Multicore Computing
Lecture17 - MPI

SUNG KYUN KWAN UNIVERSITY

남 범 석

bnam@skku.edu

Deadlock Pitfall (1)

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

If MPI_Send is non-buffered blocking, a deadlock occurs.
MPI standard does not specify whether the implementation of MPI_Send is buffered blocking or not.

Deadlock Pitfall (2)

Circular communication

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
      MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
      MPI_COMM_WORLD);
...
```

Once again, a deadlock occurs if `MPI_Send` is non-buffered blocking.

Avoiding Deadlocks

We can break the circular wait to avoid deadlocks as follows:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                MPI_COMM_WORLD);
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                MPI_COMM_WORLD);
}
else {
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                MPI_COMM_WORLD);
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                MPI_COMM_WORLD);
}
...
```

Sending and Receiving Messages Simultaneously

- Exchange messages

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
    MPI_Datatype senddatatype, int dest, int sendtag,
    void *recvbuf, int recvcount, MPI_Datatype
    recvdatatype, int source, int recvtag,
    MPI_Comm comm, MPI_Status *status)
```

  - Requires both send and receive arguments
  - Avoid the circular deadlock problem

- Exchange messages using the same buffer

```
int MPI_Sendrecv_replace(void *buf, int count,
    MPI_Datatype datatype, int dest, int sendtag,
    int source, int recvtag, MPI_Comm comm,
    MPI_Status *status)
```

Non-blocking Sending and Receiving Messages

- Non-blocking send and receive operations

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

- Tests whether non-blocking operation is finished

```
int MPI_Test(MPI_Request *request, int *flag,
    MPI_Status *status)
```

- Waits for the operation to complete

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Avoiding Deadlocks

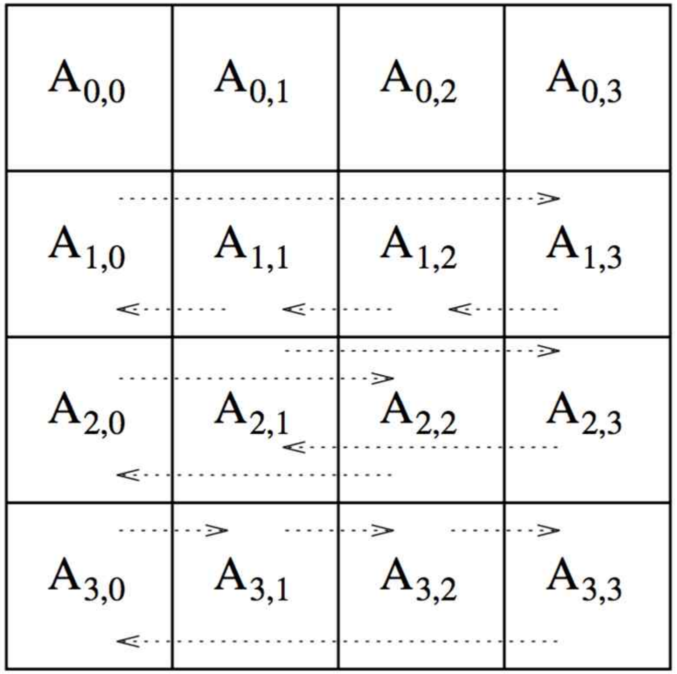
- Using non-blocking operations remove most deadlocks

```
int a[10], b[10], myrank;
MPI_Status status;
MPI_Request request1, request2;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Isend(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD, &request1);
  MPI_Isend(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD, &request2);
}
else if (myrank == 1) {
  MPI_Irecv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD, &request1);
  MPI_Irecv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD, &request2);
}
...
```

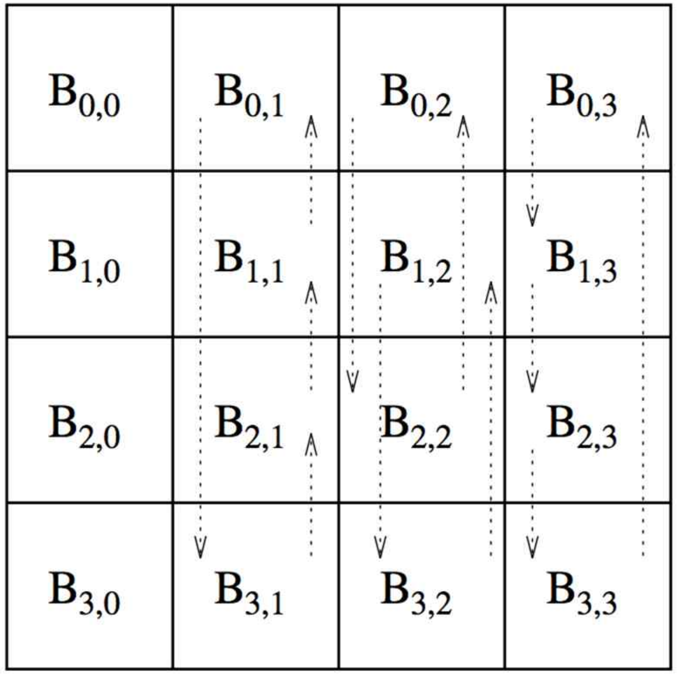
- Replacing either the send or the receive operations with non-blocking counterparts fixes this deadlock.

Overlapping Communication with Computation

- Example: Matrix-Matrix Multiplication



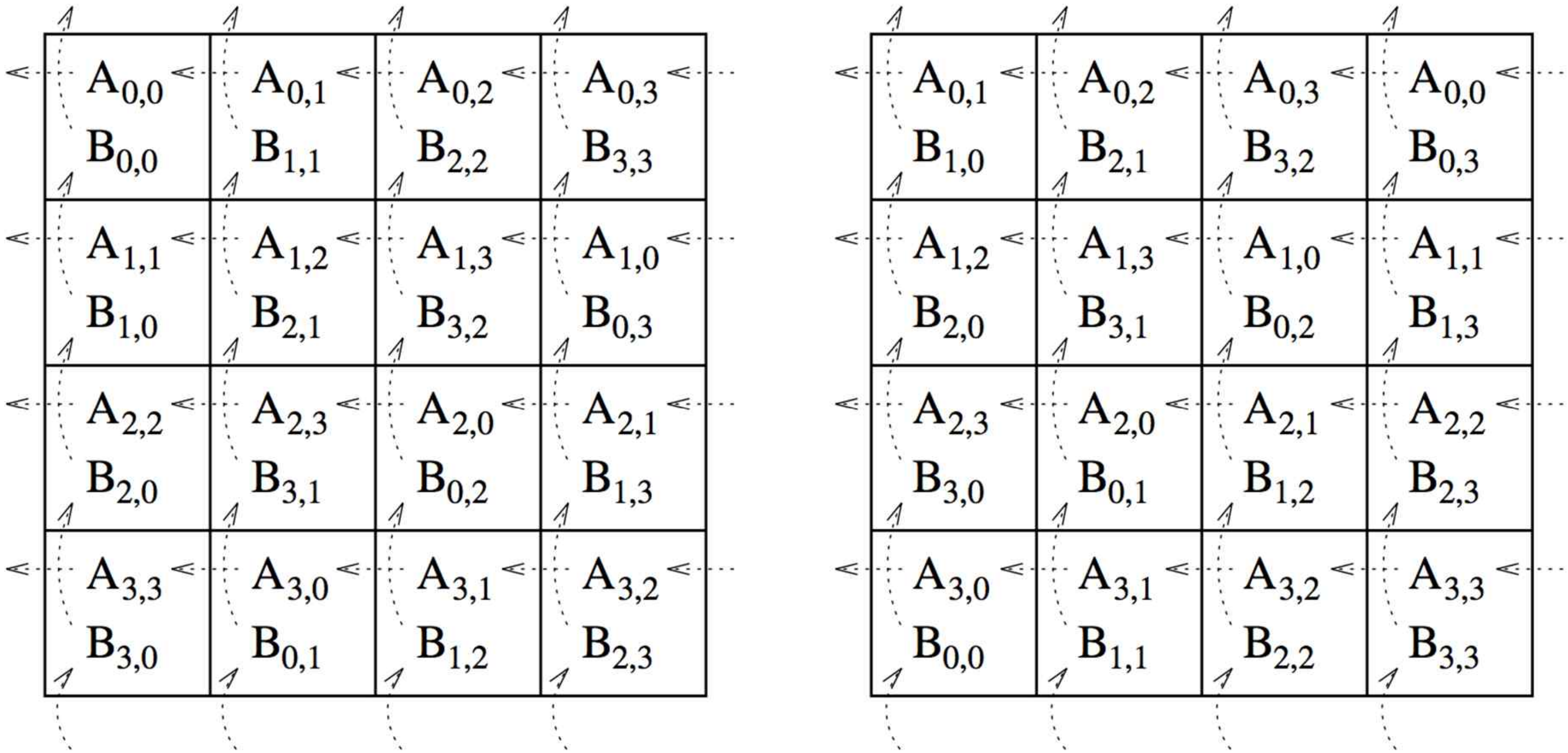
(a) Initial alignment of A



(b) Initial alignment of B

Overlapping Communication with Computation

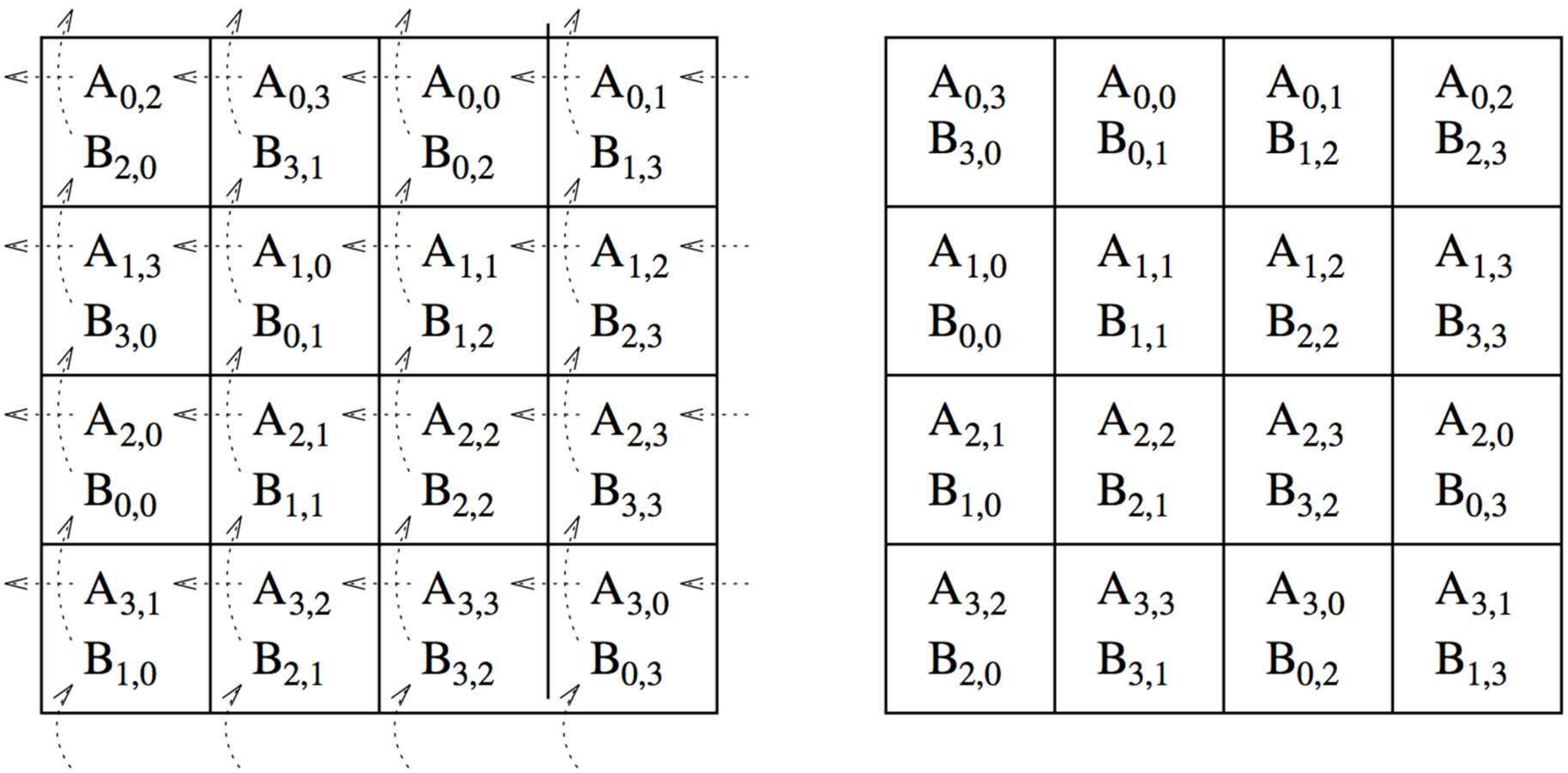
- Example: Matrix-Matrix Multiplication



(c) A and B after initial alignment

(d) Submatrix locations after first shift

- Example: Matrix-Matrix Multiplication



(e) Submatrix locations after second shift (f) Submatrix locations after third shift

Overlapping Communication with Computation

- Using blocking communications

```
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
  MatrixMultiply(nlocal, a, b, c); /*c=c+a*b*/

  /* Shift matrix a left by one */
   MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
       leftrank, 1, rightrank, 1, comm_2d, &status);

  /* Shift matrix b up by one */
   MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
       uprank, 1, downrank, 1, comm_2d, &status);
}
```

Code snippet of Cannon's matrix-matrix multiplication

Overlapping Communication with Computation

- Using non-blocking communications

```
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
  MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      leftrank, 1, comm_2d, &reqs[0]);
  MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      uprank, 1, comm_2d, &reqs[1]);
  MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      rightrank, 1, comm_2d, &reqs[2]);
  MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      downrank, 1, comm_2d, &reqs[3]);

  /* c = c + a*b */
  MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);

  for (j=0; j<4; j++)
    MPI_Wait(&reqs[j], &status);
}
```

Code snippet of Cannon's matrix-matrix multiplication

Collective Communication

- MPI provides an extensive set of functions of collective communication operations
- Operations are defined over a group corresponding to the communicator
- All processors in a communicator must call these operations

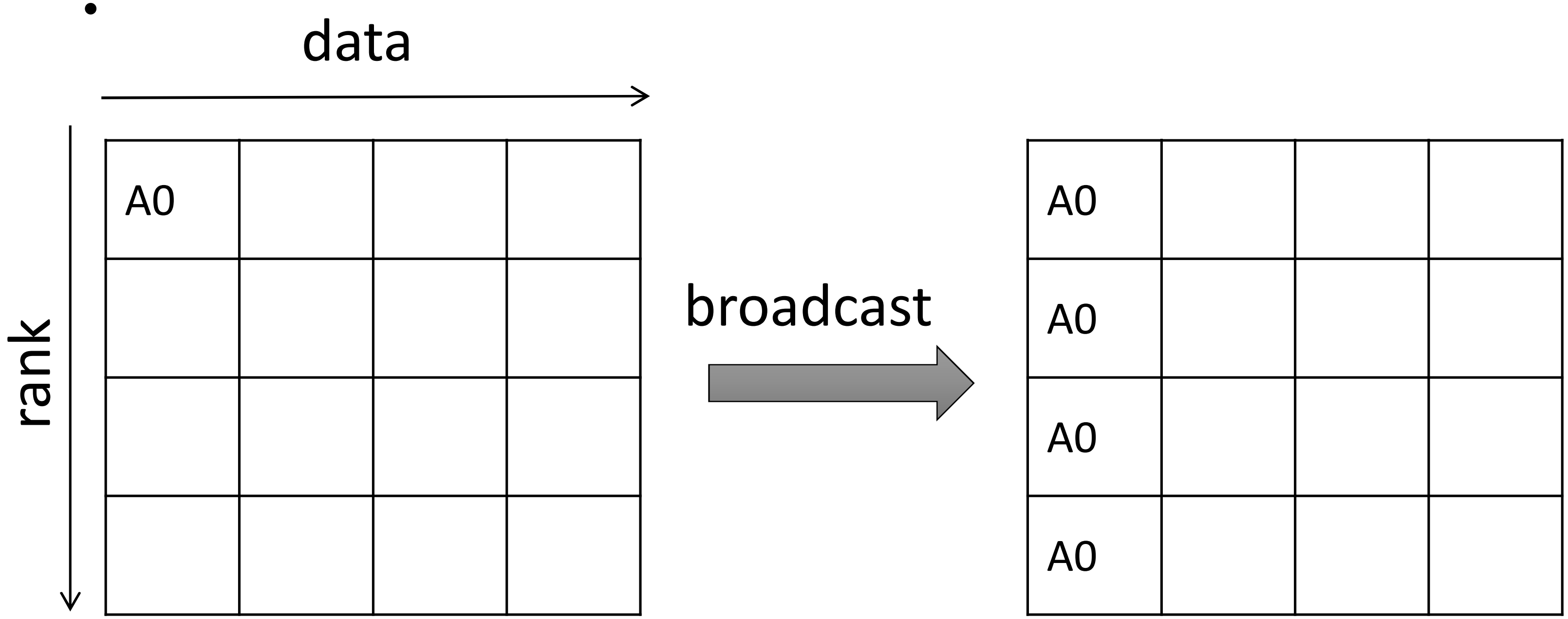
- Simple collective communication: barrier

```
int MPI_Barrier(MPI_Comm comm)
```

  - Waits until all processes arrive

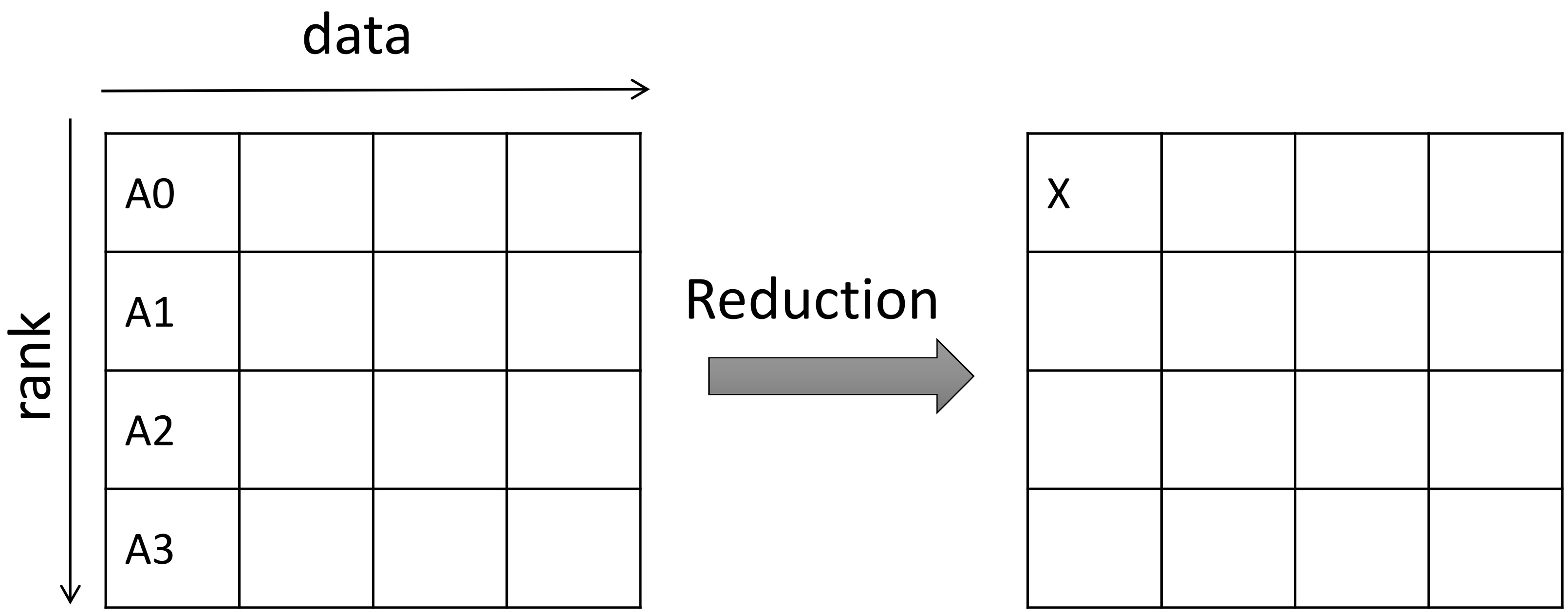
One-to-All Broadcast

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int source, MPI_Comm comm)
```



All-to-One Reduction

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int target,
               MPI_Comm comm)
```



X = A0 op A1 op A2 op A3

Predefined Reduction Operations

Operation	Meaning	Datatypes
`MPI_MAX`	Maximum	C integers and floating point
`MPI_MIN`	Minimum	C integers and floating point
`MPI_SUM`	Sum	C integers and floating point
`MPI_PROD`	Product	C integers and floating point
`MPI_LAND`	Logical AND	C integers
`MPI_BAND`	Bit-wise AND	C integers and byte
`MPI_LOR`	Logical OR	C integers
`MPI_BOR`	Bit-wise OR	C integers and byte
`MPI_LXOR`	Logical XOR	C integers
`MPI_BXOR`	Bit-wise XOR	C integers and byte
`MPI_MAXLOC`	max value-location	Data-pairs
`MPI_MINLOC`	min value-location	Data-pairs

Collective Communications (1)

- All processes must call the same collective function.
  - Ex. MPI_Recv() in P0 + MPI_Reduce() in P1 (X)
- All processes must send to the same target process
  - Ex.

Process 0

```
MPI_Reduce(in_buf,
        out_buf,
        1,
        MPI_CHAR,
        MPI_SUM,
        0,
        MPI_COMM_WORLD);
```

Mismatch!!

Process 1

```
MPI_Reduce(in_buf,
        out_buf,
        1,
        MPI_CHAR,
        MPI_SUM,
        1,
        MPI_COMM_WORLD);
```

Collective Communications (2)

- Recv buffer argument is only used in the destination process
  - But, other processes should provide the argument, even if it is NULL
  - Ex.

Process 0

```
MPI_Reduce(in_buf,
        out_buf,
        1,
        MPI_CHAR,
        MPI_SUM,
        0,
        MPI_COMM_WORLD);
```

Process 1

```
MPI_Reduce(in_buf,
        NULL,
        1,
        MPI_CHAR,
        MPI_SUM,
        0,
        MPI_COMM_WORLD);
```

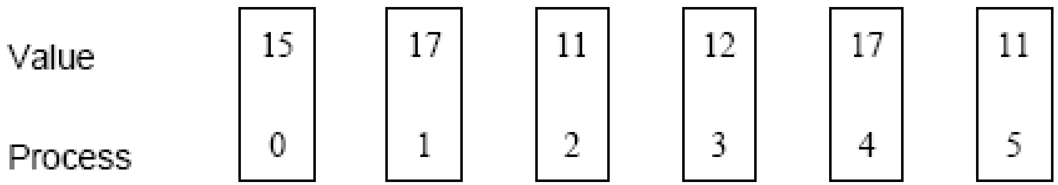
Collective vs. Point-to-Point Comm.

- Point-to-point communication
  - MPI_Send/Recv are matched on the basis of tags and ranks

- Collective communication
  - Do NOT use tags
  - They're matched solely on the basis of receiver's rank and order

MPI_MAXLOC and MPI_MINLOC

- `MPI_MAXLOC`
  - Combines pairs of values (v_i, l_i)
  - Returns the pair (v, l)
    - v is the maximum among all v_i 's
    - l is the corresponding l_i
      - (if there are more than one, it is the smallest among all these l_i 's).
- `MPI_MINLOC` does the same, except for minimum value of v_i.




```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

MPI_MAXLOC and MPI_MINLOC

MPI datatypes for data-pairs used with the MPI_MAXLOC and MPI_MINLOC reduction operations.

MPI Datatype	C Datatype
`MPI_2INT`	pair of ints
`MPI_SHORT_INT`	short and int
`MPI_LONG_INT`	long and int
`MPI_LONG_DOUBLE_INT`	long double and int
`MPI_FLOAT_INT`	float and int
`MPI_DOUBLE_INT`	double and int

MPI_MAXLOC and MPI_MINLOC

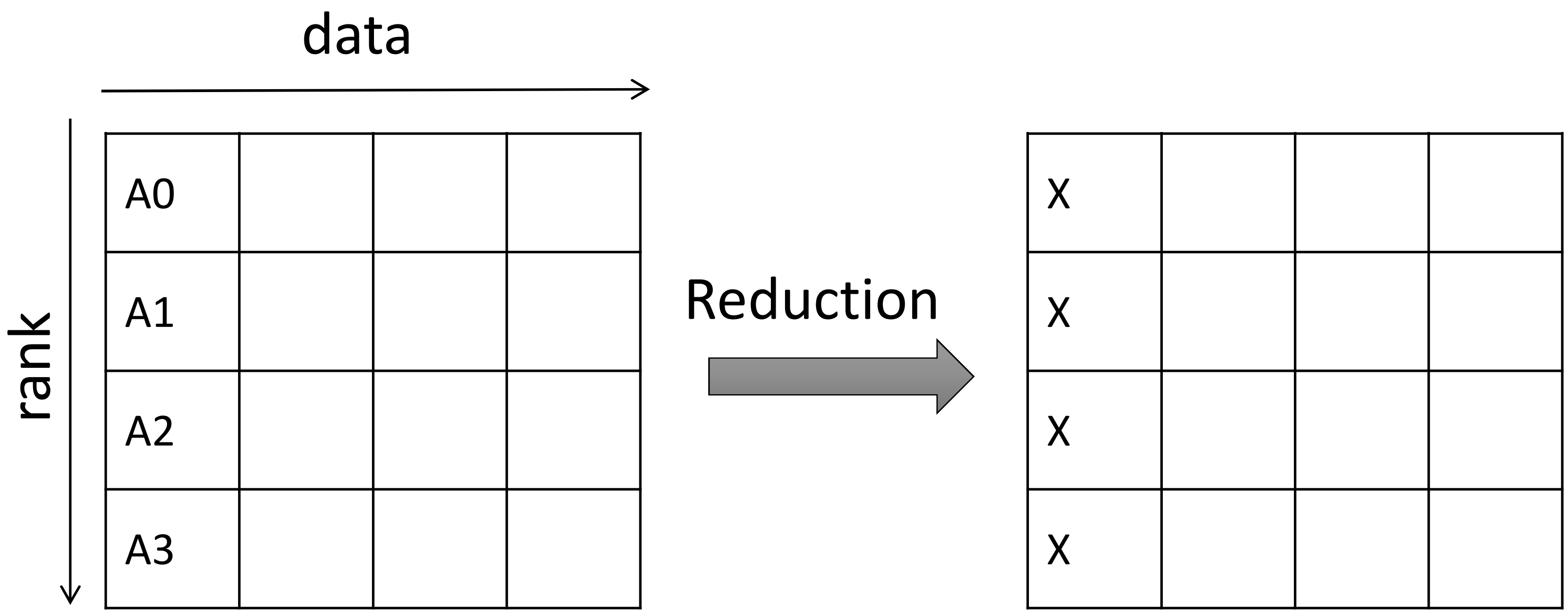
```
double ain[30], aout[30];
int ind[30];
struct {
    double val;
    int rank;
} in[30], out[30];

for (i=0; i<30; ++i) {
    in[i].val = ain[i];
    in[i].rank = myrank;
}

MPI_Reduce( in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm );
if (myrank == root) {
    for (i=0; i<30; ++i) {
        aout[i] = out[i].val;
        ind[i] = out[i].rank;
     }
 }
```

All-to-All Reduction

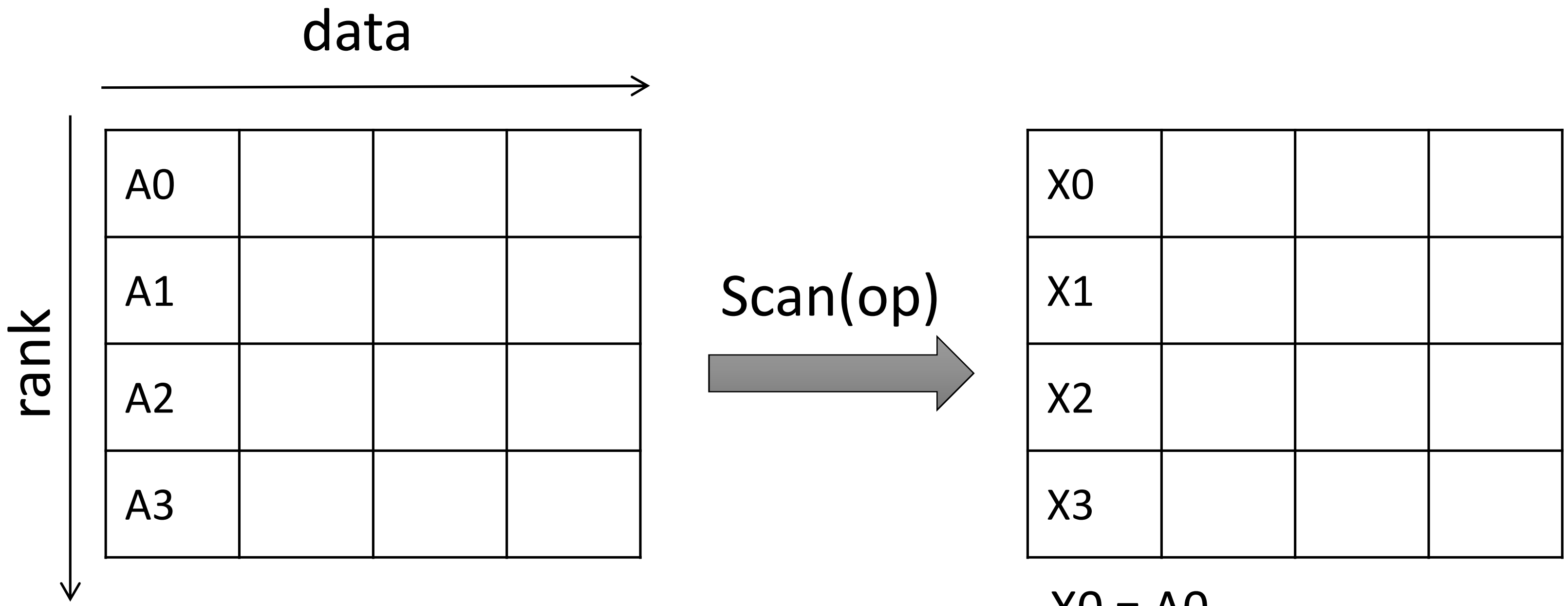
```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
  int count, MPI_Datatype datatype, MPI_Op op,
  MPI_Comm comm)
```



X = A0 op A1 op A2 op A3

Prefix Operation (Inclusive)

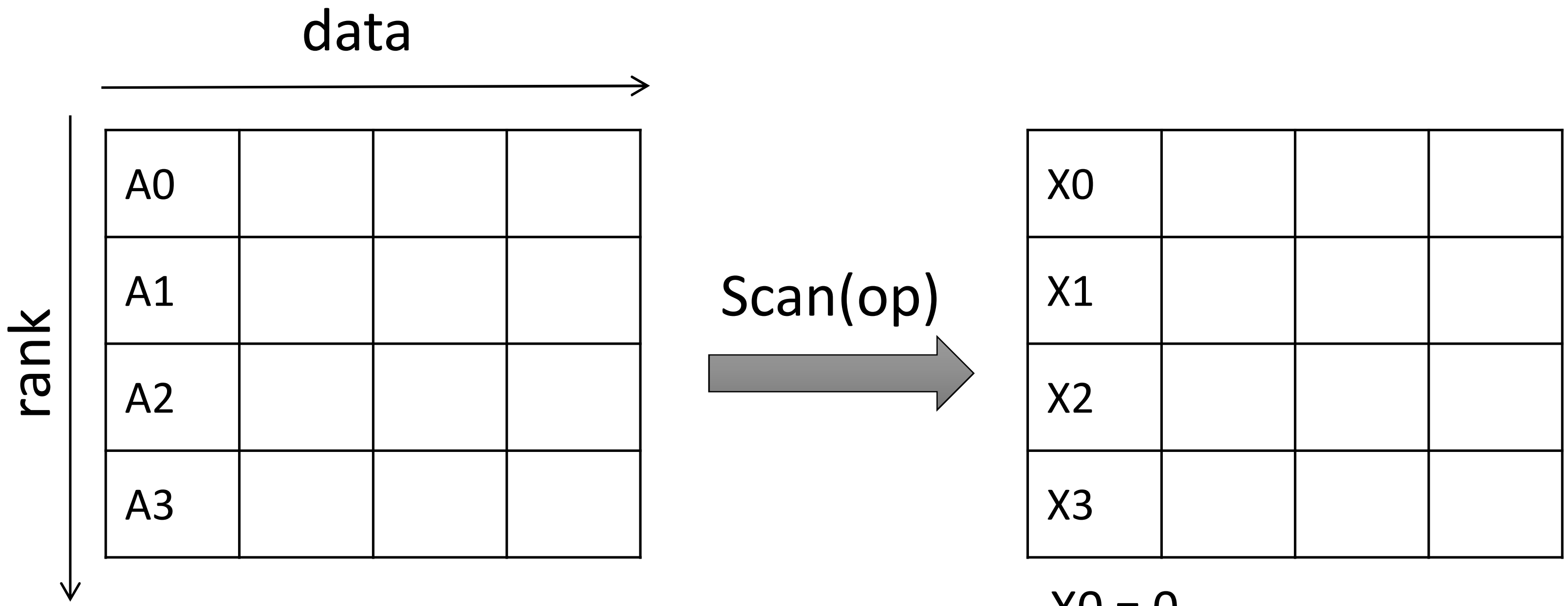
```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm)
```



X0 = A0,
X1 = A0 op A1,
X2 = A0 op A1 op A2,
X3 = A0 op A1 op A2 op A3

Prefix Operation (Exclusive)

```
int MPI_Exscan(void *sendbuf, void *recvbuf,
               int count, MPI_Datatype datatype,
               MPI_Op op, MPI_Comm comm)
```



X0 = 0
X1 = A0
X2 = A0 op A1
X3 = A0 op A1 op A2

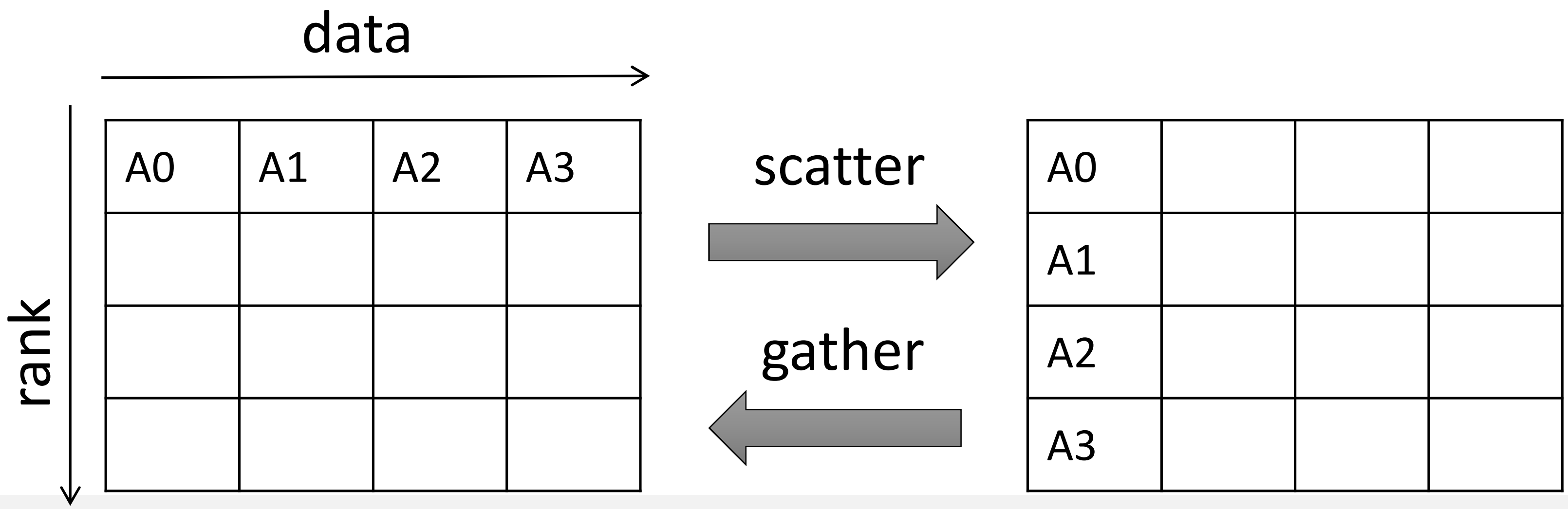
Scatter and Gather

- Gather data at one process

```
int MPI_Gather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
        int target, MPI_Comm comm)
```

- Scatter data from source to all processes

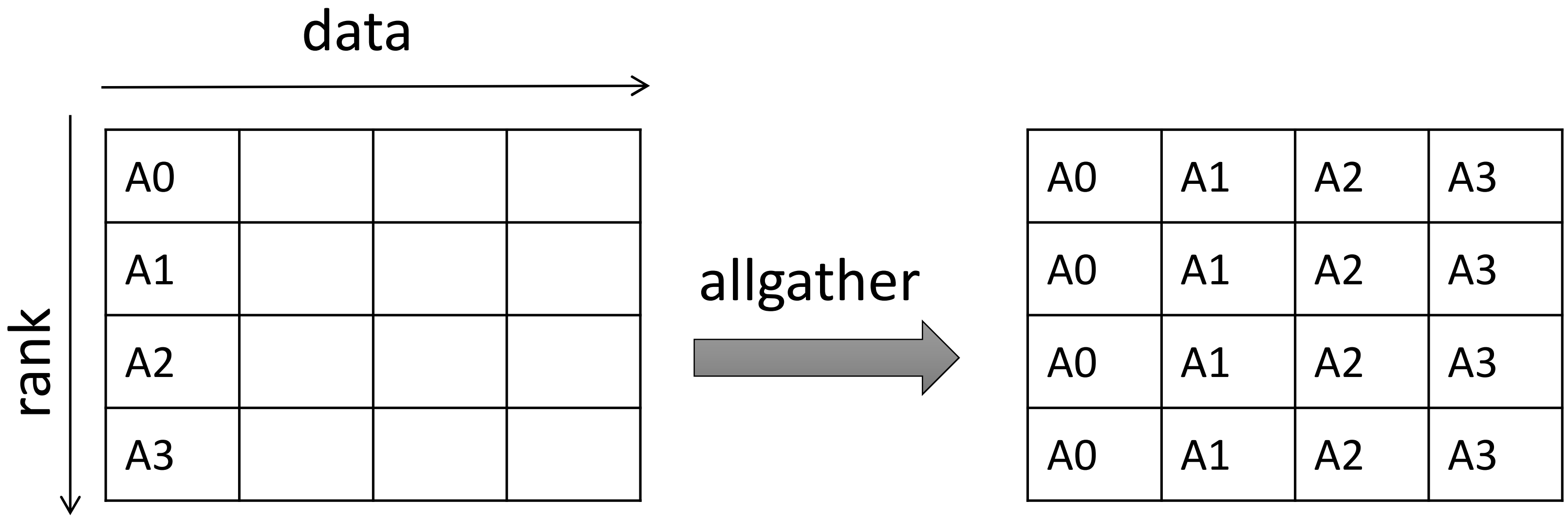
```
int MPI_Scatter(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
        int source, MPI_Comm comm)
```



Allgather

- Gather and scatter them to all processes

```
int MPI_Allgather(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, void *recvbuf,
          int recvcount, MPI_Datatype recvdatatype,
          MPI_Comm comm)
```



All-to-All

- The all-to-all personalized communication

```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
        MPI_Comm comm)
```

  - Analogous to a matrix transpose

data

rank

A0	A1	A2	A3
B0	B1	B2	B3
C0	C1	C2	C3
D0	D1	D2	D3

alltoall

A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3

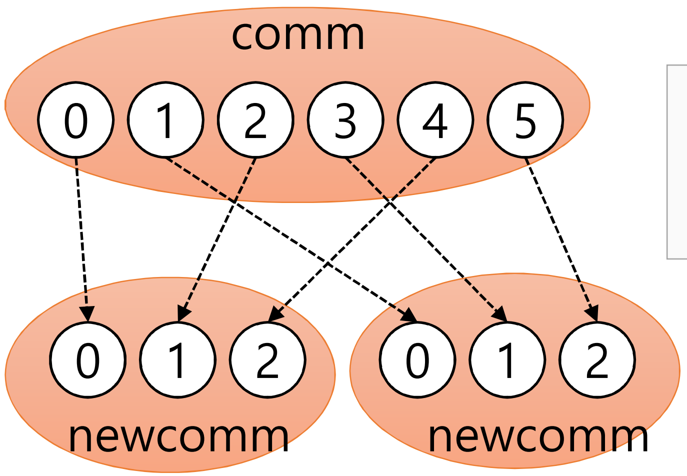
Communicators

- All MPI communication is based on a communicator which contains a context and a group
  - Contexts define a safe communication space for message-passing – viewed as system-managed tags
  - Contexts allow different libraries to co-exist
  - Group is just a set of processes
  - Processes are always referred to by the unique rank in a group

- Pre-defined communicators
  - MPI_COMM_WORLD
  - MPI_COMM_NULL // initial value, cannot be used as comm
  - MPI_COMM_SELF // contains only the local calling process

Communicator Manipulation

- Duplicate communicator
  - `MPI_Comm_dup(comm, newcomm)`
  - Create a new context with similar structure
- Partition the group into disjoint subgroups
  - `MPI_Comm_split(comm, color, key, newcomm)`
  - Each sub-comm. contains the processes with the same color
  - The rank in the sub-communicator is defined by the key




```
color = (rank % 2 == 0)? 0 : 1;
key   = rank / 2;
MPI_Comm_split(comm, color, key, &newcomm);
```

Communicator Manipulation – con't

- Obtain an existing group and free a group
  - `MPI_Comm_group(comm, group)`– create a group having processes in the specified communicator
  - `MPI_Group_free(group)` – free a group

- New group can be created by specifying members
  - `MPI_Group_incl(), MPI_Group_excl()`
  - `MPI_Group_range_incl(), MPI_Group_range_excl()`
  - `MPI_Group_union(), MPI_Group_intersect()`
  - `MPI_Group_compare(), MPI_Group_translate_ranks()`

- Subdivide a communicator
  - `MPI_Comm_create(comm, group, newcomm)`

Creating Cartesian Topologies

- Creates cartesian topologies

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
                    int *dims, int *periods, int reorder,
                    MPI_Comm *comm_cart)
```

  - Creates a new communicator with dims dimensions.
    - ndims = number of dimensions
    - dims = vector of length of each dimension
    - periods = vector indicates which dims are periodic (wrap-around link)
    - reorder = flag – ranking may be reordered

P0, P1, P2, P3

MPI_Cart_create(comm, 2, {2,2}, {0,0}, ..., new_comm)

P0	P1
P2	P3

Using Cartesian Topologies

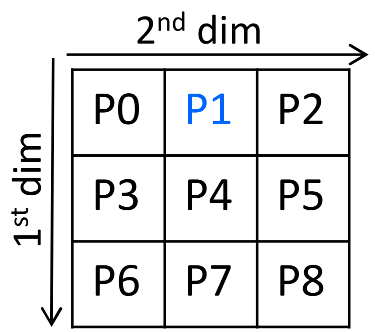
- Sending and receiving messages still require ranks (or process IDs)

- Convert ranks to cartesian coordinates and vice-versa

```
int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims,
  int *coords)
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
```

- The most common operation on cartesian topologies is a shift

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir, int s_step,
           int *rank_source, int *rank_dest)
```



Examples:

P1 calls MPI_Cart_coord() → (0, 1)
MPI_Cart_rank({0, 1}) → 1 (P1)
MPI_Cart_shift(0 (direction), 1 (step)) → src:P0, dst:P2
(assuming all dims are periodic)

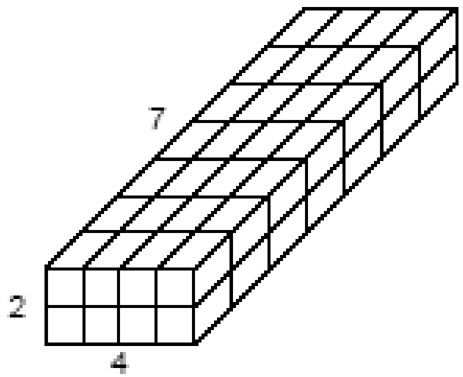
Splitting Cartesian Topologies

- Partition a Cartesian topology to form lower-dimensional grids:

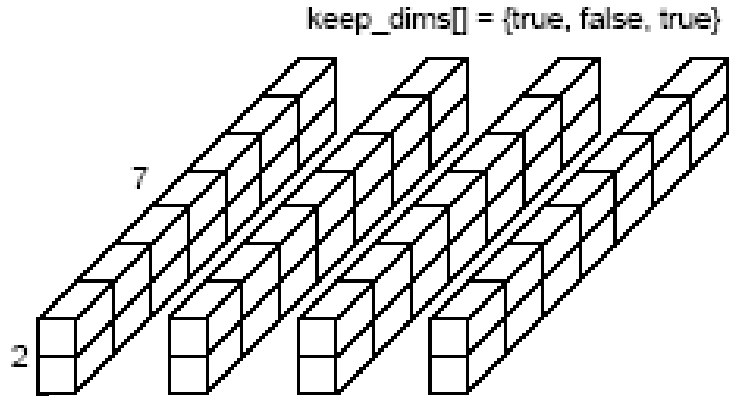
```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
                 MPI_Comm *comm_subcart)
```

  - keep_dims[i] determines whether to split or not the ith dimension
- The coordinate of a process in a sub-topology
  - Derived from its coordinate in the original topology
  - Disregarding the coordinates that correspond to the dimensions that were not retained
  - Example: (2, 3) → (2) if dims = {true, false}

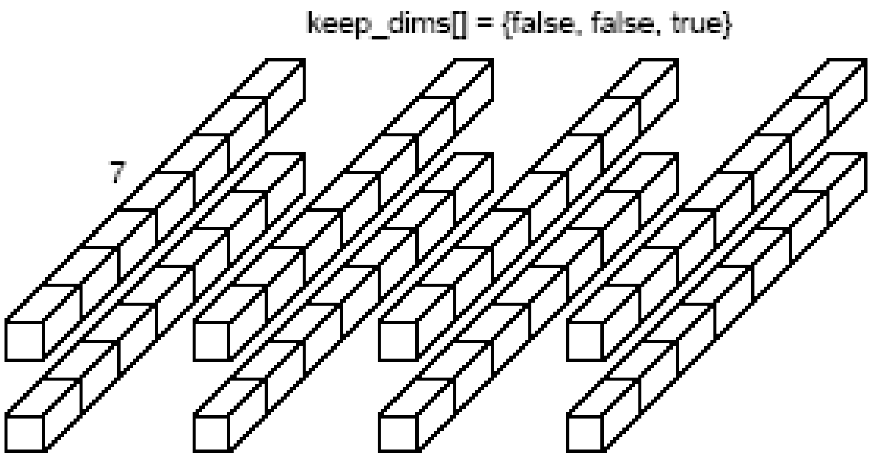
Splitting Cartesian Topologies



2 x 4 x 7



four 2 x 1 x 7



eight 1 x 1 x 7

Splitting Cartesian Topologies: Example

```
int nrow, mcol, i, lastrow, p, root;
int Iam, id2D, colID, ndim;
int coords1D[2], coords2D[2], dims[2], aij[1], alocal[3];
int belongs[2], periods[2], reorder;
MPI_Comm comm2D, commcol;
/* Starts MPI processes ... */
MPI_Init(&argc, &argv);                  /* starts MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &Iam);     /* get current process id */
MPI_Comm_size(MPI_COMM_WORLD, &p);       /* get number of processes */

nrow = 3; mcol = 2; ndim = 2;
root = 0; periods[0] = 1; periods[1] = 0; reorder = 1;

/* create cartesian topology for processes */
dims[0] = nrow;         /* number of rows */
dims[1] = mcol;         /* number of columns */
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, periods, reorder, &comm2D);
MPI_Comm_rank(comm2D, &id2D);
MPI_Cart_coords(comm2D, id2D, ndim, coords2D);

/* Create 1D column subgrids */
belongs[0] = 1;         /* this dimension belongs to subgrid */
belongs[1] = 0;
MPI_Cart_sub(comm2D, belongs, &commcol);
MPI_Comm_rank(commcol, &colID);
MPI_Cart_coords(commcol, colID, 1, coords1D);
```

Limitations of MPI Data Types

- Only primitive data types can be exchanged through MPI_Send/Recv

- Many programs use more complex data structures
  - Ex. struct in C

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Basic data types in MPI

- To make more complex data types to be exchanged through MPI communication methods
  - MPI should know the size of a data structure
  - MPI should know the members within the data structure
    - Location
    - Size of each member

```
struct a {
    MPI_DOUBLE x[2];
    MPI_DOUBLE y[2];
    MPI_LONG value[2];
};
```

Member	Offset in bytes
x	0
y	16
value	32

MPI_Type create_struct

- Builds a derived datatype consisting of individual elements

```
int MPI_Type_create_struct(
      int            count                      /* in  */,
      int            array_of_blocklengths[]    /* in  */,
      MPI_Aint       array_of_displacements[]   /* in  */,
      MPI_Datatype   array_of_types[]           /* in  */,
      MPI_Datatype*  new_type_p                 /* out */);
```

- array_of_block_lengths
  - Each member can be either a variable or an array
  - Ex. {2, 2, 2};
- array_of_displacements
  - Offsets of each member from start address
  - Ex. {0, 16, 32}
- array_of_types
  - Types of each member
  - Ex. {MPI_DOUBLE, MPI_DOUBLE, MPI_LONG}

```
struct a {
    MPI_DOUBLE x[2];
    MPI_DOUBLE y[2];
    MPI_LONG value[2];
};
```

MPI_Type create_struct

```
int main(int argc, char *argv[])
{
    struct Partstruct particle[1000];
    int i, j, myrank;
    MPI_Status status;
    MPI_Datatype Particletype;
    MPI_Datatype type[3] = { MPI_CHAR, MPI_DOUBLE, MPI_CHAR };
    int blocklen[3] = { 1, 6, 7 };
    MPI_Aint disp[3];

    MPI_Init(&argc, &argv);

    disp[0] = &particle[0].c - &particle[0];
    disp[1] = &particle[0].d - &particle[0];
    disp[2] = &particle[0].b - &particle[0];
    MPI_Type_create_struct(3, blocklen, disp, type, &Particletype);
    MPI_Type_commit(&Particletype);

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0)
        MPI_Send(particle, 1000, Particletype, 1, 123, MPI_COMM_WORLD);
    else if (myrank == 1)
        MPI_Recv(particle, 1000, Particletype, 0, 123, MPI_COMM_WORLD, &status);
    MPI_Finalize();
}
```

```
struct Partstruct
{
    char c;
    double d[6];
    char b[7];
};
```

MPI_Get_address

- To know the address of the memory location referenced by `location_p`
- The address is stored in an integer variable of type `MPI_Aint`

```
int MPI_Get_address(
        void*        location_p   /* in  */,
        MPI_Aint*    address_p    /* out */);
```

```
struct a {
    MPI_DOUBLE x[2];
    MPI_DOUBLE y[2];
    MPI_LONG value[2];
};
```

```
struct a a;
MPI_Get_address(&a.x, &x_addr);
MPI_Get_address(&a.y, &y_addr);
MPI_Get_address(&a.value, &value_addr);
array_of_displacements[0] = x_addr - &a;
array_of_displacements[1] = y_addr - &a;
array_of_dispalcements[2] = value_addr - &a;
```

Other methods

- MPI_Type_commit
  - To let MPI know the new data type
  - After calling this function, the new data type can be used in MPI communication methods

```
int MPI_Type_commit(MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

- MPI_Type_free
  - When the new data type is no longer used, this function frees any additional storages used for the new data type

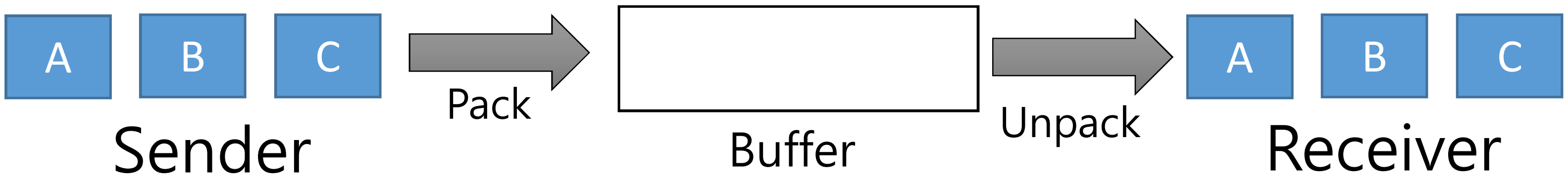
```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

MPI_Pack/Unpack

- An alternative method to send/receive a complex data structure
- Pack multiple data types into a single buffer
- One pair of MPI_Send & MPI_Recv
- Sender and receiver have to know which data types are packed in the single buffer

```
buffer = malloc()
MPI_Pack(A)
MPI_Pack(B)
MPI_Pack(C)
MPI_Send(buffer, MPI_PACKED)
```

```
buffer = malloc()
MPI_Recv(buffer, MPI_PACKED)
MPI_Unpack(A)
MPI_Unpack(B)
MPI_Unpack(C)
```



Concluding Remarks

- MPI or the Message-Passing Interface
  - An interface of parallel programming in distributed memory system
  - Supports C, C++, and Fortran
  - Many MPI implementations
    - Ex, OpenMPI, MPICH2, Intel MPI
- SPMD program
- Message passing
  - Communicator
  - Point-to-point communication
  - Collective communication
  - Safe use of communication is important
    - Ex. MPI_Sendrecv()