MPI Cartesian Communicators

Structured Process Topologies for Domain Decomposition

1. Motivation and Context

Many scientific and engineering problems operate on regular, structured grids: finite difference methods for PDEs, stencil computations, image processing, and particle-mesh methods in astrophysics. When parallelizing these applications via domain decomposition, we partition the computational domain into sub-domains, each handled by a distinct MPI process.

The fundamental challenge is **neighbor communication**: processes must exchange boundary data (ghost zones, halo cells) with their neighbors. While we could manually compute neighbor ranks using arithmetic on process coordinates, this approach is error-prone, non-portable, and fails to leverage potential hardware optimizations.

1.1 Why Not Manual Rank Arithmetic?

Consder a 2D grid decomposition with dimensions (Px × Py). Given a process with coordinates (i, j), its neighbors are:

```
// Manual neighbor calculation (fragile approach)
int rank_left  = (i > 0)    ? i-1 + j*Px : MPI_PROC_NULL;
int rank_right = (i < Px-1) ? i+1 + j*Px : MPI_PROC_NULL;
int rank_down  = (j > 0)    ? i + (j-1)*Px : MPI_PROC_NULL;
int rank_up    = (j < Py-1) ? i + (j+1)*Px : MPI_PROC_NULL;
```

The possible issues, or inconveniencies, with this approach are:

1. **Boundary handling**: Explicit conditionals for edges; periodic boundaries add complexity
2. **Dimension ordering**: Row-major vs column-major rank assignment affects all calculations
3. **Scalability**: 3D or higher dimensions require nested conditionals
4. **Optimization**: MPI implementations cannot optimize message routing without topology knowledge

2. The Cartesian Communicator Model

MPI provides **virtual topologies** to express logical process arrangements. A Cartesian topology maps processes onto a regular n-dimensional grid, providing:

- Automatic coordinate ↔ rank translation
- Built-in periodic boundary support
- Neighbor lookup via relative displacement
- Hints to MPI for process placement optimization (implementation-dependent)

2.1 Conceptual Model

A Cartesian communicator creates a *new communicator* from an existing one, assigning each process a unique coordinate tuple (c_0, c_1, ..., c_{n-1}) where $0 \leq c_i <$ dims[i]. The topology can be periodic along any dimension, creating a torus-like connectivity.

Non-periodic 3×2 Grid:

```
     j=0      j=1
   +-------+-------+
i=0|  R0   |  R1   |     Rank = i + j*3
   +-------+-------+
i=1|  R2   |  R3   |     R0:(0,0)  R1:(1,0)
   +-------+-------+     R2:(0,1)  R3:(1,1)
i=2|  R4   |  R5   |     R4:(0,2)  R5:(1,2)
   +-------+-------+
```

Periodic 3×2 Grid (torus):

```
R5's right neighbor → R4  (wraps in i-direction)
R4's down neighbor  → R0  (wraps in j-direction)
```

3. Core API Functions

3.1 MPI_Dims_create

Computes a "balanced" distribution of processes across dimensions:

```
int MPI_Dims_create(int nnodes,    // Total number of processes
                    int ndims,     // Number of dimensions
                    int dims[]);   // IN/OUT: dimension sizes
```

Key behavior: Pre-set dimensions (dims[i] > 0) are preserved; zero entries are computed. The algorithm tries to minimize surface-to-volume ratio for communication efficiency.

```
// Example: 12 processes, 2D grid
int dims[2] = {0, 0};
MPI_Dims_create(12, 2, dims);  // Result: dims = {4, 3} or {3, 4}

// Constrained: force 4 columns
int dims2[2] = {0, 4};
MPI_Dims_create(12, 2, dims2); // Result: dims2 = {3, 4}
```

⚠ **Caution:** MPI_Dims_create fails if nnodes is not divisible by the product of pre-set dimensions. Always verify dims[i] > 0 for all i after the call.

3.2 MPI_Cart_create

Creates a new communicator with Cartesian topology:

```
int MPI_Cart_create(
    MPI_Comm comm_old,       // Input communicator
    int ndims,               // Number of dimensions
    const int dims[],        // Size of each dimension
    const int periods[],     // Periodicity flags (0 or 1)
    int reorder,             // Allow rank reordering?
    MPI_Comm *comm_cart);    // Output: new Cartesian comm
```

Parameters explained:

1. `periods[i]`: If non-zero, dimension i wraps around (periodic boundary)
2. `reorder`: If non-zero, MPI may reassign ranks to optimize for hardware topology
3. If `size(comm_old) > product(dims)`, excess processes receive `MPI_COMM_NULL`

Best Practice: Set reorder=1 in production code. This allows MPI implementations (especially on clusters with complex interconnects like fat-trees or torus networks) to map logically adjacent processes to physically adjacent nodes, reducing communication latency.

3.3 Coordinate and Rank Translation

```
// Get my coordinates in the Cartesian grid
int MPI_Cart_coords(MPI_Comm comm, int rank, int ndims, int coords[]);

// Get rank from coordinates
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank);
```

These functions handle periodic wrapping automatically. For periodic dimensions, coordinates outside [0, dim-1] wrap correctly:

```
// In a periodic 4×4 grid:
int coords[2] = {-1, 5};  // Wraps to (3, 1)
int rank;
MPI_Cart_rank(cart_comm, coords, &rank);  // Returns valid rank
```

3.4 MPI_Cart_shift, the real trick

The most frequently used function for neighbor communication:

```
int MPI_Cart_shift(
    MPI_Comm comm,          // Cartesian communicator
    int direction,          // Dimension index (0, 1, ...)
    int disp,               // Displacement (+1 = forward, -1 = backward)
    int *rank_source,       // Rank to receive FROM
    int *rank_dest);        // Rank to send TO
```

Semantics: For a shift in dimension d with displacement disp, this returns the ranks for a "shift" pattern where data flows in one direction. If the shift would go outside a non-periodic boundary, `MPI_PROC_NULL` is returned, which is safe to use in MPI_Send/Recv (they become no-ops).

Understanding direction and displacement:

```
2D Grid with coords (x, y):      Shift Results:

     direction=0 (x-axis)        For process at (1,1):
     ─────────────────→          MPI_Cart_shift(comm, 0, +1, &src, &dst)
     │                           src = rank at (0,1)  // receive from left
dir=1│     (0,0) (1,0) (2,0)     dst = rank at (2,1)  // send to right
(y)  │     (0,1) (1,1) (2,1)
     ↓     (0,2) (1,2) (2,2)     MPI_Cart_shift(comm, 1, -1, &src, &dst)
                                 src = rank at (1,2)  // receive from below
                                 dst = rank at (1,0)  // send to above
```

4. Complete Working Example: 2D Heat Diffusion

This example implements a 2D heat diffusion solver using explicit finite differences. Each process owns a rectangular subdomain and exchanges boundary rows/columns with neighbors.
See the .c file in the folder

5. Advanced Patterns and Techniques

5.1 Sub-communicators via MPI_Cart_sub

Extract lower-dimensional slices from a Cartesian communicator. Useful for reduction operations along specific dimensions:

```
int MPI_Cart_sub(MPI_Comm comm,          // Parent Cartesian comm
                 const int remain_dims[],// Dimensions to keep
                 MPI_Comm *newcomm);     // Output sub-communicator

// Example: 3D grid, extract 2D planes (XY slices at fixed z)
int remain_dims[3] = {1, 1, 0};  // Keep x and y, collapse z
MPI_Comm xy_comm;
MPI_Cart_sub(cart_comm_3d, remain_dims, &xy_comm);

// Now xy_comm groups all processes with same z coordinate
// Useful for: MPI_Allreduce within each XY plane
```

5.2 MPI_Neighbor_alltoall (MPI-3.0)

MPI-3.0 introduced "neighborhood collectives" that leverage topology information for efficient communication patterns:

```
// Exchange data with all neighbors simultaneously
// More efficient than multiple MPI_Sendrecv calls
int MPI_Neighbor_alltoall(
    const void *sendbuf,     // Send buffer (to all neighbors)
    int sendcount,           // Elements to each neighbor
    MPI_Datatype sendtype,
    void *recvbuf,           // Receive buffer (from all neighbors)
    int recvcount,
    MPI_Datatype recvtype,
    MPI_Comm comm);          // Must have topology!
```

Ordering convention: Neighbors are ordered as (dim0_prev, dim0_next, dim1_prev, dim1_next, ...). For a 2D Cartesian topology, this means (left, right, down, up) assuming dims = {nx, ny}.

5.3 Querying Topology Information

```
// Retrieve topology parameters from communicator
int ndims;
MPI_Cartdim_get(cart_comm, &ndims);  // Get number of dimensions

int dims[ndims], periods[ndims], coords[ndims];
MPI_Cart_get(cart_comm,    // Get full topology info
             ndims,
             dims,         // Sizes of each dimension
             periods,      // Periodicity flags
             coords);      // My coordinates
```

5.4 Process Reordering Considerations

When `reorder=1` in MPI_Cart_create, be aware:

- Your rank in cart_comm may differ from your rank in MPI_COMM_WORLD
- I/O operations that use world ranks must translate appropriately
- Use MPI_Comm_rank on cart_comm for topology-related operations

```
// Safe pattern: always get rank from the communicator you're using
int world_rank, cart_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_rank(cart_comm, &cart_rank);
// These may be different when reorder=1!
```

6. Performance Considerations

6.1 Communication Patterns

The 5-point stencil (2D) or 7-point stencil (3D) requires communication only with face neighbors, not diagonals. Cartesian topologies are ideal for this pattern. For stencils requiring diagonal neighbors, you may need explicit corner exchanges.

Pattern	Pros	Cons
MPI_Sendrecv	Deadlock-free, explicit control	Sequential pairs, no overlap
Isend/Irecv + Waitall	Overlapped comm, pipeline potential	More complex, buffer management
Neighbor_alltoallw	Optimized collective, concise	MPI-3 required, less portable

6.2 Derived Datatypes for Efficient Packing

For non-contiguous data (columns in row-major storage), use MPI derived datatypes instead of manual packing. This allows the MPI implementation to use optimized copy routines and potentially zero-copy protocols.

```
// Compare: Manual packing vs. derived datatype

// SLOW: Manual pack/unpack
for (int i = 0; i < n; i++) buffer[i] = u[i*ny + col];
MPI_Send(buffer, n, MPI_DOUBLE, dest, tag, comm);

// FAST: Derived datatype (MPI handles it)
MPI_Type_vector(n, 1, ny, MPI_DOUBLE, &col_type);
MPI_Type_commit(&col_type);
MPI_Send(&u[col], 1, col_type, dest, tag, comm);
```

7. Common Pitfalls and Debugging

1. **Forgetting MPI_COMM_NULL check:** When size(comm_old) > product(dims), some processes get MPI_COMM_NULL. All subsequent operations on that communicator will fail or cause undefined behavior.
2. **Dimension ordering confusion:** MPI uses row-major (C-style) ordering. dims[0] varies slowest. For a 2D grid dims[2]={Nx, Ny}, coordinates are (x, y) where x ∈ [0, Nx) varies along dimension 0.
3. **MPI_Cart_shift direction misunderstanding:** The source and dest returned are for a "shift" pattern, not just neighbor lookup. For bidirectional exchange, you typically call it twice with disp=+1 and disp=-1, or use the returned source/dest correctly with MPI_Sendrecv.
4. **Memory layout mismatch:** Ensure your array indexing matches your Cartesian coordinate convention. A common bug is swapping row/column indices relative to dims[0]/dims[1].
5. **Leaking communicators:** Always call MPI_Comm_free on created communicators. In long-running applications, leaked communicators exhaust internal MPI resources.

8. Exercises

1. **3D Extension:** Modify the 2D heat diffusion code to work in 3D. You'll need to handle 6 neighbors and create datatypes for XY-planes and XZ-planes in addition to columns.

2. **Periodic Boundary Conditions:** Implement the heat equation with periodic boundaries in all dimensions. Compare the code complexity with the non-periodic version.

3. **Neighbor Collectives:** Rewrite the halo exchange using MPI_Neighbor_alltoallw. Benchmark against the MPI_Sendrecv version.

4. **Row-wise Reduction:** Use MPI_Cart_sub to create row communicators and compute the sum of each row of a distributed 2D array.

9. Summary

Cartesian communicators provide a clean abstraction for structured domain decomposition. The key workflow is:

- Use MPI_Dims_create for balanced decomposition
- Create the topology with MPI_Cart_create (enable reordering for production)
- Use MPI_Cart_coords and MPI_Cart_rank for coordinate/rank translation
- Use MPI_Cart_shift for neighbor identification
- Leverage derived datatypes for non-contiguous communication
- Consider MPI-3 neighborhood collectives for modern implementations

These abstractions eliminate error-prone manual calculations, enable MPI runtime optimizations, and make code clearer and more maintainable. For particle-mesh codes, finite-difference solvers, and image processing pipelines, Cartesian topologies should be the default choice.