

Docker cluster and FlexMPI tutorial

Alberto Cascajo
University Carlos III of Madrid
Computer Science and Engineering Department

June 22, 2022

Contents

| | | |
|----------|--|----------|
| 1 | Slurm docker cluster | 3 |
| 2 | Step-by-step FlexMPI-Jacobi I/O execution | 5 |
| 2.1 | Installation dependencies for the prototype docker cluster | 5 |
| 2.2 | Installation of the docker cluster | 5 |
| 2.3 | Compilation of FlexMPI and the examples | 6 |
| 2.4 | Execution of the Jacobi application | 6 |
| 2.5 | Details and constraints | 6 |
| 2.6 | Execution examples | 7 |

Chapter 1

Slurm docker cluster

This Docker cluster includes all the dependencies required for its correct execution. The main characteristics of this docker cluster are as following:

- Each cluster node is created as a docker container. All cluster nodes uses the same docker image.
- The docker image is created on the fly using a docker configuration file. This image starts from a minimal Ubuntu 20.04 docker image. Also, most of the installed packets are installed from an Ubuntu 20.04 distribution.
- The cluster itself is created as a docker compose containers pack. The docker compose configuration file is created on the fly from a template, in order to modify the number of nodes of the cluster.
- The home directory of the cluster accounts is shared among the nodes by using the docker volumes.

The slurm docker cluster have the following frameworks installed:

- gcc 9.4.0
- libfabric 1.12.1
- mercury 2.0.1
- argobots 1.1
- json-c 0.15
- margo 0.9.5
- redis 6.2
- hiredis 1.0.2
- mpich 3.3.2
- slurm 21.08.6

The cluster has configured three different accounts. The password of each account is the same as their login. All the accounts are included in the sudo group. Also, all share their home directory among the nodes. Finally, all the accounts have configured a remote ssh login without a password. The differences among the accounts are the following:

- The **admin** account includes FlexMPI inside its home directory. The admin account also acts as an Operator for the malleable account on the Slurm framework.

- The **user** account has an empty home directory. This account also acts as a normal user for the static account on the Slurm framework.
- The **coord** account has an empty home directory. This account also acts as a coordinator for the static account on the Slurm framework.

The slurm cluster is built, configured and launched using a bash shell script including all the commands to build the docker cluster. The script name and usage is the following

```
launch-slurm-cluster.sh [-n <number of nodes>] [-c <cores per node>]  
                        [-b <compose file basename>] [-d]
```

The script options are the following:

- **-n**: Set the number of nodes for the cluster (default: 1).
- **-c**: Set the number of cores/tasks per node (default: 1).
- **-b**: Set the base name of the docker compose configuration file (default: **docker-compose-slurm**)
.
- **-d**: Erase unused images (default: no).

The docker compose configuration file is created starting with the content of the header file and copying the content of the node template file for each one of the nodes set on the script option. The name of the header and the node template file is equal to the basename selected with the **.header** and **.template** suffix. Also the number of cores/tasks per node is used to configure the Slurm framework to admit this number of tasks slots per node.

Once the docker cluster is launched a bash shell can be open on one the cluster nodes using the following command:

```
sh> docker exec -it minicluster-node-<NUM_NODE>-1 /bin/bash
```

Where **<NUM_NODE>** refers to the number of the node on the docker cluster that we want to connect to.

Chapter 2

Step-by-step FlexMPI-Jacobi I/O execution

2.1 Installation dependencies for the prototype docker cluster

Installing the docker cluster for the prototype requires a system with the following characteristics:

- A modern Linux operating system (either as a native OS or as a virtual machine (Ex. Microsoft WSL)).
- A modern version of docker and docker compose framework.

2.2 Installation of the docker cluster

The steps to perform the installation are the following:

1. Decompress the distribution file for the docker cluster (`miniclustertgz`).

```
sh> tar zxvf miniclustertgz
```

2. Execute the script to build the docker cluster.

```
sh> cd miniclustertgz
sh> ./launch-slurm-cluster.sh -n 3 -c 3 -d
```

The script options are the following:

- `-n`: Set the number of nodes for the cluster.
- `-c`: Set the number of cores/tasks per node.
- `-d`: Erase unused images.

NOTE: After the cluster is launched the `init` script will take a few minutes to finish the installation

3. Launch shell consoles to access the cluster nodes (each node is named `miniclustertgz_node.<NODE_NUM>_1`. where `<NODE_NUM>`. is the number of the node to connect to).

```
sh> docker exec -it miniclustertgz_node-1-1 /bin/bash
```

2.3 Compilation of FlexMPI and the examples

The steps to perform the compilation of the examples are the following:

1. Launch a shell console to access a cluster node.

```
sh> docker exec -it minicluster-node-1-1 /bin/bash
```

2. Go to the FlexMPI directory and build the library

```
sh> cd /home/admin/shared/FlexMPI-master
sh> make
```

3. Go to the examples directory and built them

```
sh> cd /home/admin/shared/FlexMPI-master/examples
sh> make
```

2.4 Execution of the Jacobi application

Start a session in the cluster:

1. Launch a shell console to access the first cluster node.

```
sh> docker exec -it minicluster-node-1-1 /bin/bash
```


2. Launch the example using its script.

```
sh> cd /FlexMPI/
sh> ./Jacobi
```

2.5 Details and constraints

Regarding the compilation of FlexMPI and the examples, there are two alternatives to compile them: using local libraries or installing the dependencies. In the root of the FlexMPI directory there is a Makefile. Please, select the variables that corresponds to your case and compile using "make". The same should be done in the Makefile of the examples.

Files related to the configuration and deployment:

- controller/rankfiles/rankfile: contains the nodenames and the max number of processes that a node can execute
- 
- run/nodefile.dat: defines the nodenames, the number of cores of the node, alias, etc.

Execution details about Jacobi and FlexMPI. To easily test the malleability features, we suggest to use 'nping'

1. In the root of the FlexMPI directory, "Jacobi" contains the parameters to run the application. The default parameters are:

```
sh> ./Lanza_Jacobi_IO.sh 4 7668 7669 1 21000 0 4 0 -1.000000 500
```

2. Parameters:

- 4: num of processes
- 7668, 7669: socket ports. 7668 for clients, 7669 for controller
- 1: id
- 21000: size of the matrix to compute
- 0: I/O level
- 4: CPU level
- 0: Communication level
- -1.00000:
- 500: number of iterations

3. In order to test the malleability, nping can be executed using the following command line (it can be done in another node):

```
sh> nping --udp -p <client_port> -c 1 <controller_node> --data-string
"6:<node_to_apply_malleability>:<processes>"
```

- *controller_node*: defined in controller/controller.dat (tipycally the first node indicated in nodefile and rankfile)
- *client_port*: first socket port in script parameter
- *node_to_apply_malleability*: expand/shrink will be performed in this nodename (it should be identified in run/nodefile.dat and controller/rankfiles/rankfile)
- *processes*: number of processes to create/remove: 2 creates two processes; -2 removes two processes

For more details, please, take a look at the manual in the root of the FlexMPI folder or send us an email.

2.6 Execution examples

Jacobi I/O can be easily tuned to provide different patterns of execution. The user can set three different arguments to change the behaviour of the CPU, I/O and Communication load. Jacobi I/O performs the same phases during its execution, which allows the users to see a cyclic pattern of the performance metrics.

As it has been described in section 2.5, the parameters of the Jacobi I/O that fix the behaviour of the applications are:

- **Argument no 6:** I/O intensity.
- **Argument no 7:** CPU intensity.
- **Argument no 8:** Communication intensity.

Each intensity (i) means that, for each iteration of the Jacobi I/O, there is a loop that executes i iterations of a code related to the target feature. If I/O intensity is zero, Jacobi I/O will not execute I/O operations. Otherwise, Jacobi I/O will do it.

Taking this into account, following you can see four different configurations of Jacobi I/O and their executions. The three first use cases show different behaviours due to the intensity of different features. The fourth use case corresponds to the same execution of use case 2 but executes two application instances to show the interference between them.

1. UC1: CPU intensity 10.
2. UC2: I/O intensity 4 and CPU intensity 4.
3. UC3: CPU intensity 4 and Communication 10.
4. UC4: Use case 2, but executing two instances of Jacobi I/O. It generates interference between them due to the I/O phases.

Figure 2.1 shows the behaviour of the first use case. As this example executes consecutive CPU phases, the performance keeps constant, except when the application is finishing (due to a synchronization point).

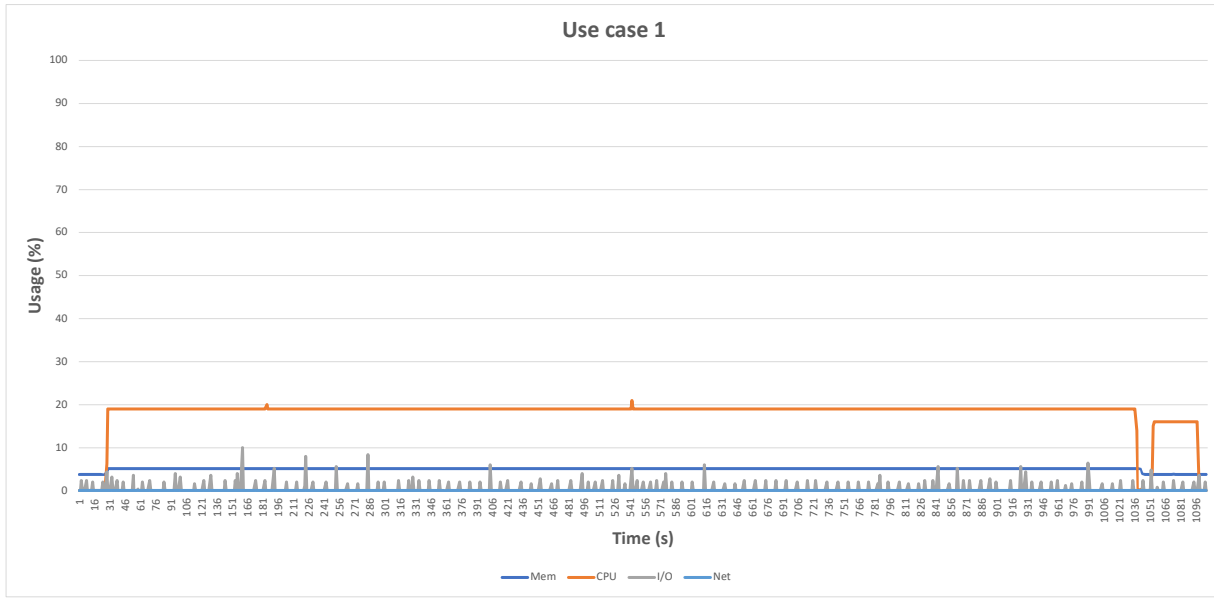


Figure 2.1: Use case 1 - Jacobi I/O executing only computation.

Figure 2.2 shows the behaviour of the second use case. This example combines CPU and I/O phases. Note that an I/O phase starts every 100 iteration.

Figure 2.3 shows the behaviour of the third use case. This example executes consecutive communication phases. Note that the usage of the network is based on the maximum speed of the interface.

Figure 2.4 shows the behaviour of the last use case. This example executes two Jacobi I/O instances concurrently. This deployment generates interference between the two Jacobi instances when both of them execute I/O operations. As it can be seen, the pattern observed in Figure 2.2 has changed due to the interference, and both instances need more time to complete the task. The first one has spent 2540 seconds and the second one 1484 seconds (676 seconds was expected based on the execution of the use case 2).

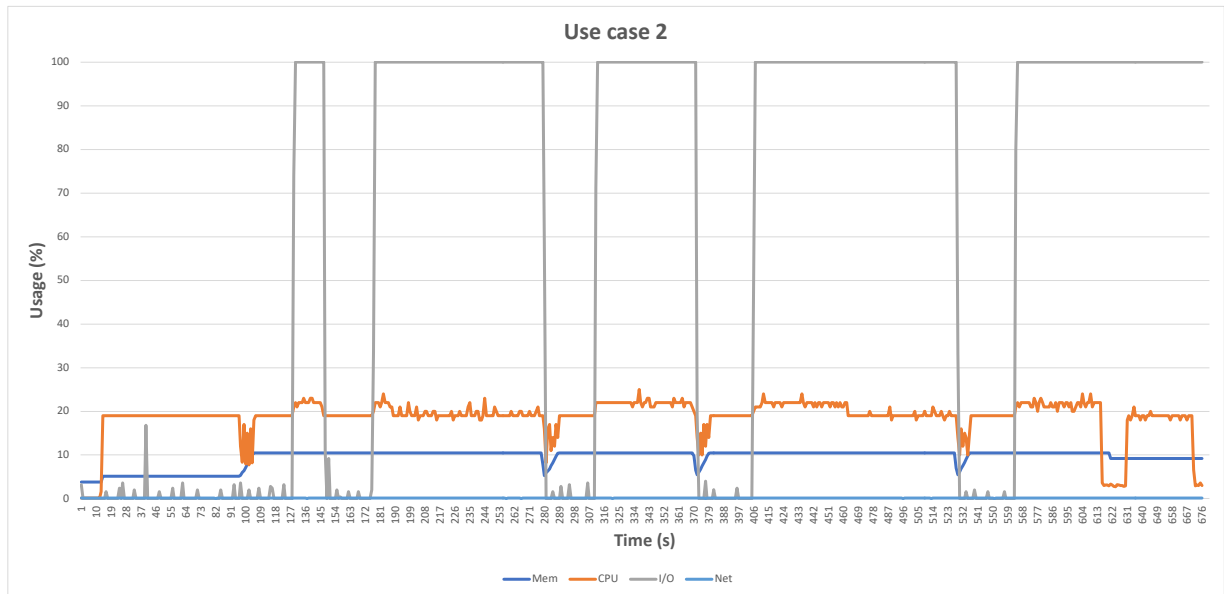


Figure 2.2: Use case 2 - Jacobi I/O combining CPU with I/O phases.

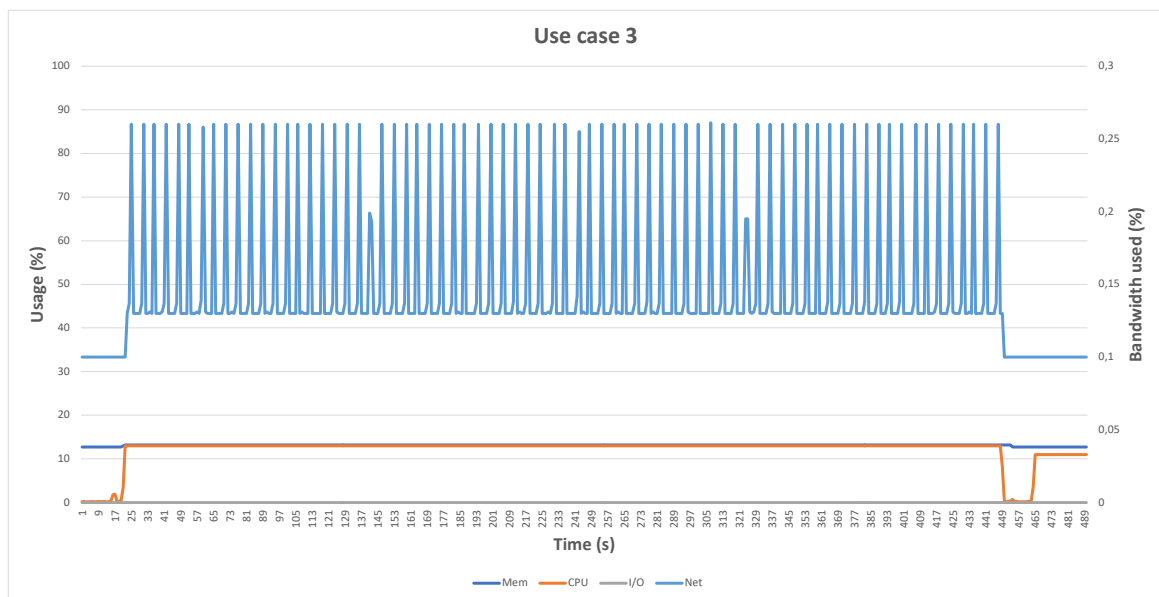


Figure 2.3: Use case 3 - Jacobi I/O executing communication between the processes.

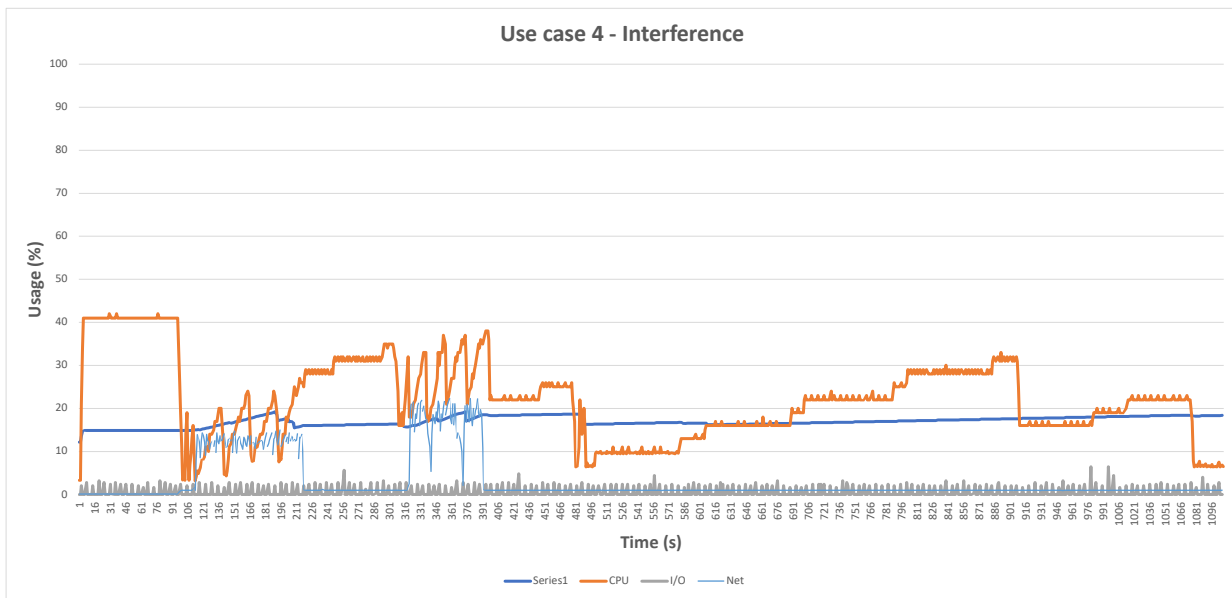


Figure 2.4: Use case 4 - Two instances of Jacobi I/O executing I/O each 100 iterations.