LESSON NAME

Intro

Prerequisites

- FIXME
- …
- …

20 min | filename

Introduction to HPC

Questions

- What is High-Performance Computing (HPC)?
- Why do we use HPC systems?
- How does parallel computing make programs faster?

Objectives

- Define what High-Performance Computing (HPC) is.
- Identify the main components of an HPC system.
- Describe the difference between serial and parallel computing.
- Run a simple command on a cluster using the terminal.

High-Performance Computing (HPC) refers to using many computers working together to solve complex problems faster than a single machine could. HPC is widely used in fields such as climate science, molecular simulation, astrophysics, and artificial intelligence.

This lesson introduces what HPC is, why it matters, and how researchers use clusters to perform large-scale computations.

Skip to content

What is HPC?

HPC systems, often called *supercomputers* or *clusters*, are made up of many computers (called **nodes**) connected by a fast network. Each node can have multiple cores which are **CPUs** (and sometimes **GPUs**) that run tasks in parallel.

Typical HPC Components

Component	Description
Login node	Where you connect and submit jobs
Compute nodes	Machines where your program actually runs
Scheduler	Manages job submissions and allocates resources (e.g. SLURM)
Storage	Shared file system accessible to all nodes

Single core performance optimization

Pure-Python loops are slow because each iteration runs in the Python interpreter. NumPy pushes work into optimized native code (C/C++/BLAS), drastically reducing overhead. Below we compare a Python for loop with NumPy vectorized operations and discuss tips for fair, single-core measurements.

Practice making Python faster on a single CPU.

Copy and paste this code

Skip to content

```python
import os
# (Optional safety if you run this inside Python, must be set BEFORE importing numpy)
os.environ.setdefault("OMP_NUM_THREADS", "1")
os.environ.setdefault("OPENBLAS_NUM_THREADS", "1")
os.environ.setdefault("MKL_NUM_THREADS", "1")
os.environ.setdefault("NUMEXPR_NUM_THREADS", "1")

import numpy as np
from time import perf_counter

def timeit(fn, *args, repeats=5, warmup=1, **kwargs):
    # warmup
    for _ in range(warmup):
        fn(*args, **kwargs)
    # timed runs
    tmin = float("inf")
    for _ in range(repeats):
        t0 = perf_counter()
        fn(*args, **kwargs)
        dt = perf_counter() - t0
        tmin = min(tmin, dt)
    return tmin

# Problem size
N = 10_000_000  # 10 million elements

# Test data (contiguous, fixed dtype)
a = np.random.rand(N).astype(np.float64)
b = np.random.rand(N).astype(np.float64)

# --- 1) Pure-Python loop sum ---
def py_sum(x):
    s = 0.0
    for v in x:          # per-element Python overhead
        s += v
    return s

# --- 2) NumPy vectorized sum ---
def np_sum(x):
    return x.sum()      # dispatches to optimized C/BLAS

# --- 3) Elementwise add then sum (Python loop) ---
def py_add_sum(x, y):
    s = 0.0
    for i in range(len(x)):
        s += x[i] + y[i]
    return s

# --- 4) Elementwise add then sum (NumPy, no temporaries) ---
def np_add_sum_no_temp(x, y):
    # np.add.reduce avoids allocating x+y temporary
    return np.add.reduce([x, y])  # equivalent to sum stacks; see alt below

# Alternative that’s typically fastest and clearer:
def np_add_sum_fast(x, y):
                y).sum()  # may allocate a temporary; fast on many BLAS builds


print("Timing on single core (best of 5 runs):")
t_py_sum   = timeit(py_sum, a)
```

Skip to content

```
t_np_sum   = timeit(np_sum, a)
t_py_add   = timeit(py_add_sum, a, b)
t_np_add   = timeit(np_add_sum_fast, a, b)

print(f"Python for-loop sum:          {t_py_sum:8.4f} s")
print(f"NumPy vectorized sum:         {t_np_sum:8.4f} s")
print(f"Python loop add+sum:          {t_py_add:8.4f} s")
print(f"NumPy vectorized add+sum:     {t_np_add:8.4f} s")
```

Execute it and following let us verify the effect on the following modifications:

1. Run the timing script with N = 1_000_000, 5_000_000, 20_000_000.
2. Try float32 vs float64.
3. Switch (a + b).sum() to np.add(a, b, out=a); a.sum() and compare.

Practical tips for single-core speed

- Prefer vectorization: Use array ops (+, , *.sum(), .dot(), np.mean, np.linalg.*) rather than per-element Python loops.
- Control temporaries: Expressions like (a + b + c).sum() may create temporaries. When memory is tight, consider in-place ops (a += b) or reductions (np.add(a, b, out=a); np.add.reduce([…])).
- Use the right dtype: float64 is standard for numerics; float32 halves memory traffic and can be faster on some CPUs/GPUs (but mind precision).
- Preallocate: Avoid growing Python lists or repeatedly allocating arrays inside loops.
- Minimize Python in hot paths: Move heavy math into NumPy calls; keep Python for orchestration only.
- Benchmark correctly: Use large N, pin threads to 1 for fair single-core tests, and report the best of multiple runs after a warmup.

–

Parallel Computing

High-Performance Computing relies on **parallel computing**, splitting a problem into smaller parts that can be executed *simultaneously* on multiple processors.

Instead of running one instruction at a time on one CPU core, parallel computing allows you to run many instructions on many cores or even multiple machines at once.

Parallelism can occur at different levels:

Skip to content

- **e CPU** (multiple cores)
- **le CPUs** (distributed nodes)
- **On specialized accelerators** (GPUs or TPUs)

Shared-Memory Parallelism

In **shared-memory** systems, multiple processor cores share the same memory space. Each core can directly read and write to the same variables in memory.

This is the model used in:

- Multicore laptops and workstations
- *Single compute nodes* on a cluster

Programs use **threads** to execute in parallel (e.g., with OpenMP in C/C++/Fortran or **multiprocessing in Python**).

Keypoints

Advantages:

- Easy communication between threads (shared variables)
- Low latency data access

Limitations:

- Limited by the number of cores on one machine
- Risk of race conditions if data access is not synchronized

Practice with threaded parallelism in Python

Example:

```
from multiprocessing import Pool

def square(x):
    return x * x

if __name__ == "__main__":
    with Pool(4) as p:
        result = p.map(square, range(8))
    print(result)
```

Distributed-Memory Parallelism

In distributed-memory systems, each processor (or node) has its own local memory. Processors communicate by passing messages over a network.

used when a computation spans multiple nodes in an HPC cluster.

Programs written with MPI (Message Passing Interface) use explicit communication. Below is an example using the Python library `mpi4py` that implements MPI functions in Python

Practice with a simple MPI program

```
# hello_mpi.py
from mpi4py import MPI

# Initialize the MPI communicator
comm = MPI.COMM_WORLD

# Get the total number of processes
size = comm.Get_size()

# Get the rank (ID) of this process
rank = comm.Get_rank()

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

# MPI is automatically finalized when the program exits,
# but you can call MPI.Finalize() explicitly if you prefer
```

For now, do not worry about understanding this code, we will see `mpi4py` in detail later.

Keypoints

Advantages:

- Scales to thousands of nodes
- Each process works independently, avoiding memory contention

Limitations:

- Requires explicit communication (send/receive)
- More complex programming model
- More latency, requires minimizing movement of data.

Hybrid Architectures: CPU, GPU, and TPU

Modern High-Performance Computing (HPC) systems rarely rely on CPUs alone.
They are **hybrid architectures**, combining different types of processors, typically **CPUs**, **GPUs**, and increasingly **TPUs**, to achieve both flexibility and high performance.

CPU: The General-Purpose Processor

Central Processing Units (CPUs) are versatile processors capable of handling a wide range of tasks.

Skip to content small number of powerful cores optimized for complex, sequential operations

CPUs are responsible for:

- Managing input/output operations
- Coordinating data movement and workflow
- Executing serial portions of applications

They excel in **task parallelism**, where different cores perform distinct tasks concurrently.

GPU: The Parallel Workhorse

Graphics Processing Units (GPUs) contain thousands of lightweight cores that can execute the same instruction on many data elements simultaneously.
This makes them ideal for **data-parallel** workloads, such as numerical simulations, molecular dynamics, and deep learning.

GPUs are optimized for:

- Large-scale mathematical computations
- Highly parallel tasks such as matrix and vector operations

Common GPU computing frameworks include CUDA, HIP, OpenACC, and SYCL.

GPUs provide massive computational throughput but require explicit management of data transfers between CPU and GPU memory.
They are now a standard component of most modern supercomputers.

TPU: Specialized Processor for Tensor Operations

Tensor Processing Units (TPUs) are specialized hardware accelerators designed for tensor and matrix operations, the building blocks of deep learning and AI.
Originally developed by Google, TPUs are now used in both cloud and research HPC environments.

TPUs focus on **tensor computations** and achieve very high performance and energy efficiency for machine learning workloads.
They are less flexible than CPUs or GPUs but excel in neural network training and inference.

Python in High-Performance Computing

Python has become one of the most widely used languages in scientific computing due to its simplicity, readability, and extensive ecosystem of numerical libraries.
Although Python itself is interpreted and slower than compiled languages such as C or Fortran, it now provides a mature set of tools that allow code to **run efficiently on modern HPC architectures**.

These tools map directly to the three fundamental forms of parallelism introduced earlier:

HPC Parallelism Type	Hardware Context	Python Solutions
Shared-memory parallelism	Multicore CPUs within a node	NumPy, Numba, Pythran
Distributed-memory parallelism	Multiple nodes across a cluster	mpi4py
Accelerator parallelism	GPUs and TPUs	CuPy, JAX, Numba (CUDA)

In practice, these technologies allow Python programs to scale from a single core to thousands of nodes on hybrid CPU–GPU systems.

Shared-Memory Parallelism (Multicore CPUs)

Shared-memory parallelism occurs within a single compute node, where all CPU cores access the same physical memory.
Python supports this level of performance primarily through **compiled numerical libraries** and **JIT (Just-In-Time) compilation**, which transform slow Python loops into efficient native machine code.

NumPy: Foundation of Scientific Computing

NumPy provides fast array operations implemented in C and Fortran.
Its vectorized operations and BLAS/LAPACK backends **automatically** exploit shared-memory parallelism through optimized linear algebra kernels.
Although users write Python, most computations occur in compiled native code.

Pythran: Static Compilation of Numerical Python Code

Pythran compiles numerical Python code — particularly code using NumPy — into optimized

Skip to content

It can automatically parallelize loops using **OpenMP**, enabling true multicore utilization without manual thread management.

Key strengths:

- Converts array-oriented Python functions into C++ for near-native speed
- Supports automatic OpenMP parallelization for CPU cores
- Integrates easily into existing Python workflows

Pythran is well-suited for simulations or kernels that need to exploit multiple cores on a node.

Numba: JIT Compilation for Shared and Accelerator Architectures

Numba uses LLVM to JIT-compile Python functions into efficient machine code at runtime.
On multicore CPUs, Numba can parallelize loops using OpenMP-like constructs; on GPUs, it can emit CUDA kernels (see below).

Main advantages:

- Minimal syntax changes required
- Explicit parallel decorators for CPU threading
- Compatible with NumPy arrays and ufuncs

Together, NumPy, Pythran, and Numba enable Python to fully exploit shared-memory parallelism.

Distributed-Memory Parallelism (Clusters and Supercomputers)

At large scale, HPC systems use **distributed memory**, where each node has its own local memory and must communicate explicitly.
Python provides access to this level of parallelism through **mpi4py**, a direct interface to the standard MPI library.

mpi4py: Scalable Distributed Computing with MPI

mpi4py enables Python programs to exchange data between processes running on different nodes using MPI.
It provides both point-to-point and collective communication primitives, identical in concept to those used in C or Fortran MPI applications.

Key features:

- Works seamlessly with NumPy arrays (zero-copy data transfer)
- Supports all MPI operations (send, receive, broadcast, scatter, gather, reduce)
- Skip to content h job schedulers such as SLURM or PBS

With `mpi4py`, Python can participate in large-scale distributed-memory simulations or data-parallel tasks across thousands of cores.

Accelerator-Specific Parallelism (GPUs and TPUs)

Modern HPC nodes increasingly include **GPUs** or **TPUs** to accelerate numerical workloads. Python offers several mature libraries that interface directly with these accelerators, providing high-level syntax while executing low-level parallel kernels.

CuPy: GPU-Accelerated NumPy Replacement

CuPy mirrors the NumPy API but executes array operations on GPUs using CUDA (NVIDIA) or ROCm (AMD).
Users can port existing NumPy code to GPUs with minimal changes, gaining massive speedups for large, data-parallel computations.

Highlights:

- NumPy-compatible array and linear algebra operations
- Native support for multi-GPU and CUDA streams
- Tight integration with deep learning and simulation frameworks

JAX: Unified Array Computing for CPUs, GPUs, and TPUs

JAX combines automatic differentiation and XLA-based compilation to execute Python functions efficiently on CPUs, GPUs, and TPUs.
It is particularly well-suited for scientific machine learning and differentiable simulations.

Key strengths:

- Just-In-Time (JIT) compilation via XLA
- Transparent execution on accelerators (GPU, TPU)
- Built-in vectorization and automatic differentiation

JAX provides a single high-level API for heterogeneous HPC nodes, seamlessly handling hybrid CPU–GPU–TPU workflows.

Summary: Python Across HPC Architectures

Python can now leverage **all layers of hybrid HPC architectures** through specialized libraries:

Architecture	Parallelism Type	Typical Python Tools	Example Use Cases
Multicore CPUs	Shared memory	NumPy, Pythran, Numba	Numerical kernels, vectorized math
Clusters	Distributed memory	mpi4py	Large-scale simulations, domain decomposition
GPUs / TPUs	Accelerator parallelism	CuPy, JAX, Numba (CUDA)	Machine learning, dense linear algebra

Together, these tools allow Python to serve as a *high-level orchestration language* that transparently scales from a single laptop core to full supercomputing environments — integrating shared-memory, distributed-memory, and accelerator-based parallelism in one ecosystem.

Keypoints

- Python's ecosystem maps naturally onto hybrid HPC architectures.
- **NumPy, Numba, and Pythran** exploit shared-memory parallelism on multicore CPUs.
- **mpi4py** extends Python to distributed-memory clusters.
- **CuPy and JAX** enable acceleration on GPUs and TPUs.
- These libraries allow researchers to combine high productivity with near-native performance across all layers of HPC systems.

Introduction to MPI with Python (mpi4py)

Questions

- What is MPI, and how does it enable parallel programs to communicate?
- How does Python implement MPI through the `mpi4py` library?
- What are point-to-point and collective communications?
- How does `mpi4py` integrate with NumPy for efficient data exchange?

Objectives

- Understand the conceptual model of MPI: processes, ranks, and communication.
- [illegible] veen point-to-point and collective operations.

Skip to content

- [illegible] NumPy arrays act as communication buffers in `mpi4py`.
- See how `mpi4py` bridges Python and traditional HPC concepts.

What Is MPI?

MPI (Message Passing Interface) is a standardized programming model for communication among processes that run on **distributed-memory systems**, such as HPC clusters.

In a distributed-memory system, each compute node (or process) has its **own local memory**. Unlike shared-memory systems, where threads can directly read and write to a common address space, distributed processes **cannot directly access each other's memory**.
To collaborate, they must explicitly **send and receive messages** containing the data they need to share.

Independent Processes and the SPMD Model

When you run an MPI program, the system launches **multiple independent processes**, each running its **own copy** of the same program.
This design is fundamental: because each process owns its own memory space, it must contain its own copy of the code to execute its portion of the computation.

Each process:

- Runs the same code but operates on a different subset of the data.
- Is identified by a unique number called its **rank**.
- Belongs to a **communicator**, a group of processes that can exchange messages (most commonly `MPI.COMM_WORLD`).

This model is known as **SPMD: Single Program, Multiple Data**:
a single source program runs simultaneously on many processes, each working on different data.

Why Copies of the Program Are Needed?

Because processes in distributed memory do not share variables or memory addresses, each process must have:

- Its **own copy of the executable code**, and
- Its **own private workspace (variables, arrays, etc.)**.

This independence is crucial for scalability:

- Each process can execute independently without memory contention.
- an scale to thousands of nodes, since no shared memory bottleneck exists.
- nt becomes explicit and controllable, ensuring predictable performance on large clusters.

Skip to content

Sharing Data Between Processes

Although memory is not shared, processes can **cooperate** by exchanging information through **message passing**.
MPI defines two main communication mechanisms:

1. **Point-to-point communication**: Data moves **directly** between two processes.
2. **Collective communication**: Data is exchanged among **all processes** in a communicator in a coordinated way.

Keypoints

- **Process:** Each MPI program runs as multiple independent processes, not threads.
- **Rank:** Every process has a unique identifier (its *rank*) within a communicator, used to distinguish and coordinate them.
- **Communication:** Processes exchange data explicitly through message passing, either **point-to-point** (between pairs) or **collective** (among groups).

Together, these three ideas form the foundation of MPI’s model for parallel computing.

---

mpi4py

`mpi4py` is the standard Python interface to the **Message Passing Interface (MPI)**, the same API used by C, C++, and Fortran codes for distributed-memory parallelism.
It allows Python programs to run on many processes, each with its own memory space, communicating through explicit messages.

Communicators and Initialization

In MPI, all communication occurs through a **communicator**, an object that defines which processes can talk to each other.
When a program starts, each process automatically becomes part of a predefined communicator called **`MPI.COMM_WORLD`**.

This object represents *all processes* that were launched together by `mpirun` or `srun`.

A typical initialization pattern looks like this:

Skip to content

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD      # Initialize communicator
size = comm.Get_size()     # Total number of processes
rank = comm.Get_rank()     # Rank (ID) of this process

print(f"I am rank {rank} of {size}")
```

Every process executes the same code, but rank and size allow them to behave differently.

Hello world MPI

Copy and paste this code and execute it using `mpirun -n N mpi_hello.py` where `N` is the number of tasks.
Note: Do not put more tasks than the number of cores that your computer has.

```python
# mpi_hello.py
from mpi4py import MPI

# Initialize communicator
comm = MPI.COMM_WORLD

# Get the number of processes
size = comm.Get_size()

# Get the rank (ID) of this process
rank = comm.Get_rank()

# Print a message from each process
print(f"Hello world")
```

This code snippet illustrates how independent processes run copies of the program.
To practice further try the following:

1. Use the `rank` variable to print the square of `rank` in each rank.
2. Make the program print only in rank 0, hint: `if rank == 0:`

Skip to content

Solution

Solution 1: Print the square of each rank

```
# mpi_hello_square.py
from mpi4py import MPI

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

# Each process prints its rank and the square of its rank
print(f"Process {rank} of {size} has value {rank**2}")
```

Solution 2: Print only one process (rank 0)

```
# mpi_hello_rank0.py
from mpi4py import MPI

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

if rank == 0:
    print(f"Hello world from the root process (rank {rank}) out of {size} total processes")
```

Method Naming Convention

In `mpi4py`, most MPI operations exist in **two versions**, a *lowercase* and an *uppercase* form, that differ in how they handle data.

Convention	Example	Description
Lowercase methods	`send()`, `recv()`, `bcast()`, `gather()`	High-level, Pythonic methods that can send and receive arbitrary Python objects. Data is automatically serialized (pickled). Simpler to use but slower for large data.
Uppercase methods	`Send()`, `Recv()`, `Bcast()`, `Gather()`	Low-level, performance-oriented methods that operate on **buffer-like objects** such as NumPy arrays. Data is transferred directly from memory without serialization, achieving near-C speed.

Rule of thumb:
Use *lowercase* methods for small control messages or Python objects,
ethods for numerical data stored in arrays when performance matters.

Skip to content

ıces

Lowercase (Python objects):

```
comm.send(obj, dest=1)
data = comm.recv(source=0)
```

- The message (obj) can be any Python object.
- MPI automatically serializes and deserializes it internally.
- Fewer arguments: simple but less efficient for large data.

Uppercase (buffer-like objects, e.g., NumPy arrays):

```
comm.Send([array, MPI.DOUBLE], dest=1)
comm.Recv([array, MPI.DOUBLE], source=0)
```

- Requires explicit definition of the data buffer and its MPI datatype. (same syntax as C++)
- Works directly with the memory address of the array (no serialization).
- Achieves maximum throughput for numerical and scientific workloads.

Point-to-Point Communication

The most basic form of communication in MPI is **point-to-point**, meaning data is sent from one process directly to another.

Each message involves:

- A **sender** and a **receiver**
- A **tag** identifying the message type
- A **data buffer** that holds the information being transmitted

These operations are methods of the class `MPI.COMM_WORLD`. This means that one needs to initialize it

Typical operations:

- **Send:** one process transmits data.
- **Receive:** another process waits for that data.

In `mpi4py`, each of these operations maps directly to MPI's underlying mechanisms but with a simple Python interface.

Skip to content allows one process to hand off a message to another in a fully parallel

Examples of conceptual use cases:

- Distributing different chunks of data to multiple workers.
- Passing boundary conditions between neighboring domains in a simulation.

Point-to-Point Communication

Copy and paste the code below into a file called `mpi_send_recv.py`.

```python
# mpi_send_recv.py
from mpi4py import MPI

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

if rank == 0:
    # Process 0 sends a message
    data = "Hello from process 0"
    comm.send(data, dest=1)  # send to process 1
    print(f"Process {rank} sent data: {data}")

elif rank == 1:
    # Process 1 receives a message
    data = comm.recv(source=0)  # receive from process 0
    print(f"Process {rank} received data: {data}")

else:
    # Other ranks do nothing
    print(f"Process {rank} is idle")
```

Run the program using: `mpirun -n 3 python mpi_send_recv.py` You should see output indicating that process 0 sent data and process 1 received it, while all others remained idle. Now try:

1. Change the roles: Make process 1 send a reply back to process 0 after receiving the message. Use `comm.send()` and `comm.recv()` in both directions.
2. Blocking communication: Notice that `comm.send()` and `comm.recv()` are blocking operations.

- Add a short delay using `time.sleep(rank)` before sending or receiving.
- Observe how process 0 must wait until process 1 calls `recv()` before it can continue, and vice versa.
- Try swapping the order of the calls (e.g., both processes call `send()` first), what happens?
- You may notice the program hangs or deadlocks, because both processes are waiting for a `recv()` that never starts.

Skip to content

Solution

Solution 1: Change the roles (reply back):

```
# mpi_send_recv_reply.py
from mpi4py import MPI

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

if rank == 0:
    data_out = "Hello from process 0"
    comm.send(data_out, dest=1)
    print(f"Process {rank} sent: {data_out}")

    data_in = comm.recv(source=1)
    print(f"Process {rank} received: {data_in}")

elif rank == 1:
    data_in = comm.recv(source=0)
    print(f"Process {rank} received: {data_in}")

    data_out = "Reply from process 1"
    comm.send(data_out, dest=0)
    print(f"Process {rank} sent: {data_out}")

else:
    print(f"Process {rank} is idle")
```

Solution 2: Blocking communication behavior:

1. Add a delay (e.g., time.sleep(rank)) before send/recv and observe that each blocking call waits for its partner. Example:

```
# mpi_blocking_delay.py
from mpi4py import MPI
import time

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

time.sleep(rank)  # stagger arrival

if rank == 0:
    comm.send("msg", dest=1)
    print("0 -> sent")
    print("0 -> got:", comm.recv(source=1))

elif rank == 1:
    print("1 -> got:", comm.recv(source=0))
    comm.send("ack", dest=0)
    print("1 -> sent")
```

Skip to content

istration:

```python
# mpi_deadlock.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank in (0, 1):
    # Both ranks call send() first -> potential deadlock
    comm.send(f"from {rank}", dest=1-rank)
    # This recv may never be reached if partner is also stuck in send()
    print("received:", comm.recv(source=1-rank))
```

Collective Communication

While point-to-point operations handle pairs of processes, **collective operations** involve all processes in a communicator.
They provide coordinated data exchange and synchronization patterns that are efficient and scalable.

Common collectives include:

- **Broadcast:** One process sends data to all others.
- **Scatter:** One process distributes distinct pieces of data to each process.
- **Gather:** Each process sends data back to a root process.
- **Reduce:** All processes combine results using an operation (e.g., sum, max).

Collectives are conceptually similar to group conversations, where every participant either contributes, receives, or both.
They are essential for algorithms that require sharing intermediate results or aggregating outputs.

Collectives

Let us run this code to see the collectives `bcast` and `gather` in action:

```python
# mpi_collectives.py
from mpi4py import MPI

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

# --- Broadcast example ---
if rank == 0:
    data = "Hello from the root process"
else:
    data = None

# Broadcast data from process 0 to all others
data = comm.bcast(data, root=0)
print(f"Process {rank} received: {data}")

# --- Gather example ---
# Each process creates its own message
local_msg = f"Message from process {rank}"

# Gather all messages at the root process (rank 0)
all_msgs = comm.gather(local_msg, root=0)

if rank == 0:
    print("\nGathered messages at root:")
    for msg in all_msgs:
        print(msg)
```

Now try the following:

1. Change the root process: In the broadcast section, change the root process from `rank 0` to `rank 1`.
2. How would be the same done with point to point communication?

Solution

Solution 1: Change the root process: The root process is the one that handles the behaviour of the collectives. So we just need to change the root of the collective **broadcast**.

```
# --- Broadcast example ---
if rank == 1:
    data = "Hello from process 1 (new root)"
else:
    data = None

# Broadcast data from process 1 to all others
data = comm.bcast(data, root=1)
print(f"Process {rank} received: {data}")
```

Solution 2: Manual broadcasting the previous code: To reproduce a broadcast manually using only send() and recv(), one could write:

```
# Manual broadcast using point-to-point
if rank == 1:
    data = "Hello from process 1 (manual broadcast)"
    # Send to all other processes
    for dest in range(size):
        if dest != rank:
            comm.send(data, dest=dest)
else:
    data = comm.recv(source=1)

print(f"Process {rank} received: {data}")
```

Integration with NumPy: Buffer-Like Objects

A major strength of `mpi4py` is its **direct integration with NumPy arrays**.
MPI operations can send and receive **buffer-like objects**, such as NumPy arrays, without copying data between Python and C memory.

Important

Remember that **buffer-like objects** can be used with the **uppercase methods** for avoiding serialization and its time overhead.

Because NumPy arrays expose their internal memory buffer, MPI can access this data directly. This eliminates the need for serialization (no `pickle` step) and allows **near-native C performance** for communication and collective operations.

Skip to content

Conceptually:

- Each NumPy array acts as a **contiguous memory buffer**.
- MPI transfers data directly from this buffer to another process's memory.
- This mechanism is ideal for large numerical datasets, enabling efficient data movement in parallel programs.

This integration makes it possible to:

- Distribute large datasets across processes using **collectives** like `Scatter` and `Gather`.
- Combine results efficiently with operations like `Reduce` or `Allreduce`.
- Seamlessly integrate parallelism into scientific Python workflows.

Collective Operations on NumPy Arrays

In this example, you will see how collective MPI operations distribute and combine large arrays across multiple processes using **buffer-based communication**.

Save the following code as `mpi_numpy_collectives.py` and run it with multiple processes:

```
# mpi_numpy_collectives.py
from mpi4py import MPI
import numpy as np

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

# Total number of elements in the big array (must be divisible by size)
N = 10_000_000

# Only rank 0 creates the full array
if rank == 0:
    big_array = np.ones(N, dtype="float64")  # for simplicity, all ones
else:
    big_array = None

# Each process will receive a chunk of this size
local_N = N // size

# Allocate local buffer on each process
local_array = np.empty(local_N, dtype="float64")

# Scatter the big array from root to all processes
comm.Scatter(
    [big_array, MPI.DOUBLE],       # send buffer (only valid on root)
    [local_array, MPI.DOUBLE],     # receive buffer on all processes
    root=0,
)

# Each process computes a local sum
local_sum = np.sum(local_array)

# Reduce all local sums to a global sum on the root process
global_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)

if rank == 0:
    print(f"Global sum = {global_sum}")
    print(f"Expected   = {float(N)}")
```

Run the program using

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

Questions:

Skip to content distribute and collect data in this program?

2. Why is it necessary to preallocate local_array on every process?
3. What would happen if you used lowercase methods (scatter, reduce) instead of Scatter, Reduce?

Solution

Solution 1: The MPI calls that distribute and collect data in this program are comm.Scatter() and comm.reduce(). Scatter divides the large NumPy array on the root process and sends chunks to all ranks, while Reduce collects the locally computed results and combines them (using MPI.SUM) into a single global result on the root process.

Solution 2: It is necessary to preallocate local_array on every process because the uppercase MPI methods (Scatter, Gather, Reduce, etc.) work directly with memory buffers. Each process must provide a fixed, correctly sized buffer so that MPI can write received data directly into it without additional memory allocation or copying.

Solution 3: If lowercase methods (scatter, reduce) were used instead, MPI would serialize and deserialize the Python objects being communicated (using pickle). This would make the program simpler but significantly slower for large numerical arrays, since it adds extra copying and memory overhead. Using the uppercase buffer-based methods avoids this cost and achieves near-native C performance.

Summary

mpi4py provides a simple yet powerful bridge between Python and the Message Passing Interface used in traditional HPC applications.
Conceptually, it introduces the same communication paradigms used in compiled MPI programs but with Python's expressiveness and interoperability.

Concept	Description
Process	Independent copy of the program with its own memory space
Rank	Identifier for each process within a communicator
Point-to-Point	Direct communication between pairs of processes
Collective	Group communication involving all processes
NumPy Buffers	Efficient memory sharing for large numerical data

mpi4py allows Python users to write distributed parallel programs that scale from laptops to supercomputers, making it an invaluable tool for modern scientific computing.

Keypoints

- MPI creates multiple independent processes running the same program.
- Point-to-point communication exchanges data directly between two processes.
- Collective communication coordinates data exchange across many processes.
- mpi4py integrates tightly with NumPy for efficient, zero-copy data transfers.
- These concepts allow Python programs to scale effectively on HPC systems.

Episode template

Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?
- `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

Objectives

- Show a complete lesson page with all of the most common structures.
- …

This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

Section

A section.

Discussion

Discuss the following.

- A discussion section
- Another discussion topic

Section

Skip to content

```
orld")
e default highlighting language
```

```
print("hello world)
```

Exercises: description

Exercise Topic-1: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- …

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

Quick Reference

Instructor’s guide

Why we teach this lesson

Intended learning outcomes

Timing

Preparing exercises

Skip to content ɘ day before to set up common repositories.

Other practical aspects

Interesting questions you might get

Typical pitfalls

Learning outcomes

FIXME

This material is for …

By the end of this module, learners should:

- …
- …

See also

Credit

FIXME

Don’t forget to check out additional course materials from …

Skip to content

License

CC BY-SA for media and pedagogical material

Skip to content