INFO-F-404 : Operating Systems II

1 HYDRA Cluster

1.1 What is Hydra ?

Hydra is a High Performance Computing cluster. HYDRA replaced older cluster called ASTER.

For more information visit https://cc.ulb.ac.be/hydra/

1.2 Connect to HYDRA

In order to log-in to HYDRA : just use ssh[1] :

1. Open a terminal
2. Type <your-ULB-netid-login>@hydra.ulb.ac.be
3. Use your ULB netid password

Now you are logged-in on HYDRA ! Now you can use all standard *Linux* commands like `ls` or `cd`, `cat`, etc.

Your default shell is *bash*, but if you want you can use *csh*.

[1] also visit https://cc.ulb.ac.be/hydra/Howto/login_hydra.php

1.3 Sent files to HYDRA

In order to send a file to hydra you may use *scp* or *sftp*. Here is an example of log-in and file transfer with *sftp* (also visit https://cc.ulb.ac.be/hydra/Howto/transfer_files.php):

```
~>sftp <your-netid-login>@hydra.ulb.ac.be
<your_netid_login>@hydra.ulb.ac.be's password:
Connected to hydra.ulb.ac.be.
sftp> put table
table.py   table.txt
sftp> put table.txt
Uploading table.txt to /sulb2/<your_home_dir>/table.txt
table.txt                                     100%  552     0.5KB/s   00:00
sftp> get test/jobfile
Fetching /sulb2/<your_home_dir>/jobfile to jobfile
/sulb2/<your_home_dir>/test/jobfile           100%  340     0.3KB/s   00:01
sftp> bye
```

Here is an example of a file transfer from a local machine to the remote server with *scp*:

```
~>scp hello_world.cpp <remote_login>@hydra.ulb.ac.be:/u/<home_dir>/<path>
<your_netid_login>@hydra.ulb.ac.be's password:
hello_world.cpp  100% 7301     7.1KB/s   00:00
```

For this class we ask you to do the following steps:

1. download the file "hello_world_mpi.cpp" from *Université Virtuelle*,
2. create a file "jobfile_hello",
3. login to HYDRA and create a new directory "MPI_test"
4. transfer files "hello_world_mpi.cpp" and "jobfile_hello" in this directory using SFTP or SCP,
5. execute "hello_world" once (using comandline) and once using a jobfile "jobfile_hello",
6. play with different parameters in commandline and in the jobfile.

Read following sections for more explanations on these steps.
You may also use *vim* to write the code and create files on HYDRA.

1.4 How to execute MPI program on HYDRA

1.4.1 How does it work?

Any interactive execution on HYDRA is limited in CPU time. If your program needs a lot of time or if you want a parallel (distributed) execution, you have to use LSF (**L**oad **S**haring **F**acility). LSF handle "big" jobs that need a lot of resources (e.g. memory, CPU time). This system will execute a process as soon as it has enough of resources available. In the other case the job will stay in a queue.

In order to submit a job (program) to LSF we have to create a *jobfile* - a file that describes needs of our program, like number of parallel processes that we need. A jobfile is very similar to a shell script with a little addition : we can use special commands that will tell to LSF how to handle our program.

1.4.2 "hello world" MPI version

This little program is an example used for this course. Each instance of "hello_world" will print its ID and the ID of the node that executes it. Here is the source code of `hello_world.cpp` (also available on UV):

```
#include <mpi.h>
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
    int id, nb_instance, len;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &nb_instance);
    MPI_Get_processor_name(processor_name, &len);

    cout<<"Hello world! I'm "<<id <<" of "<<nb_instance<<" on "<<processor_name;
    cout<<endl;

    MPI_Finalize();

    return 0;
}
```

1.4.3 Execution of "hello world" using command line

You can easily execute `hello_world` (once compiled) using shell command line without jobfile. Here is an example:

```
module load openmpi/1.4.3/gcc/4.6.1
mpiCC hello_world.cpp -o hello
mpirun -np 4 ./hello
```

The First line loads *openmpi* module (you may find all available modules by typing `module avail`). Second line compiles our `hello_world.cpp` program and the last one executes `hello` using MPI, the parameter *-np* specifies the number of parallel executions.

1.4.4 Creation of a jobfile

Here is an example of a jobfile for our little program "hello_world.cpp"

```
#!/bin/bash -l
#PBS -l nodes=4:ppn=8
#PBS -l walltime=00:02:00
#PBS -l pmem=2kb
#PBS -j oe
#PBS -N Hello
#PBS -o hello_out.txt

echo "Running job on $HOST - " `date`

module load openmpi/1.4.3/gcc/4.6.1

cd $HOME/helloWorld/

mpiCC hello_world.cpp -o hello
mpirun ./hello
echo "Done"
```

The first six lines are MPI specifications for `hello\_world`. These specifications have the following format

```
#PBS -option value
```

> You can find a full list of available options https://cc.ulb.ac.be/hydra/Howto/prepare_jobs.php. Also visit https://cc.ulb.ac.be/hydra/Howto/compile_mpi.php for compilation options.

1.4.5 How to execute "hello world" using a jobfile

Once your jobfile is created, you can submit it using the following line:

```
qsub jobfile_hello
```

Answer:

```
37688.mn01.hydra.vub.ac.be
```

After the execution you can read the output using `cat` :

```
cat filename
```

filename is the name of the file that you specified in the jobfile (in our case: *filename* is the file named "hello_out.txt").

2 MPI calls

MPI_Init(&argc,&argv) Initialise l'environnement MPI. Les deux paramètres passés à cette fonction correspondent tout simplement aux paramètres de la fonction main(). MPI_Init est la première instruction MPI (obligatoire) que doit exécuter chaque instance (processus) du programme. Un processus ne peut (et ne doit) faire appel à cette fonction qu'une seule fois durant son exécution. Cette fonction a pour but d'attribuer un ID unique à chaque instance et d'enregistrer cet ID dans un ensemble appelé MPI_COMM_WORLD.

Datatype MPI	Datatype C/C++
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack() / MPI_Unpack

Table 1: Datatypes defined by MPI.

MPI_Comm_size(comm,&size) Retourne dans la variable *size* le nombre d'instances enregistrées dans l'ensemble *comm*. En général, la variable *comm* utilisée est la variable globale *MPI_COMM_WORLD*.

MPI_Comm_rank(comm,&rank) Retourne dans la variable *rank* l'ID du processus qui fait appel à cette fonction (pour rappel, cet ID a été attribué par la fonction MPI_Init et les ID sont tous répertoriés par défaut dans la variable globale *MPI_COMM_WORLD*.).

MPI_Abort(comm,errorcode) Termine tous les processus appartenant à l'ensemble *comm*. L'argument *errorcode* indique à cette procédure le code d'erreur pour lequel on demande l'arrêt des processus.

MPI_Get_processor_name(&name,&resultlength) Retourne le nom du processeur sur lequel s'exécute le processus qui appelle cette fonction. Cette fonction retourne également la longueur de ce nom. La variable *name* utilisée doit être un vecteur d'au moins *MPI_MAX_PROCESSOR_NAME* caractères.

MPI_Initialized(&flag) Indique si la procédure MPI_Init à été appelée - retourne dans la variable *flag* la valeur **true** ou **false**.

MPI_Wtime() Retourne le temps passé (en secondes) sur le processeur.

MPI_Finalize() Retire le processus de l'environnement MPI (c'est-à-dire de l'ensemble *MPI_COMM_WORLD*) et supprime cet environnement s'il s'agit du dernier processus. Un processus ne peut utiliser aucune autre procédure MPI après avoir exécuté cette instruction.

2.1 Send and Receive

Send and Recieve use same kind of argments.

blocking Send : **MPI_Send**(*&buffer,count,type,dest,tag,comm*)
non-blocking Send : **MPI_Isend**(*&buffer,count,type,dest,tag,comm,&request*)
blocking Recieve : **MPI_Recv**(*&buffer,count,type,source,tag,comm,&status*)
non-blocking Recieve : **MPI_Irecv**(*&buffer,count,type,source,tag,comm,&request*)

- *buffer* contains a value to send or a variable that will "receive" a value.

option		Datatype C/C++
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	somme	integer, float
MPI_PROD	produit	integer, float

Table 2: MPI Gather options.

- *count* number of elements that have to be sent (received)
- *type* Datatype see Table 1. User can also create new datatypes.
- *dest* destination process ID
- *source* sender process ID
- *tag* unique message identifier, if you do not need to identifiers for messages you may use MPI_ANY_TAG.
- *comm* environment, for this course we will use the default standard MPI_COMM_WORLD.
- *status* source (*status.MPI_SOURCE*) and tag (*status.MPI_TAG*) of the message
- *request* used by non-blocking send and receive (unique id).

2.2 Multi-process communications

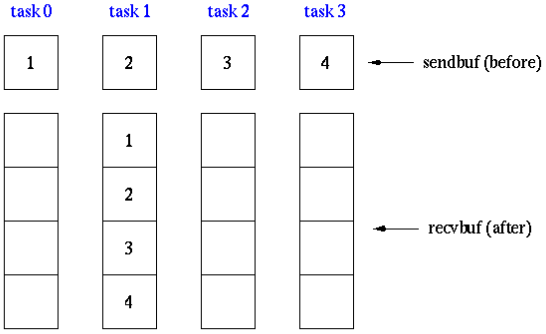
MPI_Barrier(*comm*) Synchronisation barrier.
For following functions an image is worth 1000 words, so ...

- MPI_Bcast(*&buffer,count,datatype,root,comm*)
- MPI_Scatter(*&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype,root,comm*)
- MPI_Gather(*&sendbuf,sendcnt,sendtype,&recvbuf,recvcount,recvtype,root,comm*)
- MPI_Allgather(*&sendbuf,sendcount,sendtype,&recvbuf,recvcount,recvtype,comm*)
- MPI_Reduce(*&sendbuf,&recvbuf,count,datatype,op,root,comm*)
- MPI_Allreduce(*&sendbuf,&recvbuf,count,datatype,op,comm*)
- MPI_Reduce_scatter(*&sendbuf,&recvbuf,recvcount,datatype,op,comm*)
- MPI_Alltoall(*&sendbuf,sendcount,sendtype,&recvbuf,recvcnt,recvtype,comm*)
- MPI_Scan(*&sendbuf,&recvbuf,count,datatype,op,comm*)

MPI_Gather

Gathers together values from a group of processes

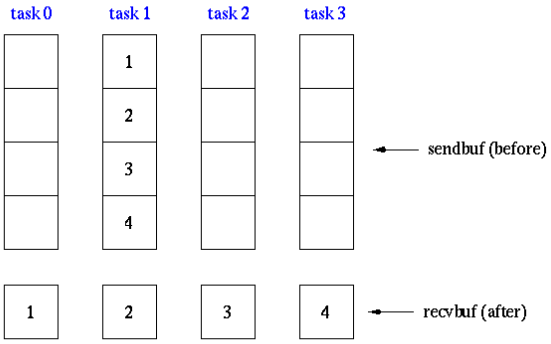
```
sendcnt = 1;
recvcnt = 1;
src = 1;                messages will be gathered in task 1
MPI_Gather(sendbuf, sendcnt, MPI_INT,
           recvbuf, recvcnt, MPI_INT,
           src, MPI_COMM_WORLD);
```



MPI_Scatter

Sends data from one task to all other tasks in a group

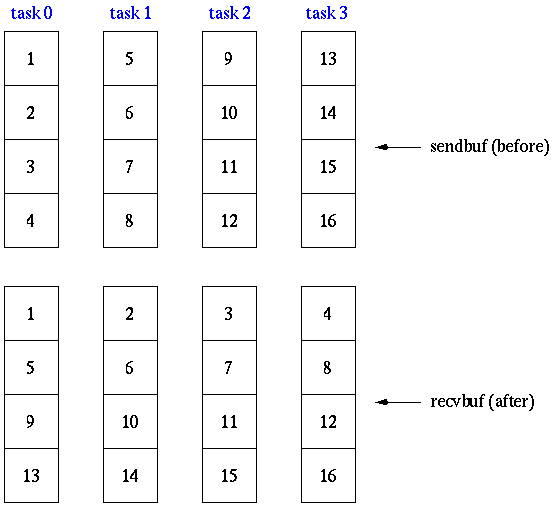
```
sendcnt = 1;
recvcnt = 1;
src = 1;                task 1 contains the message to be scattered
MPI_Scatter(sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcnt, MPI_INT,
            src, MPI_COMM_WORLD);
```



MPI_Alltoall

Sends data from all to all processes. Each process performs a scatter operation.

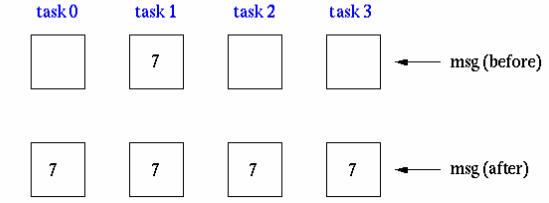
```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,
             recvbuf, recvcnt, MPI_INT,
             MPI_COMM_WORLD);
```



MPI_Bcast

Broadcasts a message to all other processes of that group

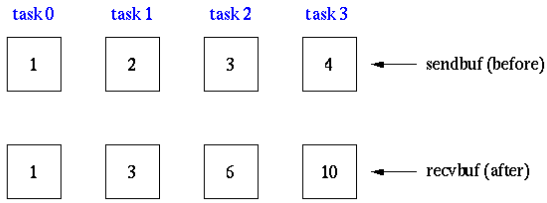
```
count = 1;
source = 1;                    broadcast originates in task 1
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```



MPI_Scan

Computes the scan (partial reductions) of data on a collection of processes

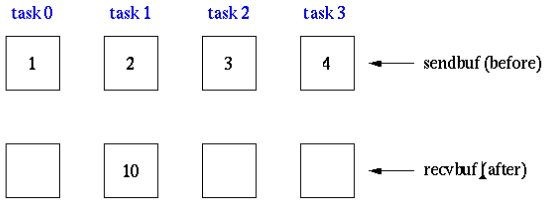
```
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
         MPI_COMM_WORLD);
```



MPI_Reduce

Perform and associate reduction operation across all tasks in the group and place the result in one task

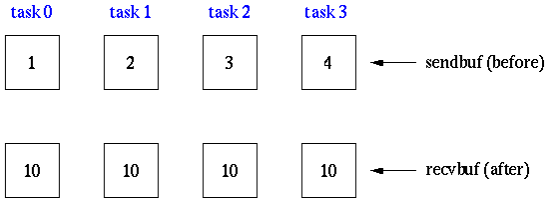
```
count = 1;
dest = 1;                      result will be placed in task 1
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
           dest, MPI_COMM_WORLD);
```



MPI_Allreduce

Perform and associate reduction operation across all tasks in the group and place the result in all tasks

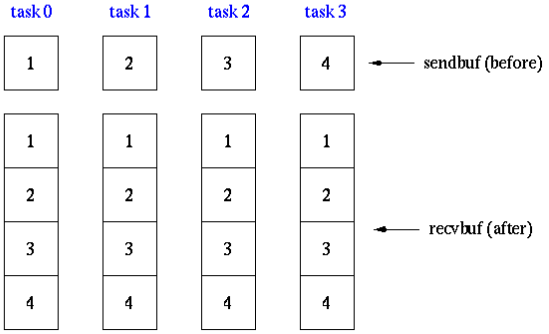
```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
              MPI_COMM_WORLD);
```



MPI_Allgather

Gathers together values from a group of processes and distributes to all

```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT,
              recvbuf, recvcnt, MPI_INT,
              MPI_COMM_WORLD);
```



MPI_Reduce_scatter

Perform reduction operation on vector elements across all tasks in the group, then distribute segments of result vector to tasks

```
recvcount = 1;
MPI_Reduce_scatter(sendbuf, recvbuf, recvcount, MPI_INT, MPI_SUM,
                   MPI_COMM_WORLD);
```