# High Performance Linear Algebra

Lecture 9: Basic functions of MPI — Part 2

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

January 19, 2026 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

# Last time on High Performance Linear Algebra



- Durign the last lecture we have introduced the basics of parallel programming using the **Message Passing Interface (MPI)** paradigm.
- We have discussed the main features of MPI and we have seen how to setup a simple MPI environment.
- We have also discussed the usage of SLURM to handle queues on HPC systems.

Today we will continue our discussion on MPI by introducing some of the most used MPI functions.

# Table of Contents

2 Message Passing Interface (MPI)

# MPI Timers



- `MPI_Wtime()` returns a double-precision wall-clock time in seconds
- `MPI_Wtick()` returns the resolution of `MPI_Wtime()`
- Suitable for measuring elapsed time of code regions (not CPU time)
- Call `MPI_Init` before and `MPI_Finalize` after using timers

## CPU time vs Wall-clock time

CPU time measures the time a CPU spends executing a program, while wall-clock time measures the real-world elapsed time from start to finish, including waiting times and delays. To **measure performance** in parallel computing, wall-clock time is often more relevant as it reflects the actual time users experience.

# Example: Timing a Loop



```
program timing_example
  use mpi
  implicit none
  integer :: ierr, rank, nprocs, i
  real(kind=8) :: t0, t1, elapsed

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)

  call MPI_Barrier(MPI_COMM_WORLD, ierr)  ! synchronize start
  t0 = MPI_Wtime()
```

# Example: Timing a Loop



```
  do i = 1, 10**7
     call random_seed()   ! dummy work
  end do

  call MPI_Barrier(MPI_COMM_WORLD, ierr)   ! synchronize end
  t1 = MPI_Wtime()

  elapsed = t1 - t0
  print '(A,I3,A,F8.4)', 'Rank ', rank, ' elapsed: ', elapsed

  call MPI_Finalize(ierr)
end program timing_example
```

# Barriers and Timing Best Practices



- **Barriers** (`MPI_Barrier`) force all ranks to synchronize
- Use barriers to align start/end of timed regions; avoid overuse
- Prefer `MPI_Reduce` (e.g., `MPI_MAX`) to collect max elapsed time across ranks
- Run multiple iterations and average to smooth variability

# Example: Timing with Reduction



```
program timing_reduce
  use mpi
  implicit none
  integer :: ierr, rank, root
  real(kind=8) :: t0, t1, tlocal, tmax

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
  root = 0
  call MPI_Barrier(MPI_COMM_WORLD, ierr)
  t0 = MPI_Wtime()
  call MPI_Bcast(tlocal, 1, MPI_DOUBLE_PRECISION, root, MPI_COMM_WORLD,
  ↪  ierr)
```

# Example: Timing with Reduction



```
  call MPI_Barrier(MPI_COMM_WORLD, ierr)
  t1 = MPI_Wtime()

  tlocal = t1 - t0
  call MPI_Reduce(tlocal, tmax, 1, MPI_DOUBLE_PRECISION, MPI_MAX, root,
  ↪  MPI_COMM_WORLD, ierr)

  if (rank == root) then
     print *, 'Max elapsed across ranks: ', tmax
  end if

  call MPI_Finalize(ierr)
end program timing_reduce
```

# The Cost of Communication



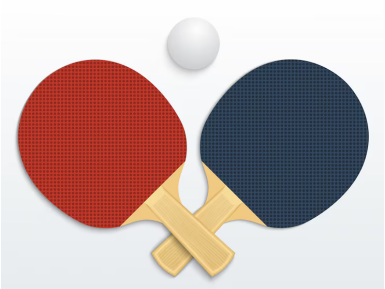
- Communication overhead increases with the number of processes
- Two main components: **latency** and **bandwidth**
  - **Latency**: time to initiate a message (startup cost)
  - **Bandwidth**: data transfer rate once communication starts
- Total communication time: $T_{\text{comm}} = \alpha + \frac{N}{\beta}$
  - $\alpha$: latency (message startup cost)
  - $\beta$: reciprocal of the bandwidth
  - $N$: message size in bytes

# Scaling Communication Costs: Point-to-Point Example



The fist measure we can do is to evaluate the scaling of point-to-point communication costs with respect to the number of MPI ranks.

An idea is to implement a simple benchmark that implements a **ping-pong test** between two MPI ranks, where rank 0 sends a message to rank $n-1$, which immediately sends it back.



The time taken for this round-trip communication is measured and reported.

# Scaling Communication Costs: Point-to-Point Example



```
program test_mpi_pingpong
   use mpi
   implicit none
   integer :: ierr, rank, nprocs
   integer :: n, i, niter
   real(kind=8), allocatable :: sendbuf(:), recvbuf(:)
   real(kind=8) :: t_start, t_end, t_local, t_avg, t_max
   call MPI_Init(ierr)
   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
   call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
   ! ------------------------------
   ! Fixed problem size (strong scaling)
   ! ------------------------------
```

# Scaling Communication Costs: Point-to-Point Example



```
n = 1000000       ! number of double-precision elements
niter = 100       ! number of repetitions
allocate(sendbuf(n), recvbuf(n))
sendbuf = 1.0d0
recvbuf = 0.0d0
```

We perform a warm-up phase to avoid measuring initial overheads,

```
! Warm-up (not timed)
if (rank == 0) then
   call MPI_Send(sendbuf, n, MPI_DOUBLE_PRECISION, nprocs-1, 0,
   ↪  MPI_COMM_WORLD, ierr)
   call MPI_Recv(recvbuf, n, MPI_DOUBLE_PRECISION, nprocs-1, 0,
   ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
else if (rank == nprocs-1) then
```



```
    call MPI_Recv(recvbuf, n, MPI_DOUBLE_PRECISION, 0, 0, MPI_COMM_WORLD,
    ↪  MPI_STATUS_IGNORE, ierr)
    call MPI_Send(sendbuf, n, MPI_DOUBLE_PRECISION, 0, 0, MPI_COMM_WORLD,
    ↪  ierr)
  end if
  call MPI_Barrier(MPI_COMM_WORLD, ierr)
```

then we perform the ping-pong test for a number of iterations, measuring the time taken for each round-trip communication.

# Scaling Communication Costs: Point-to-Point Example



```
t_local = 0.0d0
do i = 1, niter
   call MPI_Barrier(MPI_COMM_WORLD, ierr)
   t_start = MPI_Wtime()
   if (rank == 0) then
      call MPI_Send(sendbuf, n, MPI_DOUBLE_PRECISION, nprocs-1, 0,
      ↪  MPI_COMM_WORLD, ierr)
      call MPI_Recv(recvbuf, n, MPI_DOUBLE_PRECISION, nprocs-1, 0,
      ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
   else if (rank == nprocs-1) then
      call MPI_Recv(recvbuf, n, MPI_DOUBLE_PRECISION, 0, 0,
      ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
      call MPI_Send(sendbuf, n, MPI_DOUBLE_PRECISION, 0, 0,
      ↪  MPI_COMM_WORLD, ierr)
```

# Scaling Communication Costs: Point-to-Point Example



```
      end if
      t_end = MPI_Wtime()
      t_local = t_local + (t_end - t_start)
   end do
   t_avg = t_local / niter
```

Finally, we compute the average time and bandwidth, reporting the results from rank 0.

```
   ! Take the maximum time across all ranks
   call MPI_Reduce(t_avg, t_max, 1, MPI_DOUBLE_PRECISION, MPI_MAX, 0,
   ↪  MPI_COMM_WORLD, ierr)
   if (rank == 0) then
      print *, "MPI Ping-Pong strong scaling test"
      print *, "Message size (MB): ", n * 8.0d0 / 1.0d6
      print *, "MPI ranks        : ", nprocs
```

# Scaling Communication Costs: Point-to-Point Example



```
      print *, "Avg Ping-Pong time (s): ", t_max
      print *, "Avg Bandwidth (GB/s): ", (n * 8.0d0 * 2.0d0) / (t_max *
      ↪  1.0d9)
      print *, "Avg Bandwidth (GBit/s): ", (n * 8.0d0 * 8.0d0 * 2.0d0) /
      ↪  (t_max * 1.0d9)
   end if
   deallocate(sendbuf, recvbuf)
   call MPI_Finalize(ierr)
end program test_mpi_pingpong
```

# Scaling Communication Costs: Point-to-Point Example



We recall that the bandwidth is computed as the total data transferred (send + receive) divided by the time taken, i.e.,

$$\text{Bandwidth} \approx \frac{N \times 2}{T_{\text{comm}}}$$

where $N$ is the message size in bytes and $T_{\text{comm}}$ is the average time taken for the ping-pong communication.
We have an array of size $n$ double-precision elements, so the message size in bytes is $N = n \times 8$ bytes, and the total data transferred in the ping-pong test is $N \times 2$ bytes and we multiply by 8 to convert to bits.

# Scaling Communication Costs: Point-to-Point Example



To run this benchmark on a HPC system with SLURM, we use the Amelia cluster at IAC-CNR.

This is a machine whose nodes are equipped with Intel Xeon Gold 6338 processors (32 cores per socket, 2 sockets per node), connected via an InfiniBand HDR200 network with a theoretical **peak bandwidth** of $200\,\mathrm{Gbit\,s^{-1}}$.

We write a SLURM script to run the benchmark with different numbers of MPI ranks,

```
#!/bin/bash
#SBATCH --job-name=pingpong_strong_64ppn
#SBATCH --nodes=7
#SBATCH --ntasks-per-node=64
#SBATCH --time=00:20:00
#SBATCH --partition=prod-gn
#SBATCH --mem=900Gb
#SBATCH --output=pingpong_%j.out
#SBATCH --error=pingpong_%j.err
```

# Scaling Communication Costs: Point-to-Point Example



We load the necessary modules and compile the Fortran code using `mpifort`,

```
module load intel/gcc-12.2.1/openmpi-4.1.6
```

And then use bash loops to run the benchmark with different numbers of MPI ranks,

```
for NODES in $(seq 1 7); do
    NTASKS=$((NODES * 64))
    echo "Running Ping-Pong with:"
    echo "  Nodes : $NODES"
    echo "  Tasks : $NTASKS (64 per node)"
    mpirun --bind-to core --map-by ppr:64:node --mca btl ^openib --mca pml
    ↪  ucx -x UCX_NET_DEVICES='mlx5_0:1' -np $NTASKS ./pingpong
done
```

The script can be submitted to the SLURM queue using the command:
`sbatch runner-pingpong.sh`.

# Scaling Communication Costs: Point-to-Point Example



We have passed several options to `mpirun` to optimize the communication performance:

- `--bind-to core`: binds each MPI process to a specific CPU core to reduce context switching and improve cache utilization.
- `--map-by ppr:64:node`: maps 64 MPI processes per node, ensuring an even distribution of processes across the available nodes.
- `--mca btl ^openib`: disables the OpenIB BTL (Byte Transfer Layer) to avoid potential issues with certain InfiniBand configurations.
- `--mca pml ucx`: selects the UCX (Unified Communication X) PML (Point-to-Point Messaging Layer) for improved performance on high-speed networks.
- `-x UCX_NET_DEVICES='mlx5_0:1'`: specifies the network devices to be used by UCX for communication, optimizing data transfer over the InfiniBand network.

❓ The last three options are relevant only to the specific network configuration of the Amelia cluster, thus they might not be necessary on other systems.

# Scaling Communication Costs: Point-to-Point Example



We have passed several options to `mpirun` to optimize the communication performance:

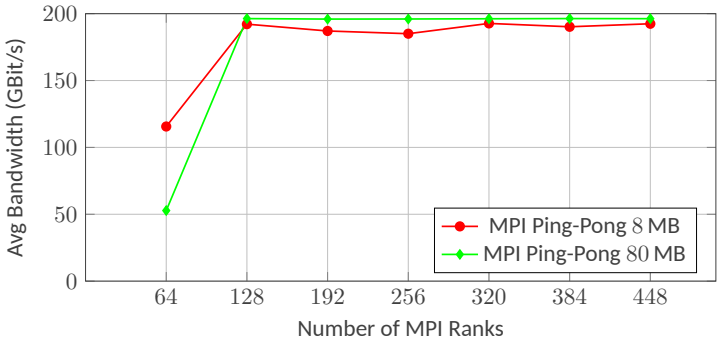
- `--bind-to core`: binds each MPI process to a specific CPU core to reduce context switching and improve cache utilization.
- `--map-by ppr:64:node`: maps 64 MPI processes per node, ensuring an even distribution of processes across the available nodes.
- `--mca btl ^openib`: disables the OpenIB BTL (Byte Transfer Layer) to avoid potential issues with certain InfiniBand configurations.
- `--mca pml ucx`: selects the UCX (Unified Communication X) PML (Point-to-Point Messaging Layer) for improved performance on high-speed networks.
- `-x UCX_NET_DEVICES='mlx5_0:1'`: specifies the network devices to be used by UCX for communication, optimizing data transfer over the InfiniBand network.

⚠ The first two options are important for what we are doing, because they ensure that we really use the network as expected.

# Scaling Communication Costs: Point-to-Point Example



We increase the number of MPI ranks from 64 to 448 (7 nodes with 64 ranks each).




The average ping-pong bandwidth approaches the theoretical peak bandwidth of the InfiniBand HDR200 network ($200\,\text{Gbit}\,\text{s}^{-1}$).

# Exercise



There are a number of possible exercises you can do to further explore the ping-pong benchmark

1. Vary the message size and perform linear regression to estimate the latency and bandwidth parameters ($\alpha$ and $\beta$) of the communication model.
2. Implement a similar benchmark using non-blocking communication (e.g., `MPI_Isend` and `MPI_Irecv`).
3. Benchmark what happens when you use MPI ranks on different nodes versus on the same node.
4. Explore the impact of different MPI implementations (e.g., OpenMPI vs MPICH vs Intel MPI) on communication performance.

# The problem of collective communications



Let's review the idea of the Broadcast operation. Algorithmically, we can implement it as:

```
if (my_rank == 0) then
    do i = 1, p-1
        call MPI_Send(a, 1, MPI_REAL, i, tag, MPI_COMM_WORLD, ierr)
    end do
else
    call MPI_Recv(a, 1, MPI_REAL, 0, tag, MPI_COMM_WORLD,
    ↪  MPI_STATUS_IGNORE, ierr)
end if
```

- This is a **collective communication** since *all* processes participate in its implementation;
- Just from a software engineering perspective, it makes sense to encapsulate it in a function: the `MPI_Bcast`.

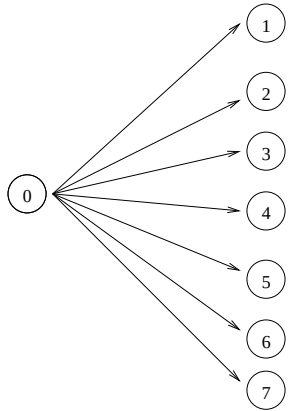
# How much does it cost?



- With fast networks, cost for 1 float is dominated by latency $\alpha$;
- Cost of this algorithm is therefore

$$T(p) \approx \alpha \cdot (p-1)$$

or, linear in $p$.



# Can we do any better?



Let's make some assumptions about the network:

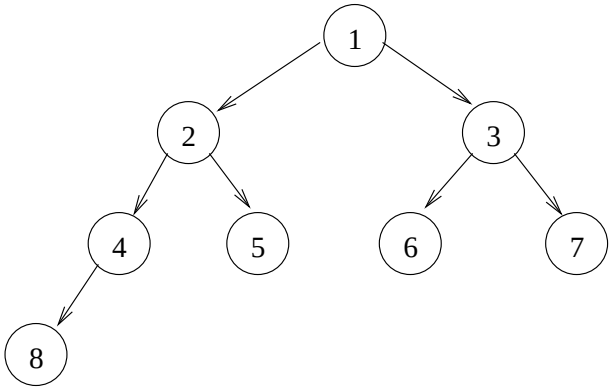
- The cost of communication between any two network nodes is uniform, and is given by $\alpha + N/\beta$;
- In particular, it is *possible* to send a message between *any* two nodes[1]
- The network is capable of sustaining multiple messages (*noisy topology*) at the same time, provided pairs of nodes involved in the messages do not overlap.

The latter assumption is especially important: we can improve communication if we can have multiple messages "flying" through the network at the same time.

[1] Historically there existed networks where only neighbouring nodes could exchange messages

# Tree broadcast — simple minded





# Tree broadcast — simple minded



Assume processes are numbered from 1:

- Each process $i$ (except 1) receives from $\lfloor (p-1)/2 \rfloor$;
- Each process such that $2i <= p$ sends first to process $2i$, then to process $2i+1$.

Cost:

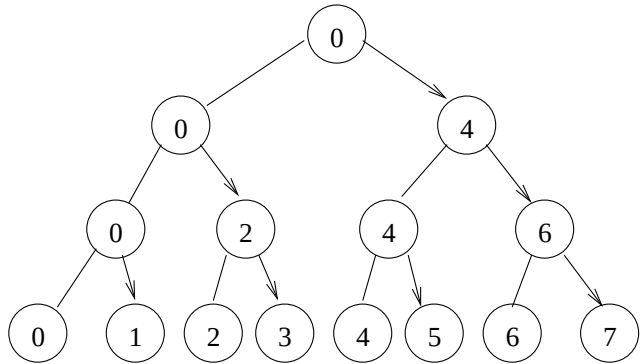
$$T(p) \approx 2\log(p),$$

or logarithmic in $p$. To be precise, with $p > 1$ then

$$T(p) = \begin{cases} 0 & \text{for } p = 1 \\ 2 \cdot (k-1) + 1 & \text{for } p = 2^k, k > 0 \\ 2 \cdot \lfloor \log_2(p) \rfloor & \text{otherwise} \end{cases}$$

# Tree broadcast — Recursive Doubling





# Tree broadcast — Recursive Doubling



- Consider that there are $p$ processes with root 0;
- Set $K$ the minimum power of 2 such that $K \geq p$;
- Process 0 sends to process $K/2$;
- Divide the processes in two groups: from 0 to $K/2 - 1$, and from $K/2$ to $\min(p-1, K-1)$;
- Apply recursively to:
  - Processes 0 to $K/2 - 1$ with root 0;
  - Processes $K/2$ to $\min(p-1, K-1)$ with root $K/2$.

Cost:

$$T(p) = \lceil \log(p) \rceil,$$

or logarithmic in $p$.

# Collective communications



Considerations for collective communications:

- Their functionality can be defined in terms of simple loops;
- There exist much better implementations;
- The **optimal implementation** for a given collective depends on:
  - The operation;
  - The network interface;
  - The network topology;
  - The amount of data.

A good MPI implementation will switch internally among different algorithms where appropriate (another advantage of encapsulating the collective)

# Collective communications



Considerations for collective communications:

- Their functionality can be defined in terms of simple loops;
- There exist much better implementations;
- The **optimal implementation** for a given collective depends on:
  - The operation;
  - The network interface;
  - The network topology;
  - The amount of data.

A good MPI implementation will switch internally among different algorithms where appropriate (another advantage of encapsulating the collective)

🔨 Let us try and take some measurements of the broadcast operation to see how its cost scales with the number of MPI ranks.

# Scaling Communication Costs: Broadcast Example



```
program test_mpi_bcast_latency
    use mpi
    implicit none

    integer :: ierr, rank, nprocs
    integer :: root
    integer :: n, i, niter
    real(kind=8), allocatable :: buffer(:)
    real(kind=8) :: t_start, t_end, t_local, t_avg, t_max

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
```

# Scaling Communication Costs: Broadcast Example



```
root = 0
! Small message size for latency measurement
n = 1               ! single element for pure latency
niter = 10000       ! more iterations for better statistics
allocate(buffer(n))
if (rank == root) then
     buffer = 1.0d0
else
     buffer = 0.0d0
end if
! Warm-up (not timed)
call MPI_Bcast(buffer, n, MPI_DOUBLE_PRECISION, root, MPI_COMM_WORLD,
↪  ierr)
```

# Scaling Communication Costs: Broadcast Example



```
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t_local = 0.0d0
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t_start = MPI_Wtime()
do i = 1, niter
     call MPI_Bcast(buffer, n, MPI_DOUBLE_PRECISION, root,
     ↪  MPI_COMM_WORLD, ierr)
end do
t_end = MPI_Wtime()
     t_local = t_local + (t_end - t_start)
t_avg = (t_end - t_start) / niter
! Take the maximum latency across all ranks
call MPI_Reduce(t_avg, t_max, 1, MPI_DOUBLE_PRECISION, MPI_MAX, root,
↪  MPI_COMM_WORLD, ierr)
```

# Scaling Communication Costs: Broadcast Example

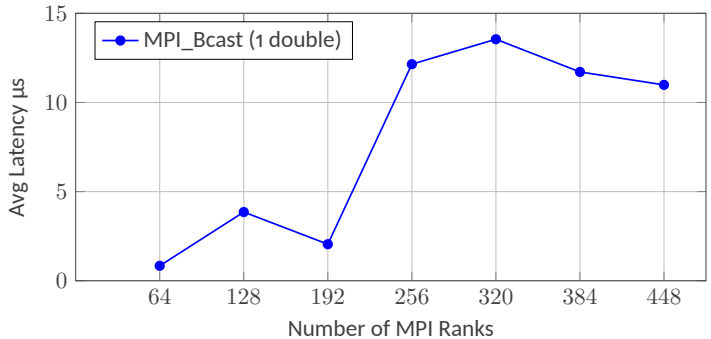


```
    if (rank == root) then
         print *, "MPI_Bcast latency test"
         print *, "MPI ranks            : ", nprocs
         print *, "Avg Bcast latency (us): ", t_max * 1.0d6
    end if
    deallocate(buffer)
    call MPI_Finalize(ierr)
end program test_mpi_bcast_latency
```

# Scaling Communication Costs: Broadcast Example



We run the benchmark on the Amelia cluster at IAC-CNR, (20 nodes with 64 ranks each).




The observed `MPI_Bcast` latency shows a non-monotonic scaling trend as the number of MPI ranks increases.

# Interpreting Broadcast Scaling Behavior



This behavior is expected and reflects the internal algorithm selection.

- **1 node (64 ranks):** Extremely low latency due to shared-memory communication.
- **2–3 nodes (128–192 ranks):** Transition to inter-node communication with efficient tree-based collectives.
- **4–5 nodes (256–320 ranks):** Sudden increase in latency caused by algorithm switching and network contention.
- **6–7 nodes (384–448 ranks):** Stabilization as MPI switches to scalable hierarchical broadcast algorithms.

Overall, broadcast latency is dominated by collective startup costs rather than message size, and is strongly influenced by communicator size, topology awareness, and algorithm thresholds.

# Table of Contents

3 Restarting with BLAS: Level 1 routines

# Restarting with BLAS: Level 1 routines



The first operation we are going to considere are the level-1 BLAS routines, which we recall are **vector-vector operations**.

- Vector scaling: $\mathbf{y} \leftarrow \alpha\mathbf{y}$ (`DSCAL`)
- Vector addition: $\mathbf{y} \leftarrow \mathbf{y} + \alpha\mathbf{x}$ (`DAXPY`)
- Vector dot product: $\alpha \leftarrow \mathbf{x}^T\mathbf{y}$ (`DDOT`)
- Vector norm: $\alpha \leftarrow \|\mathbf{x}\|_2$ (`DNRM2`)

In our model, **each process holds a local portion of the vectors**, this means that for a vector of size $N$ distributed over $P$ processes, each process holds a local vector of size $N/P$.

# How do we distribute vectors?



The **basic idea** can be represented with an easy picture:

| Proc 0 | Proc 1 | Proc 2 | Proc 3 | Proc 4 | Proc 5 | Proc 6 | Proc 7 |
|---|---|---|---|---|---|---|---|

Full Vector of Size $N$ divided across $P = 8$ Processes

This permits us to:

- Perform local computations on each process independently;
- Know what data is stored where with a closed-form formula:

$$\text{Process } p \text{ holds elements } \left[ p \cdot \frac{N}{P}, (p+1) \cdot \frac{N}{P} - 1 \right]$$

# Implementing vector distribution



To implement this distribution in code, we need to:

- Initialize MPI and get the rank and size of the communicator;
- Determine the global vector size $N$ and compute the local size $N/P$;
- Allocate local arrays for each process to hold its portion of the vector;

The inizialization code looks always the same:

```
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
```

To **compute the local size** and allocate the local vector we can use a simple-minded block distribution.

# Implementing vector distribution: block distribution



Assuming $N$ is divisible by $P$, the local size is simply:

$$N_{\text{local}} = \frac{N}{P}$$

In the general case, we can compute the local size as:

$$N_{\text{local}} = \left\lfloor \frac{N + P - 1 - p}{P} \right\rfloor$$

where $p$ is the rank of the process. This formula ensures that the last process gets any remaining elements if $N$ is not perfectly divisible.

We can implement this as a function:

```
function compute_local_size(N, P, p) result(n_local)
  integer, intent(in) :: N, P, p
  integer :: n_local
  n_local = (N + P - 1 - p) / P
end function compute_local_size
```

# Implementing Level-1 BLAS: a type for distributed vectors



The first thing we need to do is to create a distributed vector datytpe, this can be done using the **object-oriented** functionalities of Fortran:

```
type :: mpi_ddistributed_vector
    integer :: n_local(1)  ! number of local elements
    integer :: n_global    ! total number of global elements
    integer :: comm        ! MPI communicator
    real(real64), allocatable :: data(:)  ! local data array
end type mpi_ddistributed_vector
```

The **type** encapsulates all necessary information about the distributed vector, including *local size*, *global size*, *communicator*, and *local data array*.

# Implementing Level-1 BLAS: type-bound procedures



In Fortran a **type** can have **type-bound procedures**, which are functions or subroutines associated with the type. Thesere are declared within the `contains` section of the type definition.

```
type :: mpi_ddistributed_vector
  <Type members>
contains
  <Type bound procedure are declared here>
end type mpi_ddistributed_vector
```

For our distributed vector type, we need to start by implementing:

- A **constructor** to initialize the distributed vector;
- A **destructor** to free resources;

# Implementing Level-1 BLAS: Constructor



The **constructor** initializes the distributed vector by computing the local size, allocating the local data array, and setting the global s ize and communicator.

- We add the following type-bound procedure to the type:

```
procedure, pass(this) :: dinit
```

- The implementation of the constructor is then inserted in che `contains` part

```
subroutine dinit(this, comm, n_global)
 use mpi
 use iso_fortran_env, only: error_unit
 implicit none
 class(mpi_ddistributed_vector), intent(inout) :: this
 integer, intent(in) :: comm
 integer, intent(in) :: n_global
end subroutine dinit
```

# Implementing Level-1 BLAS: Constructor



The **type constructor** has a default variable `this` that refers to the instance of the type being initialized.

```
integer :: ierr, rank, nprocs
this%comm = comm
this%n_global = n_global
call MPI_Comm_rank(this%comm, rank, ierr)
call MPI_Comm_size(this%comm, nprocs,
↪  ierr)
this%n_local =
↪  compute_local_size(this%n_global,
↪  nprocs, rank)
allocate(this%data(this%n_local),
↪  stat=ierr)
```

- We assign the communicator and global size to the type members;
- We get the rank and size of the communicator;
- We compute the local size using the previously defined function;
- We allocate the local data array based on the computed local size.

We should also include **error handling** for the allocation.

# Implementing Level-1 BLAS: Constructor (error handling)



The **error handling** for the allocation can be done by checking the status of the allocation.

```
if (ierr /= 0) then
    write(error_unit, *) "Error allocating local data array on rank ", rank
    call MPI_Abort(this%comm, ierr, ierr)
end if
```

The `MPI_Abort` function is called to **terminate the MPI program** if the allocation fails, ensuring that all processes are informed of the error, and that the *program fails and exits gracefully*.

# Implementing Level-1 BLAS: Destructor



The **destructor** is responsible for freeing the resources allocated to the distributed vector.

- We add use the `final` keyword in the type bound procedures to define the destructor:

```
final :: dfinalize
```

- The implementation of the destructor is then inserted in che `contains` part

```
subroutine dfinalize(this)
  type(mpi_ddistributed_vector), intent(inout) :: this
  if (allocated(this%data)) then
      deallocate(this%data)
  end if
end subroutine dfinalize
```

# The creation and destruction of distributed vectors



With the constructor and destructor defined, we can now create and destroy distributed vector instances easily.

```
integer :: n_global, rank, ierr
type(mpi_ddistributed_vector) :: x
call MPI_Init(ierr)
n_global = 1000000  ! Total number of elements
call x%dinit(MPI_COMM_WORLD, n_global)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
if (rank == 0) then
  write(output_unit, *) "MPI ranks           : ", nprocs
  write(output_unit, *) "Global vector size  : ", n_global
end if
call MPI_Barrier(MPI_COMM_WORLD, ierr)
write(output_unit, *) "Rank ", rank, ": Local vector size : ", x%n_local
call MPI_Finalize(ierr)
```

# Conclusions and next steps



Today we have:

- ✔ Reviewed the concept of communication costs in MPI;
- ✔ Implemented a point-to-point ping-pong benchmark to measure communication latency and bandwidth;
- ✔ Explored the scaling behavior of collective communications using a broadcast benchmark;
- ✔ Introduced the concept of distributed vectors and how to implement them in Fortran using object-oriented programming.

Next steps:

- Implement Level-1 BLAS operations (vector scaling, addition, dot product, norm) for distributed vectors;
- Explore Level-2 and Level-3 BLAS operations