a MPI process is pinned on the CPU socket closest to the GPU used. Several CUDA functions (such as **cudaSetDevice** or **cudaChooseDevice**) can be used on the MPI program to choose the correct GPU.

2.8.3.2 Xeon Phi Native mode

bullxmpi-mic is a version of **bullxmpi** designed for jobs using Xeon Phi nodes. This means that some or all of the MPI ranks can run on Xeon Phis, while others can run on Xeons.

To use it, one needs to load **Intel** compilers (at least the 2013 version) and **bullxmpi-mic** in the user environment by executing the following commands (in this exact order):

```
source /opt/intel/composerxe/bin/compilervars.sh
module load bullxmpi-mic/bullxmpi-mic-1.2.5.1
```

To compile a MPI executable for Xeon Phi, **mpicc** (or **mpif90**) must be used with the **-mmic** option. This generates Xeon Phi executables and selects Xeon Phi versions of shared libraries discarding x86_64 ones. For example:

```
mpicc -mmic prog.c -o mic_prog
```

mic_prog is a Xeon Phi executable and must be available on the Xeon Phi before launching it via **mpirun**. In bullx scs 4, the Xeon Phis are available after the reservation (**salloc**) and after the execution of the **init_mic.sh** script within the reservation. Thus, users must copy the application (**mic_prog**, in our example) when the Xeon Phis are available. Users can also choose to add **mic_prog** on a NFS mount to avoid an explicit copy to the Xeon Phi.
For more information about NFS mount see the *bullx MC Administration Guide, Appendix G.4*.

Note A Xeon Phi is available when:
- it is allocated (via salloc)
- the **init_mic.sh** command is launched after the allocation

Attention: init_mic.sh takes ~1 min to complete.

Xeon Phis are generally seen as normal nodes:

- they have hostnames (following the pattern '**${host_nodename}-mic${X}**', with X in 0,1)
- they can be connected to by ssh
- NFS mounts can be configured by the system administrator (see *Appendix G* in the *bullx MC Administration Guide* for details)
- MPI jobs can be launched on them

However they do not have SLURM daemons running on them (see *Section 2.9 MIC configuration and usage* in the *bullx BM User's Guide*), which implies that:

- Xeon Phi nodes do not appear in the result of the **sinfo** command and cannot be allocated directly
- they are a resource of their host nodes in SLURM. The user must allocate those hosts nodes instead
- MPI jobs are launched on them only using **mpirun**, not using **srun**

Launching a job on Xeon Phis requires these 4 steps:

1. Allocating nodes with Xeon Phi coprocessors in SLURM using the **–gres** option.
2. Running **/etc/slurm/init_mic.sh** to ensure that Xeon Phis are available (~1min to complete)
3. Copy the application to the Xeon Phi (if no NFS mounts or no extra overlays are configured, see *Appendix G* in the *bullx MC Administration Guide*)
4. Running the MPI program with **mpirun** (because **srun** is not available for native Xeon Phi applications)

At the end of the allocation, the EPILOG script is launched. The Xeon Phi node can stay ~1min in completing state (see *Section 2.9 of the bullx BM User's Guide* for more details).

Note Xeon Phi nodes must be automatically allocated in exclusive mode. SLURM should be configured in this way, see *Section 2.9 of the bullx BM User's Guide* for more details.

For example, to launch a job on four Xeon Phi, with one MPI task per Xeon Phis, you can allocate two nodes with two Xeon Phis each:

```
salloc -N2 --gres=mic:2
/etc/slurm/init_mic.sh
mpirun -np 4 mic_prog
```

bullxmpi-mic selects automatically Xeon Phi nodes of a job for a **Xeon Phi** executable, and Xeon nodes for a **x86_64** executable. You can launch hybrid jobs after compiling a program into two executables (one for Xeon and one for Xeon Phi) and launching both **mpirun** using the ':' character as a separator between the commands.

Examples

Note We assume in these examples that **mic_prog** is on a NFS mount, so the step 3 (copy mic_prog to the Xeon Phis) is useless.

For example, on a cluster with two Xeon Phi per nodes, the following commands are equivalent:

```
salloc -N2 --gres=mic:2
/etc/slurm/init_mic.sh
mpirun -np 16 -bynode xeon_prog : -np 60 -bynode mic_prog
```

and:

```
salloc -N2 --gres=mic:2 -w node1,node2
/etc/slurm/init_mic.sh
mpirun -np 16 -bynode -H node1,node2 xeon_prog : -np 60 -bynode -H node1-
mic0,node1-mic1,node2-mic0,node2-mic1 mic_prog
```

Both launch 16 ranks on 2 Xeons and 60 on their 4 Xeon Phi (here -bynode picks nodes in a round robin fashion). Nodes are chosen manually in the last command, whereas in the first one mpirun guesses nodes to pick: Xeons nodes allocated by SLURM for the Xeon executable and corresponding Xeons Phis nodes for the Xeon Phi executable.

Note Xeon Phi processors have 4 hyperthreads per core (see section 2.7.8 bullx MPI Hyperthreading Support. In bullxmpi-mic, binding with "–cpu-per-rank" will use logical cpus, so in a hybrid application you will have to use the option **--cpu-per-rank $((4*N))** to bind to N physical cpus.

2.8.3.3 Xeon Phi co-processor Mode

Note For the co-processor mode, the application must be compiled for the host (not for the Xeon Phi). The application runs on the host and offload some part of the computation to the Xeon Phi. This mode is close to the GPU mode.

In this mode, Xeon Phis are used as coprocessors to a node, which means that the MPI runtime is not aware of them: the Xeon binary offloads the MIC code. Therefore the normal flavor of **Bullxmpi** has to be used, not **Bullxmpi-mic**. To use Xeon Phi in this mode it is advisable to:

1. Set the environment:

```
source /opt/intel/composerxe/bin/compilervars.sh
module load bullxmpi/bullxmpi-1.2.6.1
```

2. Compile your app containing OpenCL or offload pragmas normally. (See **/opt/intel/${composerxe}/Samples/en_US/[C++ | Fortran]/mic_samples/** for examples):

```
mpicc offload_prog.c -o offload_prog
```

3. Reserve the Xeon Phi with **--gres** and initialize them:

```
salloc -N1 --gres=mic:2
/etc/slurm/init_mic.sh
```

Note If the Xeon Phis are intended to be used only in offload mode for each node in the cluster, the system administrator can disable the PROLOG/EPILOG scripts (to remove the time needed to reboot the Xeon Phis). See *section 2.9.4 Managing security of the Xeon Phis* in the *bullx BM User's Guide*, for more details about the impacts of disabling PROLOG/EPILOG.

4. Launch your job with **srun** or **mpirun**:

```
mpirun -N 2 offload_prog
```

There are two executions issues with the Xeon Phi co-processor.

- The first issue is the affinity problem, which is basically the same than for GPU hardware. LEO provides ways to set up the correct GPU for a particular MPI task.
- The second issue is the thread affinity on the MIC. With the Intel OpenMP software stack, it is possible to control the number of threads created by each MPI task on the MIC (with the **MIC_OMP_NUM_THREADS** environment variable), or to control thread affinity (**MIC_KMP_AFFINITY**). These variables are defined in the Intel Compilers documentation.

  The user can specify thread affinity for each MPI process launched through environment variables. See Intel Documentation for Xeon Phi for more details.

Usage example

```
mpirun -x MIC_PREFIX=MIC -x MIC_OMP_NUM_THREADS=48 -x
MIC_KMP_AFFINITY=exeplicit,proclist=[4-51],verbose ./mpi_omp_mic_test
```

When launching several executables with one **mpirun** command, it can be useful to control where OpenMP threads will be bound.
See the standard documentation provided with the Intel compilers for more details about the runtime OpenMP provided by the Intel compilers (especially the KMP_AFFINITY syntax).

Bull Cedoc
357 avenue Patton
BP 20845
49008 Angers Cedex 01
FRANCE