FlexMPI

Providing elastic and monitoring capabilites to MPI applications

USER MANUAL

Version 3.1

Avda. de La Universidad 30. 28911 Leganes. Spain.

Maintained by David E. Singh

Last edition: 23 October 2018

Contents

Introduction

FLEX-MPI is a runtime system that extends the functionalities of the MPI library by providing dynamic load balancing and performance-aware malleability capabilities to MPI applications. Dynamic load balancing allows FLEX-MPI to dynamically adapt the application workload assignments according to the computing node performance. Performance-aware malleability permits to change the application's number of processes at runtime.

Project goals. The main goals of this project are to provide the following capabilities to MPI applications in a transparent way, without user intervention:

- To automatically create or remove new processes and redistribute the data among the existing ones.
- To perform run-time monitoring of the parallel application by means of performance counters.
- To apply different polices that can be used in combination with the malleable capabilities to automatically adjust the application performance according to different criteria.
- To control the process spawn or removal by means of external control commands, produced by the user or third-party applications (like the system scheduler).

How to cite FlexMPI?. If you are using FlexMPI and you want to cite it, please, cite the following reference: Gonzalo Martin, David E. Singh, Maria-Cristina Marinescu and Jesus Carretero. Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. Parallel Computing. Vol. 46, No. 0. Pages: 60-77. 2015.

Components

FlexMPI consists of different components: the Monitoring module uses both PAPI and RAPL interfaces to dynamically collect performance metrics from the MPI application. The approach relies on low-level PAPI interface to collect hardware events like the MIPS or FLOPS. In addition, RAPL interface is used to collect the energy consumption of each processor. The Dynamic Process Management performs the runtime addition and removal of MPI processes, as well as the inter-process communication whenever a reconfiguring action is carried out. The Load Balancing Module balances the workload. The Data Redistribution Module transparently transfers data between MPI processes when the workload is being balanced or when processes are being created or removed. The Computational Prediction Model and Power Prediction Model calculate, respectively, the application performance and energy consumption for new processor configurations that are being evaluated. Finally, the Reconfiguring Policy Model evaluates whether the application satisfies the user-given objectives. If so, it continues executing on the same processor configuration. Otherwise, it performs a reconfiguring action by adding or removing processes. A more detailed description can be found in [MSMC15].

The current distribution includes the FlexMPI library and several support programs that can be used for testing purposes. These components are:

- The FlexMPI runtime, located in `src`, `include`, and `lib` directories.
- Several configuration files, located in `configuration_files` directory, that describe the compute node characteristics of the platform. These files are internally used by FlexMPI.
- A use case of application integrated with FlexMPI, located in `examples` directory, that performs the Jacobi iterative method. This application can be used for evaluating the FlexMPI capabilities.
- Several execution scripts, located in `scripts` directory, for executing the Jacobi code under different conditions.
- An example of external controller, located in `controller` directory, that communicates with the FlexMPI application, sending configuration commands and receiving monitoring data.

Installation

This section explains how to install FlexMPI as well all the required software necessary for the program execution. Currently, FlexMPI has only been tested on Linux platforms. Support for other operating systems is not guaranteed.

Prerequisite software installation

FlexMPI is written in C and uses different libraries to implement some of the runtime's functionalities. These components have to be installed in the platform before installing FlexMPI. The following list shows the prerequisite software:

- GNU gcc compiler. Compiles the program and libraries and generate the executable. Supported version 4.9 or above.
- GNU Make. Controls the generation of FlexMPI executable from the program's source files and libraries. Supported version 3.81 or above. For installation: `sudo apt-get install make`
- MPI library. Provides support to communicate and synchronize the processes involved in the simulation execution. FlexMPI have been tested with MPICH version 3.2 or above. For installation: `sudo apt-get install mpich`
- GLPK, the GNU Linear Programming Kit. This library is used to peform automatic performance optimizations. Supported version 4.47 or above. For installation: `sudo apt-get install glpk-utils libglpk-dev`
- PAPI library [MBDH99]. It is used for accessing to the performance counters. Supported version 5.5.1.0 or above. For installation: `sudo apt-get install papi-tools libpapi-dev`
- Compute nodes have to have a public-key authentication to connect between them in a transparent way. More information in `https://kb.iu.edu/d/aews`

FLEXMPI INSTALLATION

This section describes how to install and compile FlexMPI. Once all the required software is properly installed, the installation process basically consists of two steps: (1) to extract the source files from the downloaded installation file and (2) to properly compile these files and generate the library and executables.

To complete the first step, you have to download the tarball file from the gitlab repository. Using a web browser, go to the following url:

`https://gitlab.arcos.inf.uc3m.es:8380/desingh/FlexMPI`

If you are a new user, you need to register first. Then, download the tar.gz file.

Then, you only have extract the runtime's files from the downloaded tarball file and rename the directory. In this document, we assume that the extracted files are installed in the directory `FlexMPI`. The following listing shows an example for extracting the source files. Note that xxx is the hash key of the tar file. Note: **For the current configuration of the scripts, FlexMPI should be installed in the user's home directory.**

```
# Extract the tarball
username@hostname:~/FlexMPI$  tar -zxvf FlexMPI-master-xxx.tar.gz
# Rename the directory
username@hostname:~$  mv FlexMPI-master-xxx     FlexMPI
```

The next step is the Makefile configuration. There are two Makefiles that should be edited. They are located in:

- username@hostname:~/FlexMPI/Makefile
- username@hostname:~/FlexMPI/examples/Makefile

For each one of these files, edit the first line and change the `HOME` variable value, replacing the original one of `/home/desingh`) by the current user home path.

We assume that all the required software libraries have been installed in the local user LIB directory located in the user's home directory (`~/LIBS`). For a different library path, it is necessary to edit these two Makefile files and include the new paths to the related libraries.

The next step consists of generating the different executables that includes the FlexMPI library, the application examples and the external controllers and tools. The following listing shows the sequence of commands that have to be performed. The listing also includes how to provide execution permissions to the scripts and how to add to the `LD_LIBRARY_PATH` the library paths of FlexMPI, MPICH, PAPI and GLPK.

```
# Rename the Makefile.bak files as Makefile:
username@hostname:~/FlexMPI$  mv Makefile.bak Makefile
username@hostname:~/FlexMPI/controller$  mv Makefile.bak Makefile
username@hostname:~/FlexMPI/examples$  mv Makefile.bak Makefile
# Compile the FlexMPI runtime
username@hostname:~/FlexMPI$  make
# Compile the example programs
username@hostname:~/FlexMPI/examples$  make
# Compile the controller demonstrator
username@hostname:~/FlexMPI/controller$  make
# Provide permissions to the execution scripts (only once)
username@hostname:~/FlexMPI/scripts$  chmod 755 *.sh
username@hostname:~/FlexMPI/run$  chmod 755 ./ExecutionScript.sh
# Environment variables (only once): assuming that the requited libraries are installed in $HOME/LIBS
username@hostname:~/FlexMPI$ export LD_LIBRARY_PATH=$HOME/LIBS/glpk/lib/:$HOME/FlexMPI/lib/
:$HOME/LIBS/mpich/lib/:$HOME/LIBS/papi/lib/:$LD_LIBRARY_PATH
```

FlexMPI execution

This chapter describes how to execute FlexMPI. The current distribution includes a use case that performs the Jacobi iterative method integrated with FlexMPI. This section provides three examples for running this code. The first one is the simplest example with a stand-alone code. The second one includes a coordinated execution of the code with an external controller. The last one is an example of execution in a multi-node platform.

RUNNING A SIMPLE EXAMPLE

This section shows how to run the example program in one compute node and how to communicate with it by means of the `nping` command. Two different command prompts are needed to run this example. The first one is used to execute the Jacobi application by means of the following command:

```
username@hostname:~/FlexMPI/scripts$  ./Execute1.sh 2 6666 6667 1
```

Where *Execute1.sh* is the execution script that runs the application. The first argument (2) is the initial number of processes, the second argument (6666) is the FlexMPI listening port for receiving commands, the third argument (6667) is the controller's listening port to receive the monitoring data (not applicable in this example because the controller is not used) and the last argument (1) is the application internal id.

The program output should be like the following one. Note that `compute-1` is the name of the compute node that is running the program (in your execution the name of this compute node should be different).

```
username@hostname:~/FlexMPI/scripts$  ./Execute1.sh 2 6666 6667 1
Host compute-1 maxprocs is 12. ID is 0

[DEBUG] Initializing attribute successfully

[DEBUG] Setting detached state successfully

[DEBUG] Creating thread successfully
[1] Process spawned in compute-1 | Data loaded in 0.003427 secs.
[0] Process spawned in compute-1 | Data loaded in 0.006211 secs.
[0] Configuration: dim: 500 itmax: 10000 diff_tol: 0.000010 cpu_intensity: 100 com_intensity: 1 IO_intensity: 70

Policy is EMPI_Monitor_malleability_triggered
Iter: 100 FLOPs: 36709648321 MFLOPS:: 5046.735101 RTIME:: 7.273940 CTIME:: 0.016031 IOTime:: 0.000000 Size: 2
Policy is EMPI_Monitor_malleability_triggered
Iter: 200 FLOPs: 36302258290 MFLOPS:: 5041.587494 RTIME:: 7.200561 CTIME:: 0.000592 IOTime:: 0.000000 Size: 2
Policy is EMPI_Monitor_malleability_triggered
Iter: 300 FLOPs: 36302258270 MFLOPS:: 5041.247934 RTIME:: 7.201046 CTIME:: 0.000671 IOTime:: 0.000000 Size: 2
Policy is EMPI_Monitor_malleability_triggered
Iter: 400 FLOPs: 36302257996 MFLOPS:: 5041.667974 RTIME:: 7.200446 CTIME:: 0.000568 IOTime:: 0.000000 Size: 2
Policy is EMPI_Monitor_malleability_triggered
Iter: 500 FLOPs: 36302258291 MFLOPS:: 5041.320745 RTIME:: 7.200942 CTIME:: 0.000592 IOTime:: 0.000000 Size: 2
```

Jacobi is an iterative parallel application that is compute intensive (although is also has communications). The current version does not perform I/O. The application output shows, for each 100 iterations the following data: the iteration number (Iter), the absolute number of FLOPs (floating point operations) in the sampling interval (100 iterations) for the root process, the number of MFLOPS (floating point operations per second) for the root process, the user, communication and IO times of the sampling interval and the current number of processes (size value).

The second command prompt is used to send control commands to the application by means of `nping` command. For example, for creating two new processes, you should type the following command **replacing `compute-1` by the name of the compute node that you are using**[1].

```
username@hostname:~/FlexMPI/scripts$ nping --udp -p 6666 -c 1 compute-1 --data-string "6:lhost:2"
```

This command sends to the FlexMPI's listening port 6666 in `compute-1` node the command `"6:lhost:2"`. Command 6 performs a process spawn/destruction. `lhost` is the host name in which the processes are created/destroyed. `lhost` is an alias of the current compute node, you can see it specified in file corefile in `FlexMPI/configuration_files/corefiles` directory. Finally, the last number is the number of processes modified. If the value is positive, new processes are spawned, a neg-

[1] You can obtain this name by looking at the application output ("Process spawned in...") or executing the Linux command uname -n

ative number destroys existing processes in this compute node. **Note that it is not possible to destroy more processes than the initial number, 2 for this example.** The complete list of commands can be seen in Section *FlexMPI commands* . The effect of this command can be seen in Jacobi program output.

```
....
Iter: 500 FLOPs: 36302258291 MFLOPS:: 5041.320745 RTIME:: 7.200942 CTIME:: 0.000592 IOTime:: 0.000000 Size: 2
Received packet from 10.0.40.12:52278 Data: 6:lhost:2

Command number is 6
Command: Create 2 processes in compute node: lhost
Sent 0 bytes as response
Policy is EMPI_Monitor_malleability_triggered
Spawn 2 at 43.300992
Iter: 600 FLOPs: 36302257986 MFLOPS:: 5041.678475 RTIME:: 7.200431 CTIME:: 0.000573 IOTime:: 0.000000 Size: 4
[2] Process spawned in compute-1 at 601 | Data received in 0.001189 secs.
[3] Process spawned in compute-1 at 601 | Data received in 0.001174 secs.
Policy is EMPI_Monitor_malleability_triggered
Iter: 700 FLOPs: 18272037517 MFLOPS:: 5119.567235 RTIME:: 3.569059 CTIME:: 0.074321 IOTime:: 0.000000 Size: 4
Policy is EMPI_Monitor_malleability_triggered
Iter: 800 FLOPs: 18214026665 MFLOPS:: 5104.331688 RTIME:: 3.568347 CTIME:: 0.061192 IOTime:: 0.000000 Size: 4
Policy is EMPI_Monitor_malleability_triggered
Iter: 900 FLOPs: 18213373846 MFLOPS:: 5103.804038 RTIME:: 3.568588 CTIME:: 0.060838 IOTime:: 0.000000 Size: 4
Policy is EMPI_Monitor_malleability_triggered
Iter: 1000 FLOPs: 18212326433 MFLOPS:: 5103.949614 RTIME:: 3.568281 CTIME:: 0.060708 IOTime:: 0.000000 Size: 4
```

We can observe that the number of processes is 4 after executing the command. The number of FLOPs decreases because a fraction of the root process workload has been transferred to the new spawned processes. In case of running the program in a multicore node with at least four cores, the sample interval execution time (RTIME) should be smaller than the previous configuration with two processes. Next, we can send the command to remove one of the spawned processes by executing:

```
username@hostname:~/FlexMPI/scripts$ nping --udp -p 6666 -c 1 compute-4 --data-string "6:lhost:-1"
```

The application output should be like:

```
....
Iter: 1000 FLOPs: 18212326433 MFLOPS:: 5103.949614 RTIME:: 3.568281 CTIME:: 0.060708 IOTime:: 0.000000 Size: 4
Received packet from 10.0.40.12:37057 Data: 6:lhost:-1

Command number is 6
Command: Create -1 processes in compute node: lhost
Sent 0 bytes as response
Policy is EMPI_Monitor_malleability_triggered
Remove 1 at 61.685445
Process [3] removed from compute-1
Iter: 1100 FLOPs: 18407621625 MFLOPS:: 5116.241127 RTIME:: 3.597880 CTIME:: 0.061216 IOTime:: 0.000000 Size: 3
Policy is EMPI_Monitor_malleability_triggered
Iter: 1200 FLOPs: 24929281854 MFLOPS:: 5189.151770 RTIME:: 4.804115 CTIME:: 0.215792 IOTime:: 0.000000 Size: 3
Policy is EMPI_Monitor_malleability_triggered
Iter: 1300 FLOPs: 25095791927 MFLOPS:: 5251.143398 RTIME:: 4.779110 CTIME:: 0.230193 IOTime:: 0.000000 Size: 3
```

Running FlexMPI with an external controller

We have developed an external program called controller that permits to execute different applications in a centralized way. In addition, the controller communicates with each application (by means of FlexMPI), allowing to send commands to the applications and receive monitoring information from them. Before executing the controller, it is necessary to complete a configuration stage that is described next.

CONTROLLER CONFIGURATION

There are two files that have to be configured by the user. These files are:

- The file username@hostname:~/FlexMPI/run/nodefile.dat contains the list of compute nodes in the format node_name:num_cores:node_alias. The name of the compute node is the compute node address used in a ssh connection to the node.

  For instance, in the following listing there are two compute nodes (we use short node names): compute1 with 4 cores and compute2 with 8. The names and aliases are the same for both. **The user has to edit this file and include the compute nodes that are going to be used.**

```
username@hostname:~/FlexMPI/run$  cat nodefile.dat
compute1:4:compute1
compute2:8:compute2
```

- The file username@hostname:~/FlexMPI/run/appfile.dat contains the number of applications (one per line). The application corresponds to Jacobi kernel that performs a matrix-vector multiplication, an collective MPI communication operation and a MPI parallel I/O operation. **The last one (I/O operation) is only performed every 100 iterations.** This kernel can be configured by several input parameters defined in this file. The values of each line corresponds to the values of these configuration parameters that are described next:

  - Number of processors: the number of processors that the application originally executes
  - Matrix size: the size of the input matrix. It is a square dense matrix with values automatically generated by the program.

- CPU intensity: the number of times that the matrix-vector is repeated per iteration. This parameter allows to increase the weight of the CPU part of the program.
- Communication intensity: the number of times that the `MPI_Allgatherv()` collective communication is repeated per iteration. This parameter allows to increase the weight of the communication part of the program.
- I/O intensity: the number of times that the MPI I/O operation is repeated per iteration. This parameter allows to increase the weight of the I/O part of the program.
- I/O action: allows to perform actual MPI I/O or dummy I/O. If this parameter is -1, the `MPI_File_write_all()` is executed in the I/O phase. Otherwise, if this parameter is greater or equal than 0, a dummy I/O is performed, sleeping the processes as many seconds as the parameter value. Note that I/O action is a floating-point parameter. The idea of the dummy I/O is to do not stress the I/O subsystem during debugging development. Also note that the switch from MPI I/O to dummy I/O is automatically done by FlexMPI, thus the user does not have to modify the source code (where the `MPI_File_write_all()` is called in both cases).
- Number of iterations: number of iterations that the program executes.

The following listing show two applications: the first one is executed with 2 processes with a matrix size of 5000x5000 entries. In each loop iteration, it performs the matrix-vector multiplications two times, the communication operation once and the I/O once with a dummy I/O time of 2.5 seconds. The program finished after completing 2000 iterations.

The second one is executed with 4 processes with a matrix size of 7000x7000. The CPU, communication and I/O intensities are 1. It performs actual I/O (by calling `MPI_File_write_all()`) and completes the execution after 3000 iterations.

```
username@hostname:~/FlexMPI/run$  cat appfile.dat
# Example of Application File
# Each uncommented line corresponds to an application
# Format: num_processes:matrix_size:CPU_intensity:communication_intensity:
IO_intensity:IO_action:Num_iterations
# If IO_action<0, the program performs MPI I/O calls
# If IO_action>=0, the program sleeps IO_action during the I/O call

#Application 1
2:5000:2:1:1:2.5:2000

#Application 2
4:7000:1:1:1:-1:3000
```

In the next configuration step the nodefile.dat file is parsed by the ExecutionScript.sh script and the following two output files are generated: nodefile1.dat (used by the workloadgen program, depicted below) and nodefile2.dat (used by FlexMPI and the external controller).

```
username@hostname:~/FlexMPI/run$  ./ExecutionScript.sh nodefile.dat
```

In the last configuration step the workloadgen program reads the nodefile1.dat and appfile.dat configuration files and generates the workload.dat file that is used for the controller.

```
username@hostname:~/FlexMPI/controller$  ./workloadgen  ../run/nodefile1.dat
../run/appfile.dat
```

Note that the workloadgen has two optional parameters that by default are not activated: `-differentnodes` for executing each application in a different compute node and `-noexcl` for executing the controller in the same compute nodes as the applications. We discourage the use of the second option because the controller is a multithreaded application and produces performance interferences with the running applications. Because of that, it is better to run it in a separated compute node.

The program output is the `workload.dat` file, located in `FlexMPI/controller` directory. This file has one line per application that is being executed (each application is independent). In each line, the first string is the application name (jio corresponds to the Jacobi code) then, it includes the matrix size and a list of duplets `{node_name,n_procs}`, where `node_name` is the name of the compute node and `n_procs` is the initial number of processes executed in the system. For instance, the following listing shows the file contents for the previous example. It corresponds to two Jacobi applications are executed in the node `compute2`. The first application is configured to run one process and the second one four. Note that `compute1` node is reserved for executing the controller.

```
jio:5000:2:1:1:2.500000:2000:compute2:2
jio:7000:1:1:1:-1.000000:3000:compute2:4
```

CONTROLLER EXECUTION

To execute the application controller, type the following command in a command prompt.

```
username@hostname:~/FlexMPI/controller$  ./controller
```

Using the previous workload file, it will automatically execute two Jacobi applications. The applications' output is located in the `FlexMPI/controller/logs` directory. The controller produces in the `FlexMPI/controller/execscripts` directory as many execution scripts as applications. Each execution script runs the Jacobi code with the required configuration. Note that these scripts are automatically created and executed by the controller. The controller's output is the following one:

```
************************************************************
FlexMPI program controller 2.05
************************************************************

— Initializing

### Application maleability enabled

Reading the workload ...
@@@@ App [0], root node: localhost
Creating the rankfile 1 ...
Creating the execution script 1 ...
Executing the application 1 ...
@@@@ App [1], root node: localhost
Creating the rankfile 2 ...
Creating the execution script 2 ...
Executing the application 2 ...

— Creating the listerner threads
Initializing attribute successfully
Setting detached state successfully
Creating listener thread successfully
Initializing attribute successfully
Setting detached state successfully
Creating listener thread successfully

— Displaying the application workload
Application 0, Port1 (listener): 6666 Port2 (Sender): 6667
localhost 4
Application 1, Port1 (listener): 6668 Port2 (Sender): 6669
localhost 2
Please type a command [appId command]:

— Waiting for new input commands

0 4:on
Message: 4:on. Size: 500 bytes sent to app1.
0 data from 127.0.0.1:33418 –> [compute-1] rtime 0 ptime 0 ctime 0.000000 Mflops 1 PAPI_SP_OPS 0
PAPI_TOT_CYC 0 Ratio=-nan iotime 0.000000 size 4

1 4:on
Message: 4:on. Size: 500 bytes sent to app2.
1 data from 127.0.0.1:33233 –> [compute-1] rtime 0 ptime 0 ctime 0.000000 Mflops 1 PAPI_SP_OPS 0
PAPI_TOT_CYC 0 Ratio=-nan iotime 0.000000 size 2


0 6:lhost:2
Message: 6:lhost:2. Size: 500 bytes sent to app1.
0 data from 127.0.0.1:33418 –> [compute-1] rtime 3597220 ptime 3602790 ctime 0.005926 Mflops 20196
PAPI_SP_OPS 0 PAPI_TOT_CYC 24414653641 Ratio=inf iotime 0.000000 size 4

1 data from 127.0.0.1:33233 –> [compute-1] rtime 7199452 ptime 7200341 ctime 0.001092 Mflops
10087 PAPI_SP_OPS 0 PAPI_TOT_CYC 24398573397 Ratio=inf iotime 0.000000 size 2

0 data from 127.0.0.1:33418 –> [compute-1] rtime 2399344 ptime 2463459 ctime 0.064228 Mflops
30845 PAPI_SP_OPS 0 PAPI_TOT_CYC 24975784510 Ratio=inf iotime 0.000000 size 6
```

To communicate with each application, you need to type FlexMPI commands using the following syntax: `"app_id command"`, where app_id is the application id (the first one in the workload file has id 0, the second one has id 1) and command is the FlexMPI command. For instance, the following example uses the commands `"0 4:on"` and `"1 4:on"` to activate the monitoring component of FlexMPI for each application. A new thread is created by FlexMPI in the application, and periodically collects and sends to the controller different metrics of monitoring data. On the controller, another thread is created (one per application that has the monitoring activated). This thread collects and displays the data sent by FlexMPI.

Finally, in this example, command `"0 6:lhost:2"` is used for creating two new processes in the first application. You can observe the last line of the monitoring data that the application size is increased. In the applications' output (`FlexMPI/controller/logs` directory) the application log file shows the effect of the process creation:

```
...
Iter: 480 FLOPs: 14504381078 MFLOPS:: 5019.654170 RTIME:: 2.889518 PTIME:: 2.890307 CTIME::
0.000735 IOTime:: 0.000000 Size: 2
Received packet from 10.0.40.15:41214 Data: 6:lhost:2

Command number is 6
Command: Create 2 processes in compute node: lhost
Sent 0 bytes as response
Spawn 2 at 5.406840
Host list: lhost:2

Spawn cost:: process_creation= 1.234492 data_redistribution= 0.118839 LoadBalance_computation=
0.000002
[2] Process spawned in lhost at 501 | Data received in 0.118946 secs.
[2] Jacobi started
[3] Process spawned in lhost at 501 | Data received in 0.118909 secs.
[3] Jacobi started

Iter: 500 FLOPs: 21113181613 MFLOPS:: 3906.345034 RTIME:: 5.404843 PTIME:: 5.394251
CTIME:: 0.001105 IOTime:: 2.510180 Size: 4
Iter: 520 FLOPs: 7259174452 MFLOPS:: 4995.303091 RTIME:: 1.453200 PTIME:: 1.456053
CTIME:: 0.003026 IOTime:: 0.000000 Size: 4
...
```

CONTROLLER INPUT OPTIONS

The controller includes different input options that allow to configure the runtime environment. The current supported options are:

- `-scheduling_io`, activates the I/O scheduling technique. With this option, the central controller coordinates all the I/O disk accesses of the running applications by means of a publish/subscribe technique similar to the one implemented in

CLARISSE [ICR16]. In this way, every time that an application executes an I/O action (both real and dummy ones) it requests the I/O access to the controller, which grants the access to the I/O resources only if no other application is performing the I/O. In this way, this scheduling permits the application to perform exclusive I/O access to the filesystem. The drawback of this approach is the introduction of I/O delays when two applications are performing the I/O at the same time and one of them is waiting for the completion of the other's I/O.

- `-debug`, activates the debug mode of the controller. In this mode a more detailed trace of events related to the runtime execution is displayed.
- `-monitoring`, activates the monitoring of the executing applications. This option is equivalent to sending the control signal (see next section) `"4:on"` to all the running applications.
- `-noexecute`, reads the configuration files and generates all the execution scripts but does not execute any applications. It is used for verifying that the application set out is correctly done.

CONSIDERATIONS REGARDING THE BENCHMARK PERFORMANCE

Jacobi benchmark can be tuned in order to adjust its behavior and performance. This benchmark alternates CPU, communication and I/O phases. This section summarizes different configurations of the benchmark for creating different execution scenarios.

- How to run the code without I/O. Set I/O intensity to 0.
- How to set the application CPU time. There are three parameters related to this value: the matrix size, the CPU intensity, and the number of processes.
- How to set the duration of the CPU and I/O phases. The duration of the I/O phase is related to the matrix size and I/O intensity parameter. Alternatively, if the I/O access is not an issue, it is possible to set an I/O action value greater than 0, that will delay the I/O phase the time (in seconds) given by this parameter. Note that in this case the code is not performing I/O but sleeping a certain amount of time during the MPI I/O function call.
- How to configure a code with poor scalability. Increase the communication intensity parameter. The communication phase scales worser than the CPU and I/O phases.
- How to configure a code with a good scalability. Set the communication intensity to 0. Additionally, the I/O intensity could be set to 0 and matrix sizes could be increased.

FlexMPI commands

Table 1 shows the list of FlexMPI input commands. These commands are also summarized next:

- Command 1 allows to setup malleability policies based on different optimization criteria. This option requires of tuning several internal FlexMPI parameters that are not described in the current version of this manual.
- Command 2 performs a load balancing action at the end of the sampling interval. This action considers both the computational power of the existing processors and the performance of each application process. The load balancing involves redistributing the partitioned application arrays and vectors. This is done automatically, in a user-transparent fashion.
- Command 3 is designed for testing purposes. It displays on the application's output, several performance values.
- Command 4 activates/deactivates the FlexMPI monitoring service. It requires of a external application (like the controller) that is listening to the receiver port, and receives the FlexMPI's messages. In the default configuration, the controller has to be executed in the same node as the one running the application root process. However, it is possible to specify a generic controller node by changing the FlexMPI configuration.
- Command 5 terminates the application. This is done by FlexMPI in a asynchronous manner.
- Command 6 is used to spawn or remove application processes. The syntax is a sequence of valid compute nodes with the increment of processes. Positive and negative values correspond to spawned and removed processes, respectively. It is not possible to combine in the same command different signs of values. That is, for each command, all the values have to be either positive or negative. Compute nodes with zero increment of processes can be included in the list or skipped from it.
- Command 7 permits to change the performance counter names. FlexMPI monitors two user-defined performance counters. Command seven permits These names are the ones used by PAPI library. For instance, command `"7:PAPI_STL_ICY:PAPI_TOT_INS:"` sets the first counter to *Cycles with no instruction issue* and the second one to *Number of instructions completed*. You need to make sure that the compute node architecture supports the specified counters [MBDH99].
- Command 8 binds the application processes to the compute cores. This is done by providing a sequence of process numbers (that corresponds to the rank value) and

the core id (related to the compute node where the application is running). For instance, for a 8-process application running in a single multicore node, the following sequence: `"8:0:0:1:0:2:0:3:0:4:1:5:1:6:1:7:1"`, binds the first four processes to the first core (core 0) and the remaining four to the second core (core 1).

Table 1: FlexMPI command list.

Command No.	Description	Syntax
1	Change the malleability policy	–
2	Performs load balancing	2:
3	Displays the performance values	3:
4	Turns on/off the monitoring service	4:on: or 4:off:
5	Terminates the application	5:
6	Spawns/removes processes	6:node1:p1:node2:p2:....:noden:pn
7	Changes the monitoring performance counters	7:perf_counter1:perf_counter2:
8	Performs core binding	8:p1:c1:p2:c2:...:pn:cn

Bibliography

[ICR16] F. Isaila, J. Carretero, and R. Ross. CLARISSE: A middleware for data-staging coordination and control on large-scale HPC platforms. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 346–355, May 2016.

[MBDH99] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A Portable Interface to Hardware Performance Counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[MSMC15] Gonzalo Martín, David E. Singh, Maria-Cristina Marinescu, and Jesús Carretero. Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. *Parallel Computing*, 46(0):60 – 77, 2015.