High Performance Linear Algebra

Lecture 11: Distributed BLAS level 2

Ph.D. program in High Performance Scientific Computing

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

January 26, 2026 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

Last time on High Performance Linear Algebra

We have

- Described the Message Passing Interface (MPI) programming model,
- Implemented Level-1 BLAS operations for distributed vectors using MPI,
- Analyzed performance characteristics for axpy, dot product, and norm,
- Identified scaling limits due to communication overhead in reduction operations

Next steps:

- Discuss data distribution strategies for distributed matrices,
- Implement Level-2 BLAS operations for distributed matrices and vectors,
- Explore the Level-2 and Level-3 BLAS operations in distributed settings.

Table of Contents

Distributed matrices

When dealing with large matrices that cannot fit into the memory of a single node, we need to **distribute the matrix across multiple nodes** of our distributed-memory environment.

How can we do that?

- we have already partially addressed this question when discussing the construction of matrix-vector products and matrix-matrix products in a shared-memory context;
- we have discussed this for the case of distributed vectors last time.

Key concerns in data layout

When choosing a data layout for dense matrix computations on distributed-memory systems, we must consider:

Load Balance

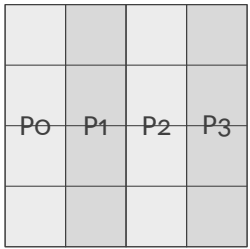
- Split work evenly among processors
- Avoid idle processors during computation

Computational Efficiency

- Utilize Level 3 BLAS on local data
- Leverage memory hierarchy

These requirements often conflict and require careful trade-offs!

1D Block Column Distribution



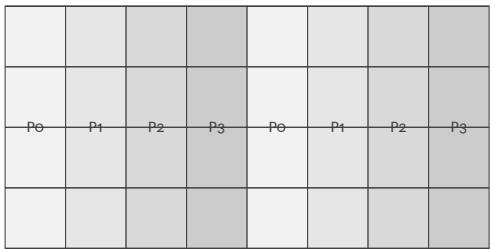
Characteristics:

- Each process stores one block of contiguous columns
- Column k goes to process $\lfloor (k-1)/n_c \rfloor \mod P$

Problems:

- **Poor load balance**: once columns are completed, processes become idle
- Not suitable for matrix operations like Gaussian elimination

1D Cyclic Column Distribution



Characteristics:

- Column k assigned to process $(k-1) \mod P$
- Single columns are interleaved across processes

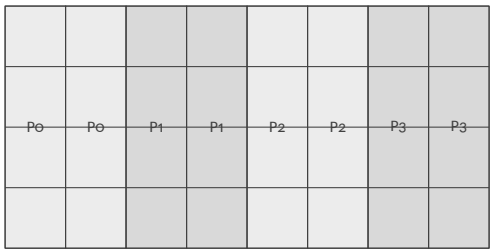
Advantages:

- Better load balance
- Good work distribution

Problems:

- Cannot use Level 2/3 BLAS efficiently (only single columns)
- Poor memory hierarchy utilization

1D Block-Cyclic Column Distribution



Characteristics:

- Choose block size NB
- Column block k assigned to process $\lfloor (k-1)/NB \rfloor \mod P$

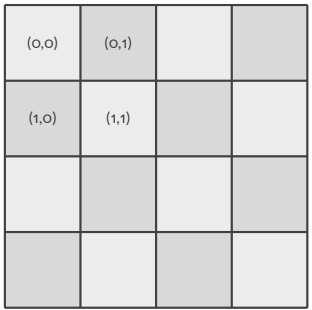
Advantages:

- Reasonable load balance
- Can use Level 2/3 BLAS locally

Problems:

- Serial bottleneck: factorization occurs on one process
- Limited parallelism in factorization step

2D Block-Cyclic Distribution



Processors in $P_r \times P_c$ grid

Characteristics:

- Arrange P processes in $P_r \times P_c$ rectangular grid
- Block size parameters: $MB \times NB$
- Block (i, j) goes to process $(i \bmod P_r, j \bmod P_c)$

Advantages:

- Excellent load balance
- Allows P_c-fold parallelism in columns
- Can use Level 2/3 BLAS
- Good scalability

This is the standard layout used by ScaLAPACK! But how do we implement it?

Comparison of Distribution Strategies

Strategy	Load Balance	BLAS Use	Scalability	Complexity
1D Block	Poor	Good	Bad	Simple
1D Cyclic	Good	Poor	Fair	Simple
1D Block-Cyclic	Fair	Fair	Fair	Moderate
2D Block-Cyclic	Excellent	Good	Excellent	Complex

Key Trade-offs:

- Load balance vs. computational efficiency
- Simple schemes vs. scalability
- The 2D block-cyclic strategy achieves the best overall balance

Implementing the 2D Block-Cyclic Distribution

We need to map global matrix indices to local storage on each process.

Global to Local Index Mapping (2D block-cyclic):

- Communicator with $P = P_r \times P_c$ processes
- Global matrix of size $M \times N$
- Process grid of size $P_r \times P_c$
- Block size $MB \times NB$
- Source process $(R_{\text{src}}, C_{\text{src}})$
- Use 0-based global indices: $i_0 = i - 1$, $j_0 = j - 1$
- Global block indices:

$$b_r = \left\lfloor \frac{i_0}{MB} \right\rfloor, \quad b_c = \left\lfloor \frac{j_0}{NB} \right\rfloor$$

Implementing the 2D Block-Cyclic Distribution

We need to map global matrix indices to local storage on each process.

Global to Local Index Mapping (2D block-cyclic):

- Communicator with $P = P_r \times P_c$ processes
- Global matrix of size $M \times N$
- Process grid of size $P_r \times P_c$
- Block size $MB \times NB$
- Source process $(R_{\text{src}}, C_{\text{src}})$
- Use 0-based global indices: $i_0 = i - 1$, $j_0 = j - 1$
- Owning process:

$$p_r = (b_r - R_{\text{src}}) \bmod P_r, \quad p_c = (b_c - C_{\text{src}}) \bmod P_c$$

Implementing the 2D Block-Cyclic Distribution

We need to map global matrix indices to local storage on each process.

Global to Local Index Mapping (2D block-cyclic):

- Communicator with $P = P_r \times P_c$ processes
- Global matrix of size $M \times N$
- Process grid of size $P_r \times P_c$
- Block size $MB \times NB$
- Source process $(R_{\text{src}}, C_{\text{src}})$
- Use 0-based global indices: $i_0 = i - 1$, $j_0 = j - 1$
- Local block indices:

$$\ell_r = \left\lfloor \frac{b_r - R_{\text{src}}}{P_r} \right\rfloor, \quad \ell_c = \left\lfloor \frac{b_c - C_{\text{src}}}{P_c} \right\rfloor$$

Implementing the 2D Block-Cyclic Distribution

We need to map global matrix indices to local storage on each process.

Global to Local Index Mapping (2D block-cyclic):

- Communicator with $P = P_r \times P_c$ processes
- Global matrix of size $M \times N$
- Process grid of size $P_r \times P_c$
- Block size $MB \times NB$
- Source process $(R_{\text{src}}, C_{\text{src}})$
- Use 0-based global indices: $i_0 = i - 1$, $j_0 = j - 1$
- Local indices (1-based, Fortran):

$$\text{local\_row} = \ell_r \cdot MB + (i_0 \bmod MB) + 1$$

$$\text{local\_col} = \ell_c \cdot NB + (j_0 \bmod NB) + 1$$

Building a descriptor for the 2D process grid

To implement the 2D block-cyclic distribution, we need a descriptor that contains:

- Global matrix dimensions `M`, `N`
- Block sizes `MB`, `NB`
- Process grid dimensions `P_r`, `P_c`
- Leading dimension of local arrays
- Starting indices for submatrices, they are usually called `RSRC` and `CSRC` parameters specify the starting process coordinates for the distribution, and are typically set to 0.

This descriptor will be used in all distributed matrix operations to correctly map global indices to local storage. We can build it as a Fortran derived type.

Fortran Descriptor Type

2 Distributed matrices

```
type :: descriptor
    integer :: comm      ! MPI communicator
    integer :: M         ! Global number of rows
    integer :: N         ! Global number of columns
    integer :: MB        ! Block size in rows
    integer :: NB        ! Block size in columns
    integer :: P_r       ! Number of process rows
    integer :: P_c       ! Number of process columns
    integer :: LLD       ! Leading dimension of local array
    integer :: RSRC      ! Row source process
    integer :: CSRC      ! Column source process
end type descriptor
```

This structure encapsulates all necessary information for managing the 2D block-cyclic distribution of matrices across a process grid.

But how do we init the descriptor?

To initialize the descriptor, we can create an initialization procedure that sets all the fields based on the global matrix size, block sizes, and process grid dimensions.

- This procedure will be called once at the beginning of the program to set up the descriptor.
- It will compute the leading dimension of the local arrays based on the block sizes and process grid.
- The process coordinates (p_r, p_c) must be provided to correctly map back to global indices, they can be retrieved from the MPI rank.

But how do we init the descriptor?


```
subroutine init(desc, comm, M, N, MB, NB, P_r, P_c, RSRC, CSRC)
    class(descriptor), intent(out) :: desc
    integer, intent(in) :: comm, M, N, MB, NB, P_r, P_c, RSRC, CSRC
    desc%comm = comm
    desc%M = M
    desc%N = N
    desc%MB = MB
    desc%NB = NB
    desc%P_r = P_r
    desc%P_c = P_c
    desc%RSRC = RSRC
    desc%CSRC = CSRC
    desc%LLD = ((M + P_r * MB - 1) / (P_r * MB)) * MB
end subroutine init
```

A distributed matrix type

We now need to build a **distributed matrix** type that uses the **descriptor** to manage its data.

This can be done by defining a Fortran derived type that contains:

- The local array to store the matrix data,
- The descriptor for the matrix distribution,
- Type-bound procedures for the BLAS matrix operations.

Distributed Matrix Type Definition

We can define the distributed matrix type as follows:

```
type :: distributed_matrix
    real, allocatable :: local_data(:,:) ! Local matrix storage
    type(descriptor) :: desc              ! Descriptor for distribution
contains
  <Type bound procedures for matrix operations go here>
end type distributed_matrix
```

We don't need to add more fields, as the descriptor contains all necessary information about the global matrix, and the size of the local is in the `desc%NB` and `desc%MB` fields.

The distributed GEMV operation

The first Level-2 BLAS operation we can implement is the distributed matrix-vector product (GEMV):

$$\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$$

where A is a distributed matrix, and $\mathbf{x}$ and $\mathbf{y}$ are distributed vectors.

Key Steps:

- Each process computes its local contribution to the matrix-vector product.
- A global reduction is performed to combine the local results into the final vector $\mathbf{y}$.

❓ But how do we distribute the vectors $\mathbf{x}$ and $\mathbf{y}$?

The distributed GEMV operation

The first Level-2 BLAS operation we can implement is the distributed matrix-vector product (GEMV):

$$\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$$

where A is a distributed matrix, and $\mathbf{x}$ and $\mathbf{y}$ are distributed vectors.

Key Steps:

- Each process computes its local contribution to the matrix-vector product.
- A global reduction is performed to combine the local results into the final vector $\mathbf{y}$.

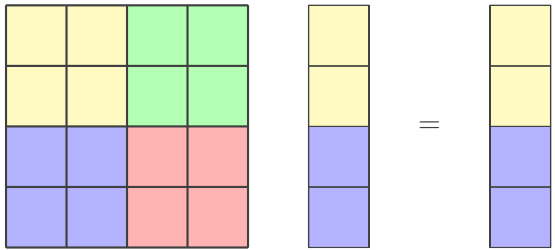
❓ But how do we distribute the vectors $\mathbf{x}$ and $\mathbf{y}$?

❗ We can use a 1D block distribution for the vectors to align with the matrix distribution.

Distributed GEMV Implementation

2 Distributed matrices

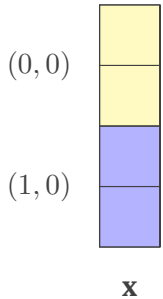
We can visualize the distributed GEMV operation with a block cyclic distributed matrix, and a 1D block distributed vector as in the following diagram:



$$\mathbf{y} = A\mathbf{x}, \qquad A \in \mathbb{R}^{N \times N}, \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^{N \times 1}$$

Vector Distribution

- Column vector: $x \in \mathbb{R}^{N \times 1}$
- Descriptor parameters:
  - `M = N`
  - `N = 1`
  - `NB = 1`
- Block-cyclic distribution in rows only
- Only **process column** 0 stores vector data



What communications are needed for GEMV?

We can represent the distributed GEMV operation as follows:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{00}x_0 + A_{01}x_1 + A_{02}x_2 + A_{03}x_3 \\ A_{10}x_0 + A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \\ A_{20}x_0 + A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \\ A_{30}x_0 + A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \end{bmatrix}$$

This means that:

- All processes with processo column 0 need x_0
- All processes with processo column 1 need x_1
- All processes with processo column 2 need x_2
- All processes with processo column 3 need x_3

What communications are needed for GEMV?

2 Distributed matrices

We can represent the distributed GEMV operation as follows:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{00}x_0 + A_{01}x_1 + A_{02}x_2 + A_{03}x_3 \\ A_{10}x_0 + A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \\ A_{20}x_0 + A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \\ A_{30}x_0 + A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \end{bmatrix}$$

This means that:

- All processes with processo column 0 need x_0
- All processes with processo column 1 need x_1
- All processes with processo column 2 need x_2
- All processes with processo column 3 need x_3

What communications are needed for GEMV?

We can represent the distributed GEMV operation as follows:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{00}x_0 + A_{01}x_1 + A_{02}x_2 + A_{03}x_3 \\ A_{10}x_0 + A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \\ A_{20}x_0 + A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \\ A_{30}x_0 + A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \end{bmatrix}$$

This means that:

- All processes with processo column 0 need x_0
- All processes with processo column 1 need x_1
- All processes with processo column 2 need x_2
- All processes with processo column 3 need x_3

What communications are needed for GEMV?

We can represent the distributed GEMV operation as follows:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{00}x_0 + A_{01}x_1 + A_{02}x_2 + A_{03}x_3 \\ A_{10}x_0 + A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \\ A_{20}x_0 + A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \\ A_{30}x_0 + A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \end{bmatrix}$$

This means that:

- All processes with processo column 0 need x_0
- All processes with processo column 1 need x_1
- All processes with processo column 2 need x_2
- All processes with processo column 3 need x_3

What communications are needed for GEMV?

We can represent the distributed GEMV operation as follows:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{00}x_0 + A_{01}x_1 + A_{02}x_2 + A_{03}x_3 \\ A_{10}x_0 + A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \\ A_{20}x_0 + A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \\ A_{30}x_0 + A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \end{bmatrix}$$

This means that:

- All processes with processo column 0 need x_0
- All processes with processo column 1 need x_1
- All processes with processo column 2 need x_2
- All processes with processo column 3 need x_3
- Each process computes its local contribution to **y** and then **reduce along the column**.

What is the right communication function?

To implement the required communication pattern, we can use:

- We need to broadcast the vector segments x_0, x_1, x_2, x_3 along the process columns.

$$x_0 \to (0,0), (1,0), (2,0), (3,0)$$

$$x_1 \to (0,1), (1,1), (2,1), (3,1)$$

$$x_2 \to (0,2), (1,2), (2,2), (3,2)$$

$$x_3 \to (0,3), (1,3), (2,3), (3,3)$$

- We need **a communicator for each process column** to perform these broadcasts.

Building communicators for GEMV

We need to create communicators for the process rows and columns:

```
dims(1) = P_r ! Define process grid dimensions
dims(2) = P_c
periods(1) = .false.
periods(2) = .false.
! Create Cartesian communicator
call MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, .true., cart_comm,
↪  ierr)
! Split into row and column communicators
call MPI_Comm_rank(cart_comm, cart_rank, ierr)
call MPI_Cart_coords(cart_comm, cart_rank, 2, coords, ierr)
call MPI_Comm_split(cart_comm, coords(1), coords(2), row_comm, ierr)
call MPI_Comm_split(cart_comm, coords(2), coords(1), col_comm, ierr)
```

These communicators allow us to perform broadcasts and reductions efficiently along the rows and columns of the process

Details and signature of the new MPI functions

The subroutine `MPI_Cart_create` has signature:

```
call MPI_Cart_create(comm_old, ndims, dims, periods, reorder,
↪  comm_cart, ierr)
```

where:

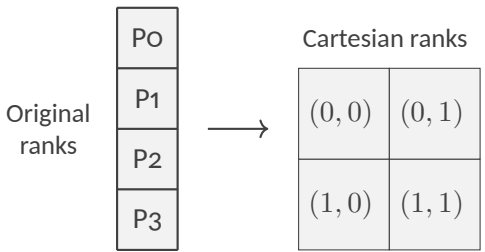
- `comm_old`: input communicator (e.g., `MPI_COMM_WORLD`)
- `ndims`: number of dimensions (2 for a 2D grid)
- `dims`: array specifying the size of each dimension
- `periods`: array specifying whether each dimension is periodic
- `reorder`: logical flag to allow process rank reordering
- `comm_cart`: output Cartesian communicator
- `ierr`: error code

Details and signature of the new MPI functions

The subroutine `MPI_Cart_create` has signature:

```
call MPI_Cart_create(comm_old, ndims, dims, periods, reorder,
↪  comm_cart, ierr)
```

This subroutine creates a **Cartesian communicator** based on the specified dimensions and periodicity, i.e., a 2D grid of processes.



Details and signature of the new MPI functions

The subroutine `MPI_Comm_split` has signature:

```
call MPI_Comm_split(comm, color, key, newcomm, ierr)
```

where:

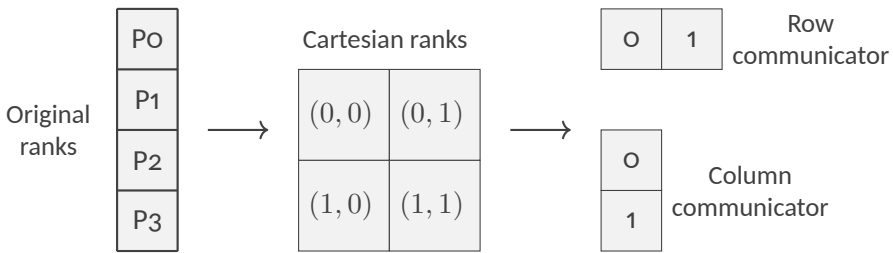
- `comm`: input communicator
- `color`: integer that determines the new communicator grouping
- `key`: integer that determines the rank ordering in the new communicator
- `newcomm`: output new communicator
- `ierr`: error code

Details and signature of the new MPI functions

The subroutine `MPI_Comm_split` has signature:

```
call MPI_Comm_split(comm, color, key, newcomm, ierr)
```

This subroutine **splits an existing communicator** into multiple new communicators based on the `color` parameter, allowing for the creation of row and column communicators from the Cartesian grid.



An example of communicator creation and splitting


```
program communicator_example
  use mpi
  use iso_fortran_env, only: output_unit
  implicit none
  integer :: ierr, rank, size
  integer :: cart_comm, row_comm, col_comm
  integer :: dims(2), coords(2)
  logical :: periods(2)
  integer :: myx, myy
  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
```

An example of communicator creation and splitting


```
! Define process grid dimensions
dims(1) = 2  ! Number of process rows
dims(2) = size / 2  ! Number of process columns
periods(1) = .false.
periods(2) = .false.

! Create Cartesian communicator
call MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, .true.,
↪  cart_comm, ierr)
call MPI_Cart_coords(cart_comm, rank, 2, coords, ierr)
```

An example of communicator creation and splitting


```
  ! Split into row and column communicators
  call MPI_Comm_split(cart_comm, coords(1), coords(2), row_comm,
  ↪  ierr)
  call MPI_Comm_split(cart_comm, coords(2), coords(1), col_comm,
  ↪  ierr)
  ! Print ranks in each communicator for verification
  call MPI_Comm_rank(row_comm, myx, ierr)
  call MPI_Comm_rank(col_comm, myy, ierr)
  write(output_unit, *) 'Global Rank:', rank, 'Row Comm Rank:', myx,
  ↪  'Col Comm Rank:', myy
  call MPI_Finalize(ierr)
end program communicator_example
```

An example of communicator creation and splitting

We can compile and run this program with 4 processes:

```
mpifort -o communicator_example communicator_example.f90
mpirun -np 4 ./communicator_example
```

The result will be something like:

```
Global Rank:      3 Row Comm Rank:        1 Col Comm Rank:      1
Global Rank:      0 Row Comm Rank:        0 Col Comm Rank:      0
Global Rank:      1 Row Comm Rank:        1 Col Comm Rank:      0
Global Rank:      2 Row Comm Rank:        0 Col Comm Rank:      1
```

Packaging these communicators in our descriptor

To facilitate the use of these communicators in our distributed matrix operations, we can add them as fields in our `descriptor` type:

```
type :: descriptor
  integer :: comm      ! MPI communicator
  integer :: M, N      ! Global number of rows/columns
  integer :: MB, NB    ! Block size in rows/columns
  integer :: P_r, P_c  ! Number of process rows/columns
  integer :: LLD       ! Leading dimension of local array
  integer :: RSRC, CSRC     ! Row, Column source process
  integer :: cart_comm, row_comm, col_comm ! Cached communicators
  integer :: myrow, mycol
end type descriptor
```

And modify the initialization routine to set these fields accordingly.

Modifying the descriptor init to include communicators

We can modify the `init` subroutine to create and store the communicators in the descriptor:

```
subroutine init(desc, comm, M, N, MB, NB, P_r, P_c, RSRC, CSRC)
  implicit none
  class(descriptor), intent(out) :: desc
  integer, intent(in) :: comm, M, N, MB, NB, P_r, P_c, RSRC, CSRC
  ! Local variables
  integer :: ierr, dims(2), coords(2)
  integer :: size
  logical :: periods(2)
```

Modifying the descriptor init to include communicators


```
! standard initialization (same as before)
desc%comm = comm
desc%M = M
desc%N = N
desc%MB = MB
desc%NB = NB
desc%P_r = P_r
desc%P_c = P_c
desc%RSRC = RSRC
desc%CSRC = CSRC
desc%LLD = ((M + P_r * MB - 1) / (P_r * MB)) * MB
```

Modifying the descriptor init to include communicators

To create the Cartesian, row, and column communicators, we need to check if the distribution is really of a 2D nature (i.e., $P_r > 1$ or $P_c > 1$):

```
! Create Cartesian communicators: created for any 2D/1D distribution
if ( P_r > 1 .or. P_c > 1 ) then
    ! Define process grid dimensions
    dims(1) = P_r
    dims(2) = P_c
    periods(1) = .false.
    periods(2) = .false.
    call MPI_Cart_create(comm, 2, dims, periods, .true.,
    ↪  desc%cart_comm, ierr)
    if ( ierr /= MPI_SUCCESS ) then
```

Modifying the descriptor init to include communicators


```
    print *, 'Error creating Cartesian communicator'
    call MPI_Abort(comm, ierr, stat)
    stop
  end if
  if ( desc%cart_comm == MPI_COMM_NULL ) then
    desc%row_comm = MPI_COMM_NULL
    desc%col_comm = MPI_COMM_NULL
    desc%my_row = -1
    desc%my_col = -1
    return
  end if
  call MPI_Comm_rank(desc%cart_comm, rank, ierr)
```

Modifying the descriptor init to include communicators


```
if ( ierr /= MPI_SUCCESS ) then
  print *, 'Error getting rank in Cartesian communicator'
  call MPI_Abort(comm, ierr, stat)
  stop
end if

call MPI_Cart_coords(desc%cart_comm, rank, 2, coords, ierr)
if ( ierr /= MPI_SUCCESS ) then
  print *, 'Error getting coordinates in Cartesian communicator'
  call MPI_Abort(comm, ierr, stat)
  stop
end if
```

Modifying the descriptor init to include communicators


```
    desc%my_row = coords(1)
    desc%my_col = coords(2)

    ! Split into row and column communicators
    call MPI_Comm_split(desc%cart_comm, coords(1), coords(2),
    ↪  desc%row_comm, ierr)
    call MPI_Comm_split(desc%cart_comm, coords(2), coords(1),
    ↪  desc%col_comm, ierr)
else
    ! For 1D distributions, use the original communicator
    desc%cart_comm = comm
    desc%row_comm = comm
```

Modifying the descriptor init to include communicators


```
    desc%col_comm = comm
    desc%my_row = 0
    desc%my_col = 0
end if
end subroutine init
```

- This modification ensures that each distributed matrix has access to the necessary communicators for efficient parallel operations.
- The process coordinates `myrow` and `mycol` are also stored for easy reference.
- We have added error checks to ensure the communicator creation is successful.

Global to Local and Local to Global Index Mapping

Now that we have the communicators set up, we also need to implement the **global to local** and **local to global** index mapping functions.

These functions are useful to query and write specific elements of the distributed matrix.

```
subroutine global_to_local(desc, i_global, j_global, &
  p_r, p_c, i_local, j_local)
  implicit none
  class(descriptor), intent(in) :: desc
  integer, intent(in)  :: i_global, j_global
  integer, intent(out) :: p_r, p_c
  integer, intent(out) :: i_local, j_local

  integer :: ig, jg          ! 0-based global element indices
  integer :: ib, jb          ! absolute block indices
```

Global to Local and Local to Global Index Mapping


```
integer :: off_i, off_j     ! offset inside block (0-based)
integer :: first_b_r, first_b_c
integer :: lbr, lbc         ! local block indices

! Convert to 0-based element indices
ig = i_global - 1
jg = j_global - 1
! Absolute block indices
ib = ig / desc%MB
jb = jg / desc%NB
! Offsets inside their blocks
off_i = mod(ig, desc%MB)
off_j = mod(jg, desc%NB)
```

Global to Local and Local to Global Index Mapping


```
  ! Owning process coordinates (0..P_r-1, 0..P_c-1)
  p_r = mod(ib - desc%RSRC, desc%P_r)
  p_c = mod(jb - desc%CSRC, desc%P_c)
  ! First absolute block index assigned to that process
  first_b_r = mod(desc%RSRC + p_r, desc%P_r)
  first_b_c = mod(desc%CSRC + p_c, desc%P_c)
  ! Local block index on owning process (non-negative)
  lbr = (ib - first_b_r) / desc%P_r
  lbc = (jb - first_b_c) / desc%P_c
  ! Local 1-based indices in Fortran storage
  i_local = lbr * desc%MB + off_i + 1
  j_local = lbc * desc%NB + off_j + 1
end subroutine global_to_local
```

Global to Local and Local to Global Index Mapping

While the converse mapping is given by:

```
subroutine local_to_global(desc, p_r, p_c, &
  i_local, j_local, &
  i_global, j_global)
  implicit none
  class(descriptor), intent(in) :: desc
  integer, intent(in)  :: p_r, p_c
  integer, intent(in)  :: i_local, j_local
  integer, intent(out) :: i_global, j_global

  integer :: il, jl          ! 0-based local indices
  integer :: lbr, lbc        ! local block indices
  integer :: off_i, off_j    ! offset inside local block
```

Global to Local and Local to Global Index Mapping


```
integer :: first_b_r, first_b_c
integer :: ib, jb          ! absolute block indices

! Convert to 0-based local indices
il = i_local - 1
jl = j_local - 1
! Local block indices and offsets (0-based)
lbr = il / desc%MB
lbc = jl / desc%NB
off_i = mod(il, desc%MB)
off_j = mod(jl, desc%NB)
! First absolute block index assigned to (p_r,p_c)
first_b_r = mod(desc%RSRC + p_r, desc%P_r)
first_b_c = mod(desc%CSRC + p_c, desc%P_c)
```

Global to Local and Local to Global Index Mapping


```
  ! Reconstruct absolute block indices
  ib = first_b_r + lbr * desc%P_r
  jb = first_b_c + lbc * desc%P_c
  ! Convert back to 1-based global indices
  i_global = ib * desc%MB + off_i + 1
  j_global = jb * desc%NB + off_j + 1
end subroutine local_to_global
```

Which can be used as needed in our distributed matrix operations.

The prototype of the distributed GEMV subroutine

We have now done all the necessary preparations to implement the distributed GEMV.

The **prototype** of the distributed GEMV subroutine can be defined as follows:

```
subroutine gemv_distributed(mat, x, alpha, y, beta)
    class(distributed_matrix), intent(in) :: mat
    class(distributed_matrix), intent(in) :: x
    class(distributed_matrix), intent(inout) :: y
    real(wp), intent(in) :: alpha, beta
end subroutine gemv_distributed
```

Where:

- `mat` is the distributed matrix A,
- `x` and `y` are the distributed input and output vectors, respectively,
- `alpha` and `beta` are scalars for the operation $y = \alpha Ax + \beta y$.

Temporary storage for the broadcasted vector

We need to create a **temporary storage** for the broadcasted vector segments, and to reduce the code redundancy, we can define a *local variable* for the descriptor of the matrix:

```
type(descriptor) :: desc
integer :: mloc, nloc, ierr, stat
real(wp), allocatable :: xbuf(:), y_partial(:), y_local(:)

desc = mat%desc
mloc = size(mat%local_data, 1)
nloc = size(mat%local_data, 2)
```

When do we need to allocate the temporary buffer `xbuf`?

Temporary storage for the broadcasted vector

We need to allocate the temporary buffer `xbuf` on all processes:

```
allocate(xbuf(nloc), stat=stat)
if (stat /= 0) then
    print *, 'Error allocating xbuf'
    call MPI_Abort(mat%desc%comm, stat, ierr)
end if
```

However, only the processes in the **source column** of the vector need to copy their local data into this buffer:

```
if (desc%my_row == desc%RSRC) then
    xbuf = x%local_data(:,1)
end if
```

And now we are ready to perform the broadcast along the process rows.

Executing the broadcast down the process columns

The call is

```
call MPI_Bcast(xbuf, nloc, MPI_REAL8, desc%RSRC, desc%col_comm, ierr)
```

where:

- `xbuf` is the temporary buffer for the broadcasted vector segment, it contains the local portion of the vector on the source process and will be filled on all other processes,
- `nloc` is the size of the local portion of the vector,
- `MPI_REAL8` is the MPI datatype for double precision real numbers,
- `desc%RSRC` is the row source process for the vector—the process which owns the local data to be broadcasted,
- `desc%col_comm` is the column communicator.
- `ierr` is the error code.

Executing the local GEMV

We now have the necessary data to perform the local GEMV operation:

$$\mathbf{y}_{\text{partial}} = \alpha \mathbf{A}_{\text{local}} \mathbf{x}_{\text{buf}}$$

```
allocate(y_partial(mloc), stat=stat)
if (stat /= 0) then
    print *, 'Error allocating y_partial'
    call MPI_Abort(mat%desc%comm, stat, ierr)
end if
call dgemv('N', mloc, nloc, alpha, mat%local_data, mloc, xbuf, 1,
↪  0.0_wp, y_partial, 1)
```

Now on each process we have the local contribution to the output vector $\mathbf{y}$.

Reducing the partial results along process columns

The last two step are

1. Reduce the partial results along the process columns,
2. Scale and update the output vector $\mathbf{y}$.

The **reduction** can be performed using:

```
if (desc%my_row == desc%RSRC) then
allocate(y_local(mloc), stat=stat)
  if (stat /= 0) then
      print *, 'Error allocating y_local'
      call MPI_Abort(mat%desc%comm, stat, ierr)
  end if
end if
call MPI_Reduce(y_partial, y_local, mloc, MPI_REAL8, MPI_SUM,
↪  mat%desc%RSRC, mat%desc%col_comm, ierr)
```

Reducing the partial results along process columns

The last two step are

1. Reduce the partial results along the process columns,
2. Scale and update the output vector $\mathbf{y}$.

The **scaling and update** of the output vector can be done using:

```
if (desc%my_row == desc%RSRC) then
    ! Scale existing y by beta
    y%local_data(:,1) = beta * y%local_data(:,1)
    ! Add the reduced result
    y%local_data(:,1) = y%local_data(:,1) + y_local
    deallocate(y_local)
end if
```

Let’s summarize the distributed GEMV implementation

1. Create communicators for process rows and columns.
2. In the distributed GEMV subroutine:
   - 2.1 Allocate a temporary buffer for the broadcasted vector segment.
   - 2.2 Copy local vector data into the buffer on the source process.
   - 2.3 Broadcast the vector segment along the process columns.
   - 2.4 Perform the local GEMV operation to compute the partial result.
   - 2.5 Reduce the partial results along the process columns.
   - 2.6 Scale and update the output vector on the source process.

As you can see, implementing distributed GEMV requires **careful management** of data distribution and communication patterns to ensure efficiency and correctness in a parallel computing environment.

Running a test

2 Distributed matrices

We can now run a test of our distributed GEMV implementation by computing:

$$\mathbf{y} = 1.0\,A\mathbf{x} + 0.0\,\mathbf{y}, \text{ with } \{(A)_{i,j} = 1.0\}_{i,j=1}^{n}, \ \mathbf{x} = \mathbf{1}.$$

```
program test_distributed_gemv
   use mpi
   use distributed_gemv
   use iso_fortran_env, only: wp => real64
   implicit none
   integer :: ierr, rank, world_size, colrank, rowrank
   ! Matrix size
   integer, parameter :: M = 900, N = 900
   ! Matrix descriptor
   type(descriptor) :: desc_matrix, desc_vector
   type(distributed_matrix) :: mat, x, y
```

Running a test

2 Distributed matrices

```
! Check variables
integer :: i
logical :: correct

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

! Create a descriptor for a square matrix distributed over a 3x3 process grid
call desc_matrix%init(MPI_COMM_WORLD, M, N, M/3, N/3, 3, 3, 0, 0)
! Initialize distributed matrix
call mat%init_matrix(desc_matrix, 1.0_wp)

! Create a distributed vector (which is an M x 1 distributed matrix)
call desc_vector%init(MPI_COMM_WORLD, M, 1, M/3, 1, 3, 1, 0, 0)
```

Running a test

2 Distributed matrices

```
call x%init_matrix(desc_vector, 1.0_wp)
call y%init_matrix(desc_vector, 0.0_wp)

! Perform distributed matrix-vector multiplication
call mat%gemv_distributed(x, 1.0_wp, y, 0.0_wp)

! Check results: we are multiplying a matrix of ones by a vector of ones
! So the result should be a vector of size M with all entries equal to N
! we need to check only the local part of y only on the ranks that own data
correct = .true.
call MPI_Comm_rank(desc_matrix%col_comm, colrank, ierr)

if (colrank == 0) then
   do i = 1, size(y%local_data, 1)
      if (y%local_data(i,1) /= real(N, wp)) then
```

Running a test

2 Distributed matrices

```
            correct = .false.
            exit
         end if
      end do
      if (correct) then
         print *, 'Test passed: Distributed GEMV result is correct on rank', rank,
         ↪  'colrank', colrank
      else
         print *, 'Test failed: Distributed GEMV result is incorrect on rank', rank,
         ↪  'colrank', colrank
      end if
   end if

   call MPI_Finalize(ierr)
end program test_distributed_gemv
```

We compile and run the test

We can **compile** and **run** the test program using:

```
mpifort -o distributed_gemv distributed_gemv.f90 -lopenblas
```

And then **run** the program with 9 processes (3x3 grid):

```
mpirun -np 9 ./distributed_gemv
```

Which should output:

```
Test passed: Distributed GEMV result is correct on rank  2 colrank  0
Test passed: Distributed GEMV result is correct on rank  0 colrank  0
Test passed: Distributed GEMV result is correct on rank  1 colrank  0
```

Run another test

We can run **another test** with a non unifor vector x:

```
if (colrank == 0) then
  y%local_data(:,1) = 1.0_wp
  ! Reinitialize x to have values ((rank-1)*(N/3)+1):(rank)*(N/3)
  do i = 1, size(x%local_data, 1)
    x%local_data(i,1) = real((rowrank+1)*(N/3)+i, wp)
    ! write(*,*) 'Rank', rank, 'x local(', i, ') =', x%local_data(i,1)
  end do
end if
```

And perform the distributed GEMV again:

```
call mat%gemv_distributed(x, 1.0_wp, y, 1.0_wp)
```

The expected result is now:

$$y_i = \sum_{j=1}^{N} A_{i,j}x_j + y_i = \sum_{j=1}^{N} 1.0 \cdot x_j + 1.0 = \sum_{j=1}^{N} x_j = \frac{N(N+1)}{2} + 1.0 = 405451.0$$

Check the result of the second test


```
correct = .true.
if (colrank == 0) then
  do i = 1, size(y%local_data, 1)
    if (y%local_data(i,1) /= real(N*(N+1), wp)/2.0+1.0) then
      correct = .false.
      exit
    end if
  end do
  if (correct) then
      print *, 'Test passed: Distributed GEMV with beta=1.0 result is
       ↪  correct on rank', rank, 'colrank', colrank
  else
      print *, 'Test failed: Distributed GEMV with beta=1.0 result is
       ↪  incorrect on rank', rank, 'colrank', colrank
  end if
end if
```

Run and check the second test

We can now **run** the modified test program again:

```
mpirun -np 9 ./distributed_gemv
```

Which should output:

```
Test passed: Distributed GEMV with beta=1.0 result is correct on rank 1
↪  colrank 0
Test passed: Distributed GEMV with beta=1.0 result is correct on rank 2
↪  colrank 0
Test passed: Distributed GEMV with beta=1.0 result is correct on rank 0
↪  colrank 0
```

Summary, conclusions, and outlook

We have seen:

- How to implement a descriptor for distributed matrices,
- How to create communicators for process rows and columns,
- How to implement a distributed GEMV operation using MPI communication routines

Next **s**teps are:

- Explore the ScaLAPACK library for distributed linear algebra,
- Implement distributed matrix-matrix multiplication (GEMM),
- Investigate the performance of distributed operations.