Lab1: introduction to MPI

September 26, 2022

1 Grid'5000

For our lab sessions, we will use the Grid'5000 platform (`https://www.grid5000.fr`), and more precisely one of the clusters of the Lille site (see: `https://www.grid5000.fr/w/Lille:Hardware`). Each cluster is composed of multiple (homogeneous) compute nodes, with possibly several CPUs (= processor chip) per node, and multiple cores per CPU (currently one or more dozen).

A first tutorial on Grid'5000 can be found here: `https://www.grid5000.fr/w/Getting_Started`

Please read the usage policy: `https://www.grid5000.fr/w/Grid5000:UsagePolicy`

The current state of use of the Lille clusters is available here:
`https://intranet.grid5000.fr/oar/Lille/monika.cgi`

Warning: the Grid'5000 access which has been granted to you is only valid for the HPC teaching unit. **Any other non-academic usage is forbidden.** It is possible to use Grid'5000 compute resources for other academic work (project, internship . . .), but you must warn me first (since I am your account manager).

1.1 Remote access

1.1.1 From a lab computer

In order to access the Lille site in Grid'5000, you must first obtain a `ssh` access to the outside of the lab room by requesting a session on: `https://vpn.centralelille.fr`

Then, please write these lines in the `~/.ssh/config` file **on your local account** (create this file if it does not exist), replacing USERNAME by your Grid'5000 login:

```
Host␣g5k
␣␣User␣USERNAME
␣␣Hostname␣access.grid5000.fr
␣␣ForwardAgent␣no

Host␣*.g5k
␣␣User␣USERNAME
␣␣ProxyCommand␣ssh␣g5k␣-W␣"$(basename␣%h␣.g5k):%p"
␣␣ForwardAgent␣no
```

You can now check that you can access Grid'5000 by running[1]: (from a computer where your `ssh` key is stored) in a terminal (hereafter referred to as terminal 1):

```
$␣ssh␣USERNAME@lille.g5k
```

You should now be logged on a computer named `flille`.

1.1.2 From your laptop

On a Windows laptop, to have an `ssh` access with `PuTTY` or `OpenSSH` see here:
`https://www.grid5000.fr/w/SSH#Windows_users`

Then you should access Grid'5000 with something similar to (depending on your `ssh` client):

```
ssh␣-t␣votre_login@access.grid5000.fr␣ssh␣lille
```

On a Linux laptop, see Section 1.1.1 (the VPN authorization may not be required).

[1]The `$` character indicates the shell prompt.

1.2 Remote file editing

1.2.1 From a lab computer

For editing files on Grid'5000, we will mount the Grid'5000 file system through `ssh` with the following commands (to be run on your local computer, in another terminal – hereafter referred to as terminal 2):

```
$␣chmod␣755␣$HOME
$␣mkdir␣G5K_lille
$␣sshfs␣-o␣idmap=user␣USERNAME@lille.g5k:/home/USERNAME␣G5K_lille
```

You can now check that you can create directories and edit files on your local computer in the `G5K_lille` directory (in terminal 2), and that these are actually stored on the `flille` computer (`ls` and `cat` commands in terminal 1).

Important (possible file corruption otherwise!): at the end of the lab session, you have to "unmount" the file system mounted through ssh in `G5K_lille` with

```
$␣fusermount␣-u␣G5K_lille/
```

The next time you want to mount/unmount the Grid'5000 file system through `ssh`, only the `sshfs`/`fusermount` commands are required.

1.2.2 From your laptop

On a Windows laptop, you should use an editor in text mode (`vi`, `vim`, `nano...`) in your `ssh` terminal to edit your source files.
On a Linux laptop, you can try `sshfs` as in Section 1.2.1, or `ssh` and an editor in text mode.

2 OpenMPI

On Grid'5000, we will use **OpenMPI** which implements the **MPI** standard. The programming mode is the **SPMD** mode: same executable for all processes, branchings depending on the process number (its **rank**) allowing them to perform different tasks.

To compile with MPI a C program named `foo.c` in an executable named `foo`, you have to use `mpicc` as:

```
$␣mpicc␣-o␣foo␣foo.c
```

With OpenMPI, one has to use the `mpirun` command to run this MPI program on a parallel architecture. However, within Grid'5000 the `mpirun` command has to be used with the `oarsub` and `oarsh` ones in order to obtain and access compute resources (CPU cores and memory) on one cluster. To do so, you must first configure OpenMPI **on your Grid'5000 account** (i.e. on `flille`) by adding these lines to the `~/.openmpi/mca-params.conf` file (create the `~/.openmpi` directory and this file if they do not exist):

```
plm_rsh_agent=oarsh
filem_rsh_agent=oarcp
btl_base_warn_component_unused␣=␣0
```

We can now allocate resources (here 4 cores within 1 cluster) and run our MPI program on these resources thanks to the following command (which will create and run a "job"):

```
$␣oarsub␣-l␣cluster=1/core=4␣'mpirun␣--mca␣pml␣^ucx␣-machinefile␣$OAR_NODEFILE␣./foo'
```

`mpirun` will then use all allocated resources, and run one MPI process per core. For our labs, we will use one cluster at at time and only the requested number of cores (usually powers of 2 between 1 and 16) or the executable name should be changed. If one wants to provide arguments (e.g. `a1` and `a2`) on the command line of `foo`, these should be specified as:

```
$␣oarsub␣-l␣cluster=1/core=4␣'mpirun␣--mca␣pml␣^ucx␣-machinefile␣$OAR_NODEFILE␣./foo␣a1␣a2'
```

The cluster name can also be specified (in order to perform multiple performance tests on the same hardware) as:

```
$ oarsub -l {"cluster='chetemi'"}/core=4 'mpirun --mca pml ^ucx -machinefile $OAR_NODEFILE ./foo'
```

If your job is long enough, you can see it executing by (re)loading the Monika web page (displaying the current state of use: `https://intranet.grid5000.fr/oar/Lille/monika.cgi`). Once your job is completed, you can check the output in a file named as `OAR.1234567.stdout` (where `1234567` is the job id). The error messages are printed in `OAR.1234567.stderr`. If your job does not start its execution, you can print informations on all submitted jobs with `oarstat` and delete your job with `oardel`.

For debugging your MPI programs, the easiest way (for our labs) is to rely on `printf` (taking care of the print order between different processes: see section 3).

3 Labs

You can start by downloading the latest MPI documentation (`pdf` format) at: `http://www.mpi-forum.org/`
For this first lab, we will start by using `Send` et `Recv` functions.

Exercice 1

What should be printed by the following MPI program?
Check this by writing, compiling and running this program on Grid'5000.

```c
#include <stdio.h>
#include <mpi.h>

int main( int argc, char* argv[]){
  int rank, p, val, tag = 10;
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if ( rank == 1){
    val = 18; MPI_Send(&val, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
  }
  else if ( rank == 0 ){
    MPI_Recv(&val, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
    printf("I received the value %d from process with rank 1.\n",val);
  }

  MPI_Finalize();
}
```

Exercice 2

Here is a more advanced program:

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>
#include <unistd.h>
#define SIZE_H_N 50

int main(int argc, char* argv[]){
  int          my_rank;      /* process rank         */
  int          p;          /* number of processes */
  int          source;     /* sender rank         */
  int          dest;       /* receiver rank       */
  int          tag = 0;    /* message tag         */
  char         msg[100];
  MPI_Status   status;
  char  hostname[SIZE_H_N] ;  gethostname(hostname,SIZE_H_N);

  /* Initialization:  */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  /* Communications: */
  if (my_rank != 0){
    /* Message creation */
    sprintf(msg, "Hello from process #%d from %s!", my_rank, hostname);
    dest = 0;

    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  } else {
    for (source = 1; source < p; source++) {
      MPI_Recv(msg, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
      printf("On %s, the process #%d received the message: %s\n",
             hostname, my_rank, msg);
    }
  }

  /* Termination: */
  MPI_Finalize();
}
```

question 1: same ordering of the displays
question 2: for a given process all the printf instructions appear on the 'correct ordering
question 3: The reception ordering varies from execution to the next
--> first message arrival means first message printed
question 4 : MPI_send >> MPI_Ssend
question 6: if the message size increase MPI Eager limit=ko

1. Run this MPI program multiple times with the same number of processes, and then by varying this number.
2. Add some `printf` at different places in the source code. What can you conclude?
3. Replace the `source` variable in the `MPI_Recv` call by the `MPI_ANY_SOURCE` wild card. Run some tests multiple times. What can you conclude?
4. Write an MPI program (in `ex_send.c`) where each process sends a string to its successor (i.e. to the process with rank $my\_rank + 1$ if $my\_rank < p - 1$, and 0 otherwise), and receives a message from its predecessor. Once your program runs correctly, replace `MPI_Send` with `MPI_Ssend` in a new program (stored in `ex_ssend.c`). What is happening when executing this new program?
5. Copy `ex_ssend.c` in a new `ex_ssend_correct.c` file. In `ex_ssend_correct.c`, while still using `MPI_Ssend`, change the algorithm such that process 0 sends first its message to process 1, which will send its message to process 2 only after having received the message from process 0. In the same way, process 2 will send its message to process 3 only after having received the message from process 1, and so on...
6. In the original `ex_send.c` file, gradually increase the size of the data sent with `MPI_Send` until 100 KB. What is happening?

Exercice 3 – (optional)

Considering P processes, we want to implement a reduction algorithm which sums integer values from all processes (one integer per process). The root of the reduction algorithm will be the process with rank 0 (and the final result will thus be printed by this process only).

1. Design an efficient algorithm *without* any collective communication routine.
2. Write a parallel MPI program implementing such algorithm. The program will compute the global sum of random integer values generated by each process, such that the process with rank 0 will receive and print this global sum.
3. Rewrite your MPI program with a collective communication routine, and then modify it such that the global sum is available in all processes.