EPFL

HPC for numerical methods and data analysis

Fall Semester 2024

Prof. Laura Grigori

Assistant: Mariana Martínez Aguilar

Session 2 – September 17, 2024

Clusters, Python, and MPI

1 Clusters

A workload manager like Slurm is designed to provide the system administrator with increased control over how the scheduler virtual memory manager (VMM) and the disc I/O subsystem allocate resources to processes. Our cluster, helvetios is managed using Slurm.

How to connect to EPFL's cluster?

We're going to make sure that we can correctly connect to the cluster available for this course. If you have more questions you can read the documents in depth here. **Make sure you have a GASPAR account**, let your username be ⟨username⟩. The cluster available for us is called *helvetios*. The account for this course is math-505. Take into consideration that you can only connect if you're physically at EPFL, otherwise you'll need to use a VPN.

To connect we're going to use the SSH protocol. Click here to read more about SSH. In the terminal connect as follows with your GASPAR username

```
$ ssh <username>@helvetios.hpc.epfl.ch
```

You should be prompted to the home directory. There is a tips section, announcements (when the clusters are going to be under maintenance, etc.), and Sausage (information about your account, how many jobs you've ran, etc). To end your session just type

```
$ exit
```

Notice that you get asked to type your password. There is an option to generate ssh keys so that you don't need to type it every time you want to use the cluster. To learn how to do this read this webpage.

How to get your code on the cluster?

If you have never used helvetios then if you're signed into the cluster and type the ls command you're going to notice that there are no folders. In order to run *something* on the cluster you need to get your files in the cluster *somehow*. Then, in order to get the results of your program locally, you need to get files from the cluster to your computer *somehow*. That somehow can be done in two ways: either with the rsync command or the scp command.
The rsync command will only synchronize data that has changed, thus reducing the network usage and transfer time (somehow like github). It can resume synchronization from where it was in case of an interruption. This is the preferred way of transferring files to and from the cluster. The syntax is the following:

```
$ rsync [options] <src> <dest>
```

where the options are the following:

- -a: archive mode. Among other things, it copies recursively, keeps symlinks, preserves file permissions and modification times.
- -z: use data compression.
- -P: show progress.

For example, to load files into the cluster you can run

```
$ rsync -azP /path/to/src <username>@helvetios.hpc.epfl.ch:/path/to/dest
```

whereas to load files from the cluster to your computer you can run

```
$ rsync -azP <username>@helvetios.hpc.epfl.ch:/path/to/dest /path/to/src
```

Remember that you need to run these commands while not connected to the cluster. You can click here to learn more and read about the usage of scp. Another way of getting files to and from the cluster would be via github.

How to run code on the cluster?

Once you have your scripts loaded in the cluster, you can specify how they should be used. There are two main steps for creating a job:

- Request resources. This involves specifying things like required number of CPUs/GPUs, expected job duration, amounts of RAM to be reserved, disc space, etc.
- Define job steps. This is describing what needs to be done: computing steps, which software to run, parameter space to try out, etc.

A job is created via a submission script (e.g. a shell script). Simply running the typical `mpiexec ...` won't work. An example of such script can be found below

```
#!/bin/bash -l
```

```
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=19
#SBATCH --cpus-per-task=1
#SBATCH --time=1:20:00
#SBATCH --account=math-505
#SBATCH --qos=parallel
#SBATCH --output=./result0.txt

module load gcc openmpi python py-mpi4py

srun ./script.py
```

This script has to be saved as a file, for example `exercise0.run`. A good idea is to use the extension `.run` (even though this is a shell script) to separate this from other types of scripts we might have in the working directory. Don't just copy paste this script directly to the terminal, it won't work. The first line is called hashbang, the submission file has to start with this. The next lines must be SBATCH directives. #SBATCH is understood by Slurm as a parameter describing resource requests. After such lines we can input any other line, in this case we're loading certain modules that allow us to run mpi and Python. The last line tells Slurm to run a certain script in the current folder.
Note that in helvetios when we set the parameter "qos" to be "parallel" we need at least 1 node and at least 37 cores. If this requirement is not met then "qos" needs to be set to "serial" (for example, when you want to run your code with only 4 processors).
Slurm directives is the way Slurm determines how to allocate jobs in a cluster. They're indicated by lines starting with #SBATCH. Some of the most used ones are

- - -cpus-per-task Specifies the number of vCPUs required per task It must be equal or less than the number of vCPUs available on a single compute node.
- - -job-name Specifies a name for the job. The specified name will appear along with the job id when querying running jobs on the system. The default is the name of the batch script.
- - -mem or –mem-per-cpu Specifies the memory in MB required per node or per vCPU respectively for the job. Parallel cluster doesn't support Slurm memory directives hence including these directives will cause computer nodes to go into a "DRAINED" state and prevent successful allocation of the job. It's recommended ommiting these directives from your script, and by default the job will have access to all of the memory on each compute node.
- - -nodes Requests that a minimum number of nodes are allocated to the job. If not specified, the default behaviour is to allocate enough nodes to satisfy other requested resources.
- - -ntasks Advises Slurm that a certain number of tasks will be launched from the job. It is usually only required for MPI workloads and requires the use of the srun command to launch the separate tasks from the job script.
- - -ntasks-per-node Requests that a certain number of tasks be called on each node.
- - -time Sets a limit on the total run time of the job.

Exercise 0 A simple MPI code with Slurm

We're only going to be implementing single thread programs. Consider the following Python code called "exercise0.py"

```
from mpi4py import MPI

# Intialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
sz = comm.Get_size()
print("Number of processors used: ", sz)
print("Rank: ", rank)
```

Then consider the following submission script, exercise0.run , for it

```
#!/bin/bash -l

#SBATCH --nodes=2
#SBATCH --ntasks-per-node=20
#SBATCH --cpus-per-task=1
#SBATCH --time=1:20:00
#SBATCH --account=math-505
#SBATCH --qos=parallel
#SBATCH --output=./result0.txt

module load gcc openmpi python py-mpi4py
srun python exercise0.py
```

Create this job, run it, and transfer the resulting file (result0.txt) to your local computer. Answer the following questions

a) What is the output result0.txt? *Solution:* it just prints the ranks in random orders. The number of ranks mpi2py registers equals the number of nodes times the number of tasks per node.

b) How many processors are being used? *Solution:* 40.

c) Change the number of nodes to 4 and the number of tasks per node to 10. Is there a difference in the output? What is happening? How is this parallelization done? *Solution:* Nothing apparently happened, mpi4py still registers 40 different ranks. What happened is that we requested different resources from the cluster. There are different ways of running a parallel code with different resources.

d) What is the difference between a task, a job, and the number of CPUs used? *Solution:*

- A job comprises one or more steps, each step has one or more parallel tasks.
- A task is an instance of a running program.
- A CPU in Slurm context is a consumable resource offered by a node.

Exercise I Reminder of a simple MPI code in Python

Given two vectors, b, c we want to compute $d = 2b + c$. Let this script be exercise1.py

```python
from mpi4py import MPI
import numpy as np

# Initialize the variables
b = np.array([1, 2, 3, 4])
c = np.array([5, 6, 7, 8])
a = np.zeros_like(b)
d = np.zeros_like(b)

# Intialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
sz = comm.Get_size()
print("Number of processors used: ", sz)
print("Rank: ", rank)
if rank == 2:
    print("Rank 2")

# Do different things in the processes
if rank == 0:
    for i in range(4):
        a[i] = b[i] + c[i]
    comm.Send(a, dest = 1, tag = 77)
else:
    comm.Recv(a, source = 0, tag = 77)
    for i in range(4):
        d[i] = a[i] + b[i]

# Print
print("I am rank = ", rank )
print("d: ", d)
```

Consider the following submission script, exercise1.run for it

```
#!/bin/bash -l
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --time=20:00
#SBATCH --qos=parallel
#SBATCH --account=math-505
#SBATCH --output=./result1.txt

module load gcc openmpi python py-mpi4py
srun python exercise1.py
```

a) What is the output? Is there a problem with it? What goes wrong? Fix it while keeping "qos" set to "parallel". *Solution:* we're using qos = parallel so we have to have at least 37 cores. In this example we only have 2.

b) What's the order in which the prints take place and the value of d at the end? *Solution:* the order is random but the outcome is always the same, the sum $d = 2b + c$.

c) How many processors are available on this Python script? (i.e. what's the output of sz) *Solution:* depends how you "fixed" the problem but there should be at least 37.

d) Are all the processors being used? *Solution:* No, we only needed 2 so Slurm only used 2, regardless of the amount of resources we asked for.

e) Change the number of nodes to 4. How do your answers from a-c change?

Exercise II Point to point communication - blocking and non-blocking communication

a) Provide a brief definition of MPI. What is a communicator? *Solution:* MPI stands for "Message Passing Interface", it is a standard released in 1994 for message passing library for parallel programs. Communicators are objects that provide the appropriate scope for all communication operations. Click here for the source.

b) Execute the following simple code on 4 processors.

```
from mpi4py import MPI
import numpy as np

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    print("From process: ", rank, "\n data sent:", data, "\n")
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print("From process: ", rank, "\n data received:", data, "\n")
elif rank == 2:
    data = np.array([1, 1, 1, 1, 1])
    print("From process: ", rank, "\n data sent:", data, "\n")
    comm.send(data, dest=3, tag = 66)
else:
    data = comm.recv(source = 2, tag = 66)
    print("From process: ", rank, "\n data received:", data, "\n")
```

In this case, why do we need to be careful when specifying the `dest` and `tag` parameters on both `comm.send` and `comm.recv`? *Solution:* In this case we need to be careful when specifying `dest` because we have 4 processes. We are sending data from process 0 to process 1 and from process 2 to process 3. We don't want to make sure this is done accordingly. The parameter `tag` acts as a filter and ensures that even if we send two messages they are received correctly.

c) Describe the difference between blocking communication and non-blocking communication in MPI. Modify the code above such that it uses `comm.isend` instead of `comm.send` and `comm.irecv` instead of `comm.recv` while ensuring the messages are passed correctly. *Solution:* In blocking communication, the sender or receiver is not able to perform any other actions until the corresponding message has been sent or received (technically, until the buffer is safe to use). This is done with `comm.send` and `comm.recv`. In non-blocking communication, the program is allowed to continue execution while the message is being sent or received. This is achieved with `comm.isend` and `comm.irecv`. These two methods return an instance of the class `Request`. The completion can then be managed using the Test, Wait, and Cancel methods of this class.

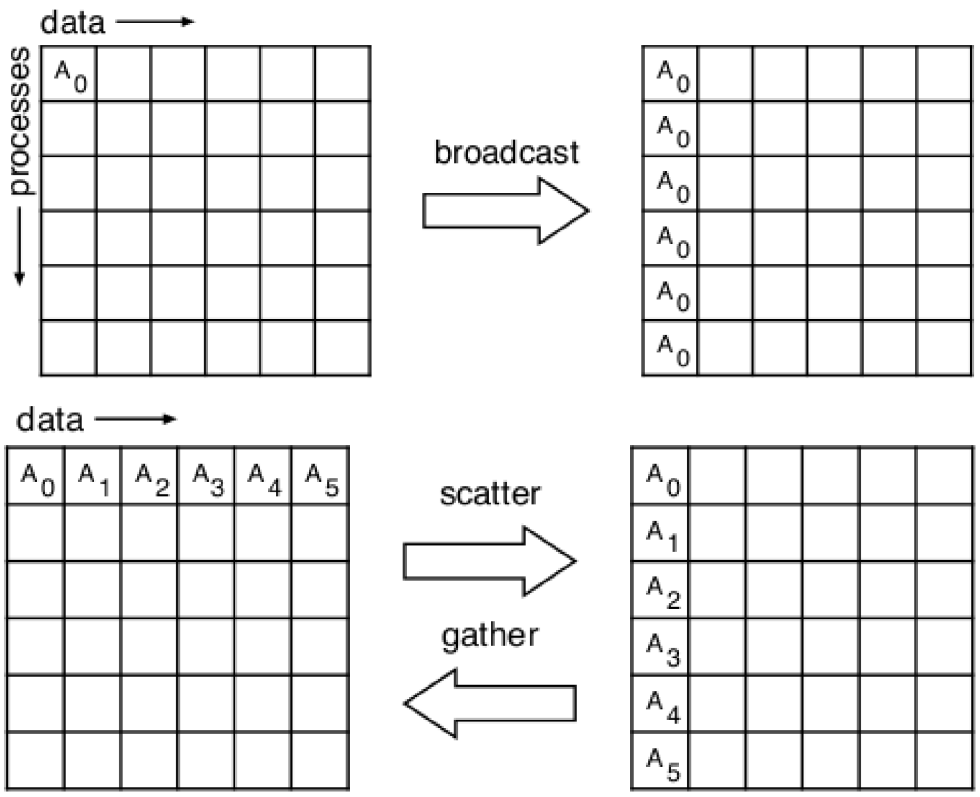
As seen below, the method `wait` immediately following the non-blocking methods blocks the process until the corresponding send and receives have completed.

Exercise III Collective communication - scattering and broadcasting

a) Run the following script on 4 processors:

```
from mpi4py import MPI
import numpy as np

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Define the vector
if rank == 0:
    vector = np.array([16, 62, 97, 25])
else:
    vector = None

data1 = comm.bcast(vector, root = 0)
data2 = comm.scatter(vector, root = 0)

print("rank: ", rank, " data1: ", data1, " data2: ", data2)
```

What is the difference in MPI between scattering and broadcasting? *Solution:* A broadcast and scatter are two of the standard collective communication techniques. During a broadcast, one process sends the same data to all processes in a communicator. One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes. In comparison, scatter involves a designated root process sending data to all processes in a communicator. Broadcast sends the same piece of data to all processes while scatter sends chunks of an array to different processes. Gather is the inverse of scatter. Instead of spreading elements from one process to many processes, the gather operation takes elements from many processes and gathers them to one single process. Below are some diagrams illustrating these communication techniques.



Click here or here to read the sources of this answer.

b) Consider the multiplication of a matrix $A \in \mathbb{R}^{m \times n}$ with a vector $v \in \mathbb{R}^n$. Write a Python file containing a script that:

- Creates a matrix of dimension $m \times n$
- Creates a vector of dimension n
- Makes sure that the dimensions of the matrix and the vector agree in such way that we can compute Av
- Computes Av using MPI's scattering, make sure you execute your code on the right amount of processors (*Hints: you'll need to use* `comm.gather`. *What are the entries of* Av*?*)

Solution: the following code does this. The submission file to run this in the cluster is not provided since this answer was based on running such code with only 4 processors.

```
# Generated with ChatGPT (modified)
from mpi4py import MPI
import numpy as np

# Function to perform matrix-vector multiplication
def matrix_vector_multiplication(matrix, vector):
    result = np.dot(matrix, vector)
    return result
```

```
# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Define the matrix and vector
matrix = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])
vector = np.array([7, 8, 9, 10])

# Check if the matrix and vector dimensions are compatible
if matrix.shape[1] != len(vector):
    if rank == 0:
        print("Matrix and vector dimensions are not compatible for multiplication.")
elif size != matrix.shape[1]:
    print("Number of processors used doesn't match with number of rows in matrix.")
else:
    # Split the work among processes
    print(" rank: ", rank)
    local_row = comm.scatter(matrix, root=0)

    # Compute local multiplication
    local_result = matrix_vector_multiplication(local_row, vector)

    # Gather results on the root process
    global_result = comm.gather(local_result, root=0)

    # Print the result on the root process
    if rank == 0:
        print("Matrix:")
        print(matrix)
        print("Vector:")
        print(vector)
        print("Result:")
        print(global_result)
        print("Result with numpy: ")
        print( matrix@vector)
```

Exercise IV Collective communication - all-to-all and reduce

- Run the following code on 4 processors:

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

senddata = rank*np.ones(size, dtype = int)

recvdata = comm.alltoall(senddata)

print(" process ", rank, " sending ", senddata, " receiving ", recvdata )
```

- What is `comm.alltoall` doing? Compare it to `comm.scatter`. *Solution:* `alltoall` combines the scatter and gather functionality. It is better explained in the diagram below.

- In this exercise we are going to use reduction operations on MPI. Run the following code on 4 processors:

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

senddata = rank*np.ones(size, dtype = int)

global_result1 = comm.reduce(senddata, op = MPI.SUM, root = 0)
global_result2 = comm.reduce(rank, op = MPI.MAX, root = 0)

#Print
print(" process ", rank, " sending ", senddata)

#Print the result on the root process
if rank == 0:
    print(" Reduction operation1: ", global_result1,
          "\n Reduction operation2: ", global_result2)
```

  What is a reduction operation? What is the difference between this and `comm.gather`? *Solution:* Data reduction involves reducing a set of numbers into a smaller set of numbers via a function. The `comm.reduce` method takes an array of input elements and returns an array of output elements to the root process. The output elements contain the reduced result. MPI contains a set of common reduction operations that can be used, although custom reduction operations can also be defined. Click here to read the source of this answer.

- In the previous code, change `comm.reduce` to `comm.allreduce`. What is the difference between the two? (Note, `comm.allreduce` doesn't use the argument `root`). *Solution:* In `comm.reduce` the output array of the reduction is saved just in the root process but with `comm.allreduce` such output is saved in all the processes.

```
# Exericise 4: All-to-all
# Source: https://subscription.packtpub.com/book/programming/9781785289583/3/ch03lvl1sec49/colle

from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

senddata = rank*np.arange(size, dtype = int)

global_result1 = comm.reduce(senddata, op = MPI.SUM, root = 0)
global_result2 = comm.reduce(rank, op = MPI.MAX, root = 0)
global_result3 = comm.allreduce(senddata, op = MPI.SUM)
global_result4 = comm.allreduce(rank, op = MPI.MAX)

#Print
print("Process: ", rank,
      " operation1: ", global_result1,
```

```
      " operation2: ", global_result2,
      " operation3: ", global_result3,
      " operation4: ", global_result4)
```

Exercise V Deciding what to use - Mid point rule

Numerical integration describes a family of algorithms for calculating the value of definite integrals. One of the simplest algorithms to do so is called the Mid Point Rule. Assume that $f(x)$ is continous on $[a, b]$. Let n be a positive integer and $h = (b - a)/n$. If $[a, b]$ is divided into n subintervals, $\{x_0, x_1, ..., x_{n-1}\}$, then if $m_i = (x_i + x_{i+1})/2$ is the midpoint of the i-th subinterval, set:

$$M_n = \sum_{i=1}^{n} f(m_i)h.$$

Then:

$$\lim_{n\to\infty} M_n = \int_a^b f(x)dx.$$

Thus, for a fixed n, we can approximate this integral as:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} f(m_i)h$$

Set $n = s * 500$, $f(x) = \cos(x)$, $a = 0$, $b = \pi/2$. Write a Python script such that:

- Defines a function that given x_i, h, n first calculates 500 mid points on a subinterval $[x_i, x_{i+1}]$ and returns the approximation of the integral on this subinterval.
- Using MPI approximates the integral of f on $[a, b]$
- Run your script on s processors

to approximate the integral of f.

```
# Source: https://nyu-cds.github.io/python-mpi/05-collectives/
# (modified)
import numpy
from math import acos, cos, pi
from mpi4py import MPI


comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

def integral(x_i, h, n):
    integ = 0.0
    for j in range(n):
        x_ij = x_i + (j + 0.5) * h
        integ += cos(x_ij) * h
    return integ

a = 0.0
```

```
b = pi / 2.0
my_int = 0
integral_sum = numpy.zeros(1)

# Initialize value of n only if this is rank 0
if rank == 0:
    n0 = 500 # default value n = 500
else:
    n0 = 0 # if not n = 0

# Broadcast n to all processes
n = comm.bcast(n0, root=0)

# Compute partition
h = (b - a) / (n * size) # calculate h *after* we receive n
x_i = a + rank * h * n
my_int = integral(x_i, h, n)

# Send partition back to root process, computing sum across all partitions
print("Process ", rank, " has the partial integral ", my_int)
integral_sum = comm.reduce(my_int, MPI.SUM, root = 0)

# Only print the result in process 0
if rank == 0:
    print('The Integral Sum =', integral_sum)
```