Meluxina: Accelerating Python with Multiple Nodes with MPI for Python

Oscar J. Castro-Lopez

Research Associate

University of Luxembourg

May 2025

UNIVERSITÉ DU LUXEMBOURG

1. Introduction
2. Description of mpi4py
3. Launch mpi4py with SLURM
4. Basic examples
5. Use case: parallel hyperparameter search
6. Use case: parallel string matching
7. Performance Considerations, Best Practices, and Summary /Takeaways

Introduction

Objective

- Leverage multiple compute nodes to accelerate Python workloads using mpi4py.

Motivation

- Python is widely used for ML, simulation, and data processing — but it's limited by the GIL.
- Single-node, multi-core use is often not enough for large-scale problems.
- MPI (Message Passing Interface) enables true parallelism across multiple machines.

Why mpi4py?

- Seamless integration with existing Python code
- Portable across HPC clusters and cloud
- Ideal for distributed Grid Search, Simulations, ML Training, etc.

Goal of This Training

- Understand how to use mpi4py to run Python code across multiple nodes, not just multiple cores.

Why to use Python for HPC?

Simplicity and Productivity:

- Easy to write, read, and debug.
- Ideal for rapid prototyping of scientific workflows and models.

Rich scientific (and in general) ecosystem.

- Powerful libraries like Numpy, Scipy, Pandas, Scikit-learn, Pytorch.
- Specialized HPC tools, mpi4py, Dask, numba, Cython, CuPy, PyCUDA.

- Interoperability with HPC languages: C, C++, Fortran.
- Community and Ecosystem
- Widely used in Research
- **Dominance in Data Science and AI**

Why Multi-Node Parallelism Matters

- Growing data and model sizes demand more compute power.
- Single-node CPU/memory often becomes a bottleneck.
- Multi-node execution scales horizontally, enabling faster processing.

Python's Limitations

- GIL (Global Interpreter Lock): only one thread executes Python bytecode at a time.
- **threading** is mostly useful for I/O-bound tasks.
- **multiprocessing** works across cores but is limited to one machine.

Why MPI and mpi4py?

- MPI = Message Passing Interface, the standard for distributed computing.
- mpi4py: Python bindings for MPI, allowing parallelism across multiple nodes.
- Well-suited for simulations, optimization, ML tasks, etc.

Single-node: Shared Memory

- All processes/threads access the same memory space.
- Easier data sharing but limited by the hardware of one machine.

Multi-node: Distributed Memory

- Each process runs independently on a separate machine with its own memory.
- Requires explicit communication between processes.

Communication Models in MPI

- Point-to-Point: Direct send/receive between two processes.
  - comm.send(), comm.recv()
- Collective Communication: Involves multiple processes.
  - comm.bcast(), comm.scatter(), comm.gather(), comm.reduce()

- **Key Concept: Think in terms of processes and messages, not threads and shared state.**

mpi4py for Multinode computing

UNIVERSITÉ DU LUXEMBOURG

Installation and code example

- Example on how to install mpi4py:

```
pip install mpi4py
```

Requires an MPI implementation:

- OpenMPI, MPICH, or Intel MPI.

```python
from mpi4py import MPI

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

print(f"Hello from rank {rank} out of {size}")
```

- Running the Code:

```
mpirun -n 4 python script.py
```

- **Cluster (Multiple Nodes)** (with SLURM or similar):

```
srun -n 4 python script.py
```

- **rank** = process ID; **size** = total number of processes.

What is MPI? (Message Passing Interface)

- **Standard for distributed computing** across multiple nodes/machines
- Enables **parallel programs** to communicate via **messages**, not shared memory
- Designed for **scalability** — from a few nodes to thousands

Typical Use in Supercomputers:

- Core tool for scientific simulations, weather models, fluid dynamics, etc.
- Each process runs independently, and exchanges data only as needed.
- Used on HPC clusters with high-speed interconnects (e.g., Infiniband).
- Compatible with C, C++, Fortran — and now Python via mpi4py.

MPI for Python provides Python bindings for the Message Passing Interface (MPI) standard, allowing Python applications to exploit multiple processors on workstations, clusters and supercomputers.

Supports:

- Point-to-point and collective communication of any *picklable* Python object.
- Efficient communication of Python objects exposing the Python buffer interface (e.g. NumPy arrays and builtin bytes/array/memoryview objects).

Link: https://mpi4py.readthedocs.io/en/stable/index.html

Communicators:

- A group of processes that can communicate with each other
- Default: MPI.COMM_WORLD – includes all processes

Ranks and Size:

- rank = comm.Get_rank() → ID of the current process
- size = comm.Get_size() → Total number of processes in the communicator

Every process runs the same code — only rank differentiates behavior

Point-to-Point:

- Direct communication between two processes:
  - `comm.send(data, dest=1)` → Send data to process 1
  - `data = comm.recv(source=0)` → Receive data from process 0
- Useful for fine-grained control, async behavior

Collective Operations:

- Involve all processes in the communicator
- Examples:
  - `comm.bcast(data, root=0)` → broadcast from root to all
  - `comm.scatter(data, root=0)` → distribute chunks to each process
  - `comm.gather(data, root=0)` → collect results to root
  - `comm.reduce(op=MPI.SUM, root=0)` → reduce (e.g., sum) across processes

Efficient for global coordination and data aggregation

Communication of generic Python objects

- Methods with all-lower-case names, like:
  - `comm.send, comm.recv, comm.bcast, comm.scatter, comm.gather.`
- An object to be sent is passed as a parameter to the communication call, and the received object is simply the return value.
- Collective calls expect a single value or a sequence of comm.size elements at the root or all process. They return a single value, a list of comm.size elements, or None.

Communication of buffer-like objects

- Methods starting with an upper-case letter, like:
  - `comm.Send, comm.Recv, comm.Bcast, comm.Scatter, comm.Gather.`
- In general, buffer arguments to these calls must be explicitly specified by using a list/tuple.

Data Types and mpi4py

- **Built-in types**: int, float, str, list, etc. → work out-of-the-box with automatic pickling.
- **Numpy arrays**: Use mpi4py-optimized methods for performance.
  - `comm.Bcast(array, root=0)` avoids pickling.
- **Pandas DataFrames**: Must be serialized (e.g., `to_dict()` or `to_numpy()`).

Tips for Efficient Communication

- Avoid Python objects when possible; use numpy for numerical data.
- Minimize the number and size of messages.
- Test with small data before scaling.

Performance note: Prefer native buffers (e.g., numpy) for large data to avoid pickle overhead.

Launching mpi4py programs in a supercomputer with SLURM

UNIVERSITÉ DU LUXEMBOURG

What is SLURM?

- **SLURM** (Simple Linux Utility for Resource Management) is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters.

- On a supercomputer, user applications are run through a job scheduler, also called a batch scheduler or queueing system.

- Users can run their applications in two essential ways:
  - **batch mode**: users submit a script 'launcher' file to SLURM, the commands and applications inside are run by SLURM.
  - **dev mode (interactive)**: users get connected by SLURM to a (set of) computing nodes directly and can run their applications interactively.

Connect to Meluxina and get the material

- First step is to connect to Meluxina:

```
ssh meluxina
```

Enter passphrase if needed.

- Copy the content of the course:

```
cp –r /project/home/p200884/python_mpi4py .
```

- Enter the folder with the scripts:

```
cd python_mpi4py
```

- Contains two folders, one with data, another with code examples. In the folder there is list of SLURM scripts.

- **Lmod** allow users to easily load, unload, and switch between different software packages and versions on a High-Performance Computing (HPC) cluster

- Load a module:

```
module load <module_name>
```

- Unload a module:

```
module unload <module_name>
```

- Search for modules:

```
module spider <keyword>
```

- In this way we do not need to install everything!
- More information on:
  - https://lmod.readthedocs.io/en/latest/010_user.html

0: Basic Batch Script in SLURM


```
#!/bin/bash -l
#SBATCH --account=p200884                    # project account
#SBATCH --job-name=0_mpi_simple              # Job name
#SBATCH --partition=cpu                      # Cluster partition
#SBATCH --qos=short                          # SLURM qos
#SBATCH --time=0-1:00:00                     # Time to run the Job(HH:MM:SS)
#SBATCH --nodes=1                            # Number of nodes
#SBATCH --ntasks-per-node=2                  # Number of tasks per node
#SBATCH --cpus-per-task=128                  # CORES per task
#SBATCH --output=0_mpi_simple.txt            # name of the file to save the output

# Load Modules
module load env/release/2024.1
module load Python
module load mpi4py/4.0.1-gompi-2024a

cd code/
srun -n 2 python -u 0_mpi_simple.py
```

- To launch a SLURM batch job: `sbatch 0_batch_mpi4py_simple.sh`

0: Mpi4py simple example Point-to-Point

Steps:

1. Send data from process 0 to process 1.
2. Process 1 receives a value, do some computation and returns a result to process 0.

```python
from mpi4py import MPI

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

if rank == 0:
    data = 42
    comm.send(data, dest=1)
    result = comm.recv(source=1)
    print(f"Rank 0 received result: {result}")

elif rank == 1:
    received = comm.recv(source=0)
    print(f"Rank 1 received value: {received}")
    computed = received * 2
    comm.send(computed, dest=0)
```

Launch the script:

```
sbatch 0_batch_mpi4py_simple.sh
```

1: Basic Batch Script in SLURM


```
#!/bin/bash -l
#SBATCH --account=p200266                    # project account
#SBATCH --job-name=1_mpi_simple              # Job name
#SBATCH --partition=cpu                      # Cluster partition
#SBATCH --qos=short                          # SLURM qos
#SBATCH --time=0-1:00:00                     # Time to run the Job(HH:MM:SS)
#SBATCH --nodes=1                            # Number of nodes
#SBATCH --ntasks-per-node=128                 # Number of tasks per node
#SBATCH --cpus-per-task=2                    # CORES per task
#SBATCH --output=1_mpi_simple.txt            # name of the file to save the output

# Load Modules
module load env/release/2024.1
module load Python
module load mpi4py/4.0.1-gompi-2024a

cd code/
srun -n 128 python -u 1_mpi_simple.py
```

- To launch a SLURM batch job: `sbatch 1_batch_mpi4py_simple.sh`

1: Mpi4py simple example Point-to-Point

Steps:

1. Send data from process 0 to other processes.
2. Processes different than 0 receive a value, do some computation and returns a result to process 0.

```python
from mpi4py import MPI

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

if rank == 0:
    data = 42
    for i in range(1, size):
        comm.send(data+i, dest=i)
    for i in range(1, size):
        result = comm.recv(source=i)
        print(f"Rank 0 received result: {result} from rank {i}")
else:
    received = comm.recv(source=0)
    print(f"Rank {rank} received value: {received}")
    computed = received * 2
    comm.send(computed, dest=0)
```

Launch the script:

```
sbatch 1_batch_mpi4py_simple.sh
```

In a SLURM batch script we need to setup the following parameters to allocate resources for our MPI jobs:

Parameter (example)	Value	Meaning
--nodes=1	1 Node	The job will use **one compute node**
--ntasks-per-node=2	2 Tasks/Node	**Two MPI processes** will run per node
--cpus-per-task=128	128 Cores/Task	Each MPI process gets **128 CPU cores**
This will launch a total of two processes (rank 0 and 1).		

Parameter (example)	Value	Meaning
--nodes=2	2 Nodes	The job will use **two compute nodes**
--ntasks-per-node=128	128 Tasks/Node	**128 MPI processes** will run per node
--cpus-per-task=2	2 Cores/Task	Each MPI process gets **2 CPU cores**
This will launch a total of 256 processes (rank 0, 1, 2, … and 255).		

2: Mpi4py simple example Collective Operations: Parallel sum

Steps:

1. Scatter data chunks to each process.
2. Each process computes a local sum.
3. Reduce all local sums to the master (rank 0).

To launch this script:

```
sbatch 2_batch_mpi4py_sum.sh
```

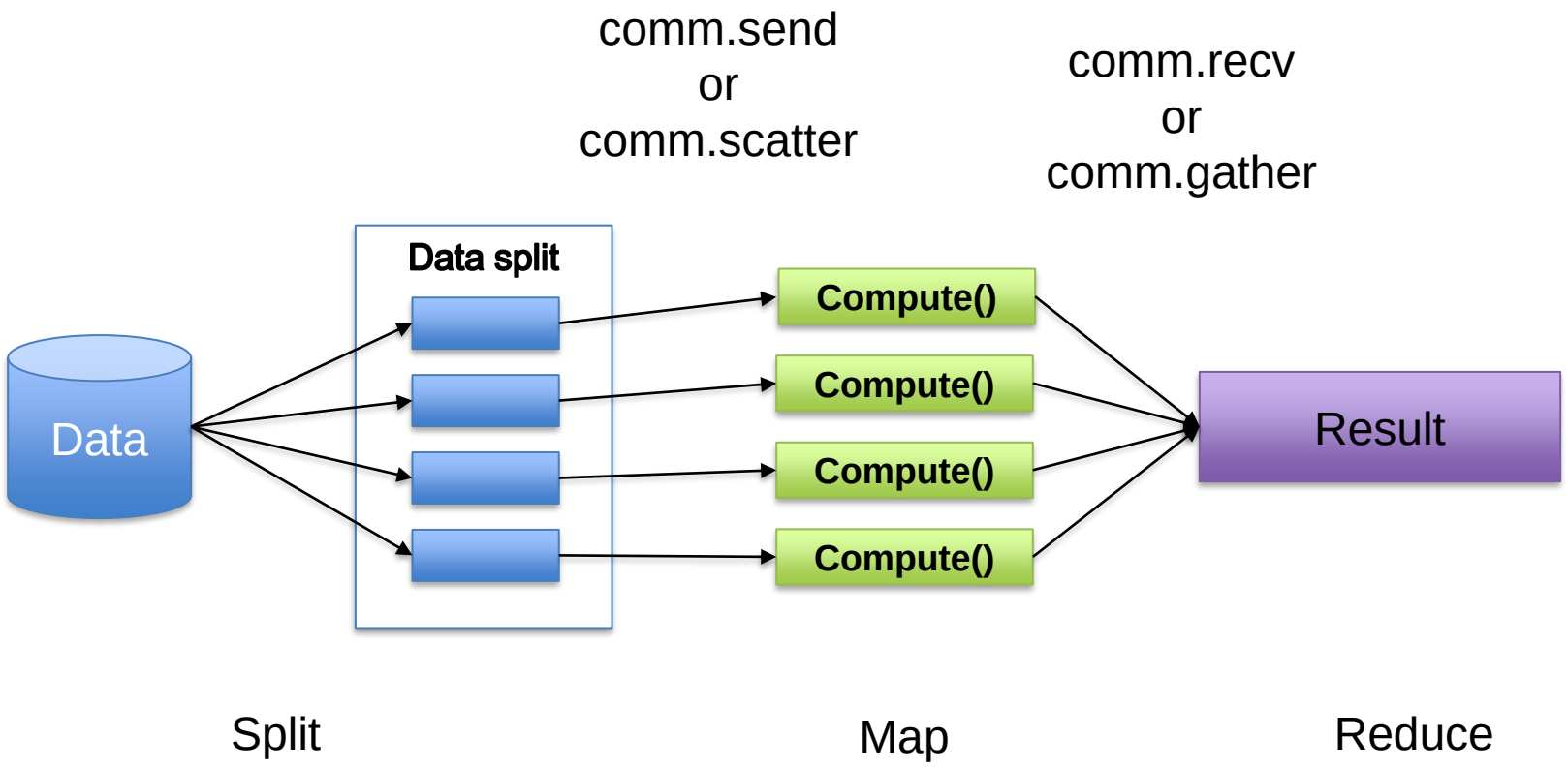
```python
from mpi4py import MPI

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

# Step 1: Create data only on root
if rank == 0:
    data = list(range(100))
    chunks = [data[i::size] for i in range(size)]
else:
    chunks = None

# Scatter chunks
local_data = comm.scatter(chunks, root=0)

# Step 2: Local sum
local_sum = sum(local_data)

# Step 3: Reduce to total sum
total = comm.reduce(local_sum, op=MPI.SUM, root=0)

if rank == 0:
    print(f"Total sum is: {total}")
```

MPI Workflow at a High Level

1. **Initialization**
   - Import mpi4py, initialize the communicator (MPI.COMM_WORLD)
   - Determine rank (process ID) and size (total number of processes)
2. **Broadcast Configuration / Parameters**
   - Rank 0 defines input parameters (e.g., model settings, filenames)
   - Use `comm.bcast()` to send these to all other processes
3. **Scatter Data**
   - Rank 0 splits a dataset or workload into chunks
   - Use `comm.scatter()` to distribute pieces to each process
4. **Local Computation**
   - Each process independently computes on its chunk (e.g., train model, simulate, process data)
5. **Gather or Reduce Results**
   - Use `comm.gather()` to collect results at Rank 0 (e.g., predictions, metrics)
   - Or use `comm.reduce()` or `comm.allreduce()` for aggregated results (e.g., sum, mean, min)
6. **Postprocessing (at Rank 0)**
   - Combine results, select best outcome, save output, etc.
7. **Finalize (optional but clean)**
   - `MPI.Finalize()` (optional when using mpi4py, handled on exit)

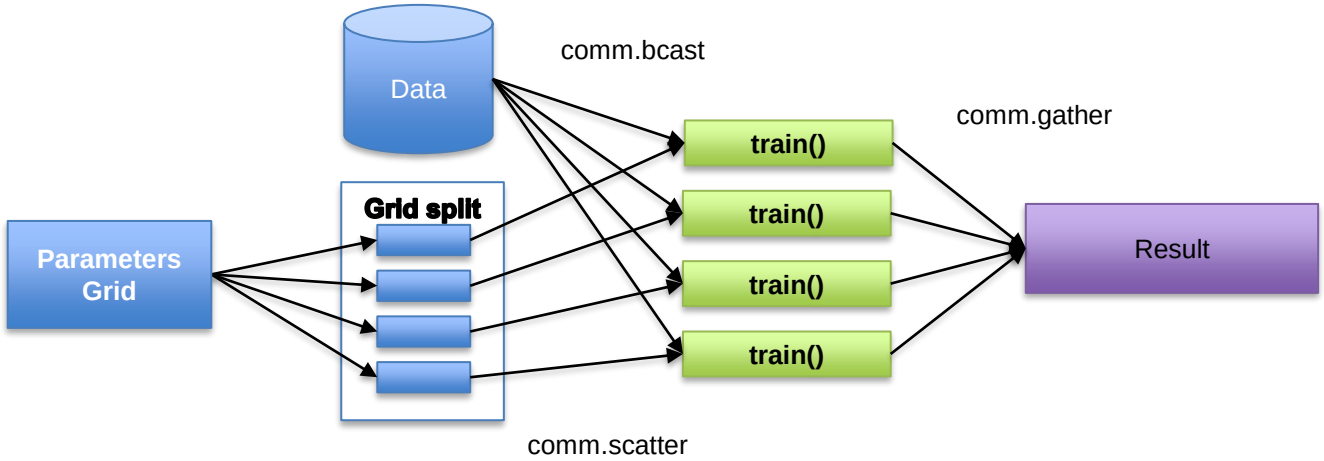
Typical cases to use mpi4py (or mpi in general)

- “Map reduce” style workloads:

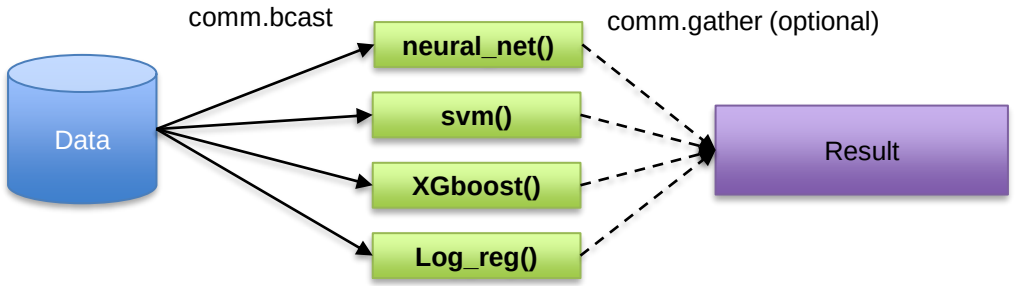


Typical cases to use mpi4py (or mpi in general)

- Parameter grid search / Hyperparameter Tuning:



- Train ML models in parallel



The script 3_mpi_np.py does the following:

1. Creates a Python list and a NumPy array on Rank 0.
2. Broadcasts and scatters both using:
   - Standard communication for the list.
   - Buffered communication for the NumPy array.
3. Performs a dummy computation (squaring the values).
4. Gathers the results.
5. Compares the execution time.

Note: check that the data is divisible by the number of processes.

To launch this script:

```
sbatch 3_batch_mpi4py_np.sh
```

Use case: hyperparameter search

Manual hyper parameter search

- Goal: Find the best hyperparameters to train an XGBoost model for credit score classification.
- Challenge:
  - Training one model takes approximately one minute.
  - Total parameter combinations: 216.
  - Without optimization: ~216 minutes (over 3.5 hours) to complete the grid search.
  - Objective: Accelerate the search process using High-Performance Computing (HPC).

- Important context:
  - XGBoost internally uses multi-core CPU parallelism.
  - Dataset size: relatively small (~40 MB).
  - Key focus:
    - Distribute different model trainings across multiple nodes/processes.
    - Not parallelize individual model training (XGBoost already does that).

  - Run `sbatch 4_batch_hyper_boost.sh`

4: Manual hyper parameter search

- To leverage XGBoost's multicore capabilities, each MPI process is assigned a fair number of CPU cores.
- All MPI processes:
  - Independently read the dataset.
  - Perform identical data preprocessing before training.
- Hyperparameter combinations are generated and scattered across MPI processes.
- Each process:
  - Trains and evaluates a subset of models independently.
  - All training results are gathered back to process 0 (also called rank 0).
- Results:
  - Printed and saved to a CSV file for further analysis.

Use case string matching

Parallel String Matching with Levenshtein Ratio

- Goal: Find similar strings between two DataFrames (or two list of strings).
- Challenge:
  - $O(n^2)$ complexity (all pairwise comparisons)
  - Slow with Python's single-threaded execution
  - Example: 10K x 10K = **100M comparisons**

Levenshtein distance is a string metric for measuring the difference between two sequences.

- Solution with “small” data:
  - Each process computes the Levenshtein ratio of one left string against all right strings.
  - Each process return a list of valid pairs.
  - Aggregate results.

- Note that this approach won’t work with very large datasets because of memory issues.

- Run the following script: `sbatch 5_sbatch_py_leven.sh`
- The script is going to do the following:
  1. Load required modules.
  2. If virtual environment does not exist:
     - Create a new virtual environment.
     - Enter the virtual environment.
     - Install Levenshtein package.
  3. If virtual environment already exists, then it enters the virtual environment.
  4. Run sequential python implementation.

- The output is stored in the file: `5_python_leven.txt.`

- There could be many ways to parallelize the code.

- The sample code does the following:
  1. Rank 0 loads and broadcast the data to the rest of the processes.
  2. All processes compute the Levenshtein ratio between a chunk of the left strings against the whole list of right strings.
  3. All results are gathered in rank 0.
  4. Rank 0 prints the results.

- Launch the following script:

```
sbatch 6_batch_mpi4py_leven.sh
```

- Launch the following script afterwards:

```
sbatch 7_batch_mpi4py_leven.sh
```

- The benefit comes only if data/computations are big enough.
- Minimizing communication between processes.
- Make it more challenging by increasing the size of the data and varying mpi configuration:
- Launch the following script:

```
sbatch 8_batch_mpi4py_leven.sh
```

- Launch the following script afterwards:

```
sbatch 9_batch_mpi4py_leven.sh
```

Summary results (offline)

Version (#)	Ratios to compute	Nodes	Tasks per node	CPUs per task	Elapsed time in seconds
Python (5)	2,975,625	N/A	N/A	N/A	144.38
Python*	297,562,500	N/A	N/A	N/A	14,438.00
Mpi4py (6)	2,975,625	1	128	2	6.22
Mpi4py (7)	2,975,625	2	256	2	16.67
Mpi4py (8)	297,562,500	2	256	2	68.31
Mpi4py (8)	1,190,250,000	2	256	2	263.87
Mpi4py (9)	297,562,500	5	256	2	28.82
Mpi4py (9)	1,190,250,000	5	256	2	112.78

Conclusion remarks

Load Balancing

- Distribute work evenly to avoid idle processes.
- Use dynamic or balanced scheduling where tasks vary in duration.

Communication Overhead

- Communication cost grows with the number of processes.
- Minimize frequency and volume of data exchange.

Scalability and Bottlenecks

- Amdahl’s Law: Speedup is limited by serial portions of code.
- Collective operations can become bottlenecks at scale.
- Measure and profile regularly.

Debugging MPI Programs

- Use print(f"Rank {rank}: message") or logging with ranks.
- Tools: gdb, valgrind, TotalView, Intel VTune, etc.

Efficient I/O

- Avoid multiple processes reading large files independently.
- Use parallel I/O libraries or let rank 0 handle file access.

Avoiding Common Pitfalls

- Do not scatter/broadcast None or inconsistent data types.
- All ranks must call collective ops (e.g., Gather, Broadcast).
- Ensure deterministic behavior — avoid race conditions.

When to Use Multi-node Parallelism

- Training or simulations too large for a single node.
- Tasks can be clearly divided among processes.

Benefits of mpi4py

- Brings powerful MPI semantics to Python.
- Easy to prototype parallel logic.
- **Interoperable with NumPy and scientific Python stack.**

Limitations

- GIL limits threading within a rank.
- Less efficient for very fine-grained tasks compared to C/C++.

Resources to Go Further

- mpi4py documentation **https://mpi4py.readthedocs.io/en/stable/index.html**

Python is production-ready for serious parallel work — and fun to use!.

Thank you!

oscar.castro@uni.lu

UNIVERSITÉ DU LUXEMBOURG