TPs MPI

Mandatory exercises after Lab 1: 1, 2, 3, 4, 5

Mandatory exercises after Lab 2: 6, 7, 8

Mandatory exercises after Lab 3: 9, 10, 12

Animated examples of the behavior of MPI collectives is available there.

<https://www.irit.fr/~Georges.Da-Costa/par/mpi_collectives/>

# Exercise 1: warm up [Check the environment work, basic collective operations]

Run the **test\_graphic.py** and **test\_mpi.py** from the files for the **codes.zip** archive (in the **Labs** directory)

* **python3 test\_graphic.py**
* **mpirun -n 2 python3 test\_mpi.py**

Run the codes in the **Lesson** directory

* Run and analyze at least **bcast.py**, **gather.py**, **scatter.py** and **reduce.py**
* What is the difference between **scatter.py** and **scatter\_generic.py**? Try to run them with 3 nodes. Try with 4 nodes.
* Try **reduce.py** with 10 processes using the following command: **mpirun --oversubscribe -n 10 python3 reduce.py**

# Exercise 2: passnumber [Choose the right collective]

Consider that you have a number that is only available on the process with **rank==0**. You want to print it on the screen in all processes.

As an example, with 2 processes and the number **42** only available on the process with **rank==0**, the command **mpirun -n 2 number.py 42** could print: (missing python)

**At start in process of rank 0 the passnumber is 42**

**At start in process of rank 1 the passnumber is 0**

**After collective in process of rank 1 the passnumber is 42**

**After collective in process of rank 0 the passnumber is 42**

**From process of rank 0 the passnumber is 42**

**From process of rank 0 the passnumber is 42**

# Exercise 3: random generator [Choose the right collective]

We will check if the random generator from numpy is reproducible. The goal is to generate arrays of random numbers on multiple processes, then to check if all the values are the same. We will use the command **np.random.seed(0)** that initializes the random generator with a particular starting point. The goal is the following:

* On each process, initialize the random generator

On each process, generate 10 integers between 0 and 99 (see the **rand.py** file in the **Labs** folder of **codes.zip** for an example)

* Verify that the min and max of each of these 10 elements are the same on each array
* Print either **true** or **false** depending on the result
* Run it again without initializing the random generator (commenting the **seed** command)

# Exercise 4: sum of N first integers [compute the bounds locally in each process]

Compute in parallel the sum between 0 and an argument passed on the command line. We have a function **cumul(a,b)** that computes the sum of all values between a (included) and b (excluded). **cumul(1,4)** = 1+2+3 = 6. We assume the argument is a multiple of the number of processes. The goal of this exercise is to be able to compute the bounds (**a** and **b**) on each process independently

* Start with the sequential version (**Labs** folder of **codes.zip**) and test it (python3 **cumul.py 100** should give 4950)
* Compute the bounds **a** and **b** of **cumul** for each processwithout communication
* Finish the parallel version (using only one collective operation)

# Exercise 5: teams of process [Choose the right collective]

We want to make two teams of processes. The blue and the green ones. Only the process with **rank==0** has access to the file containing the information. As shown in the file **teams.py** (in the **Labs** folder of **codes.zip**), we will simulate this by generating random numbers in the node with **rank==0**. As an example, if the file contains **[0, 1, 0, 1]** and there are 4 processes, (with 0 for blue team and 1 for the green one), we could obtain:

**The file contains [0 1 0 1]**

**I am 0 and my team is blue**

**I am 2 and my team is blue**

**I am 3 and my team is green**

**I am 1 and my team is green**

# Exercise 6: position of the max [Focus on the memory management]

We want to know the position of the maximum in an array. The array is considered to be only in the process with **rank==0** but all processes know its length. A sequential example is in **max\_pos.py**. To compute this position in parallel, we will

1. Distribute the array using a single collective operation
2. Find the position and value of the local maximum
3. Send using a single collective operation these two values to the node with **rank==0**
4. Compute using these values the position of the overall maximum

# Exercise 7: Monte Carlo Simulation to compute π [Performance evaluation]

Surface of circle is **π \* r²** we will use this equation to compute π

1. Run the sequential code in **monte\_carlo.py**
2. Implement a parallel version of the same algorithm where
   1. All the processes know the total number **nb** of random draws to do
   2. There is no communication at the beginning
   3. There is only one communication at the end (no **send**/**receive**, only collectives)
   4. Only process of rank 0 prints the result
3. Print the value of **inside** for each process. Do not forget to initialize the random generator differently for each process.
4. Compare the time for 1, 2 and 4 processes with a constant total number of random draw
5. Same with higher value of **nb** taking at least 2 seconds for 1 process

# Exercise 8: Contrast stretching [Multiple collectives]

The goal is to stretch the contrast of images converted to grayscale.

1. Run the code in **stretching-base.py**

In this code, two different methods are used to stretch the contrast (**f\_one** and **f\_two**): Test both. You have to close the color picture to go to the next step. You have to close the gray picture to finish the execution.

1. Parallelize the code
   * Only process with **rank==0** loads and saves the image
   * Compute *max* and *min* in parallel
   * Use the *stretch functions* (**f\_one** and **f\_two**) in parallel
   * Make it so that processes with an even rank use **f\_one**, while processes with an odd rank use **f\_two**
   * Compare the time for 1, 2 and 4 processes

The result for the input image on the right (in color) should be like the one in grayscale for 4 processes.

# Exercise 9: Power of a matrix [Iterative use of collective operations]

We want to obtain the nth power of a square matrix. We assume that **n** and the matrix size **N** are known constants. The file **mult.py** contains an example of a sequential method. For information, you can manipulate numpy matrices in the same way as arrays. To create a matrix of size NxN containing only zeros, you can write **mat=np.zeros((N, N))**. The signature should be equal for your code and for the sequential code.

**Hint**:

use MPI to parallelize the matrix multiplication by dividing the computation of the matrix product across different processes.

Steps:

1. Broadcast the matrix A from the root process to all processes.
2. Distribute the work of computing parts of the matrix multiplication.
3. Use collective operations to combine the results from all processes (e.g., MPI.Reduce, MPI.Allreduce, etc.).
4. Gather the result and preint it out.

# Exercise 10: Maximum number of divisors [Load balancing, performance evaluation]

The number of divisors of a number nb is the number of positive integers i such as

nb % i == 0

The goal of this exercise is to compute the maximum number of divisors for all numbers below N, i.e. for all nb from 1 to N.

For each version of the code (question 1., 2. and 3. below), print also the time needed to do the computation (both total time, and time to run the loop without the collective operation).

1. Run the sequential program **primes.py**
2. Implement the parallel version with blocks of work (only using collective operations)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

1. Implement the parallel version by distributing the work one by one (only using collective operations)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

1. Compare the time of each version and explain

1. Understanding the sequential code**.**

**nb\_primes(n): Computes the number of divisors of n by iterating through all integers i from 1 to n and checking n%i==0.**

Main Loop:

* + Iterates over all values from 1 to N.
  + Computes the number of divisors for each value.
  + Tracks the maximum number of divisors encountered so far.

Output**: Prints the maximum number of divisors.**

**2. 1st question:** Parallelizing the code using MPI.

Divide the range of numbers [1,N] among the processes.

Compute the number of divisors locally on each process.

Use MPI collective operations to combine results and find the global maximum.

Total time: Measures the time from the start of the program to the end.

Loop-only time: Measures the time taken for the local computation, excluding MPI communication.

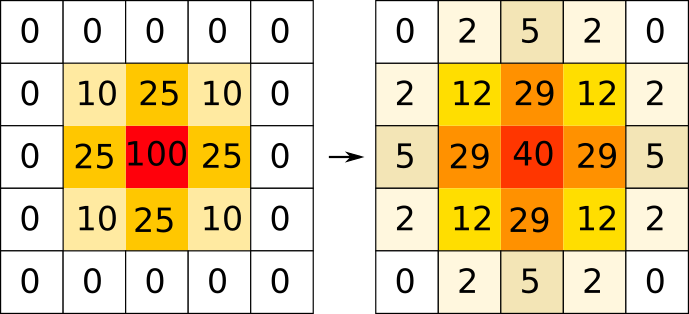
3. 2nd question: Parallelizing the code using MPI.

Each process computes the number of divisors for numbers assigned to it in a cyclic manner:

* + Process 0 handles 1,size+1,2×size+1,…,
  + Process 1 handles 2,size+2,2×size+2,…, and so on.

After local computation, the maximum value is determined using MPI collective operations.

# Exercise 11: Heat propagation [Point to point operation, performance evaluation]



To compute heat propagation in a matrix representing an object, we have to apply the following formula for all points:

Vk+1(i,j) = ( Vk(i-1, j) + Vk(i, j-1) + Vk(i, j) + Vk(i+1, j) + Vk(i, j+1) ) / 5

The code in **heat.py** generates such a matrix and makes it evolve over several iterations (representing time).

Modify the code to make it run in parallel. Check the signature and plot time in function of the number of processes. You can use Send, Isend, Recv and Irecv functions for this sole exercise.

# Exercise 12: C version [Generalization to another other language]

Compile and run all the code in the **Labs/C** directory. For example, to compile and run scatter.c:

1. **mpicc scatter.c -o scatter**
2. **mpirun -n 4 scatter**

Modify **scatter.c** to run on any number of nodes as the scatter code can only run on 4 nodes.

Write a C version of Position of the max (exercise 6).

compare perf (time etc).