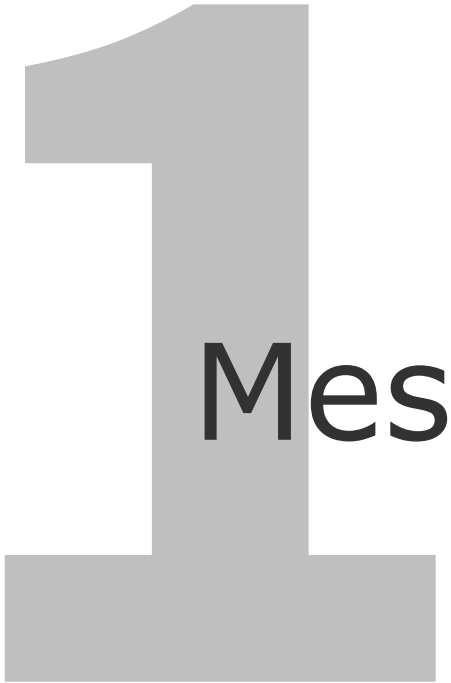
Parallel Programming and Heterogeneous Computing

D2 - Shared-Nothing: MPI

Max Plauth, *Sven Köhler*, Felix Eberhardt, Lukas Wenzel, and Andreas Polze

Operating Systems and Middleware Group

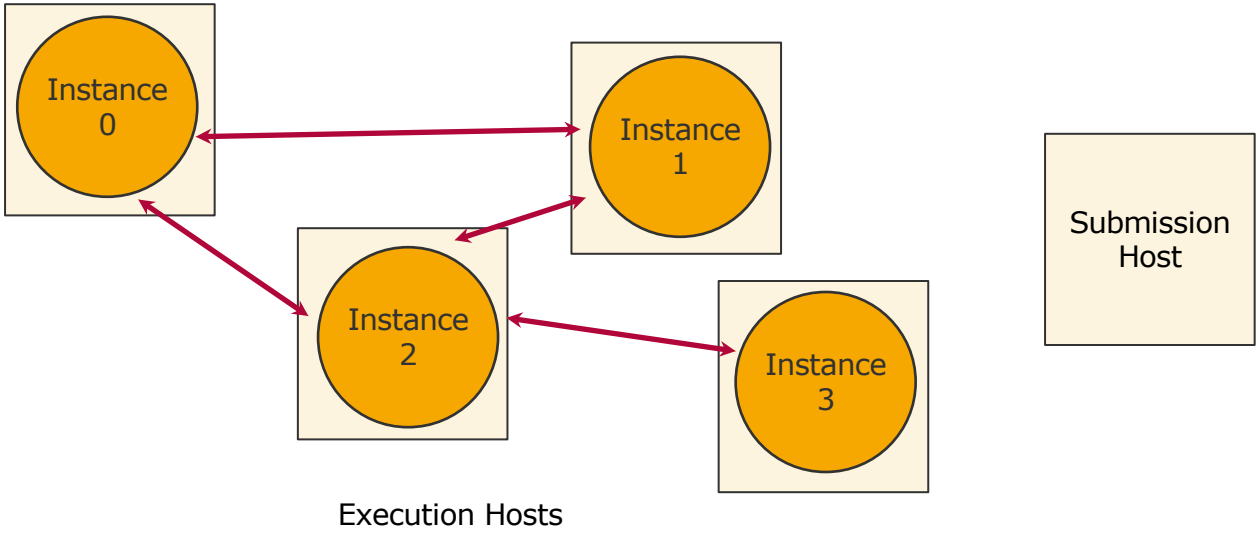
Hasso Plattner Institut
Digital Engineering · Universität Potsdam



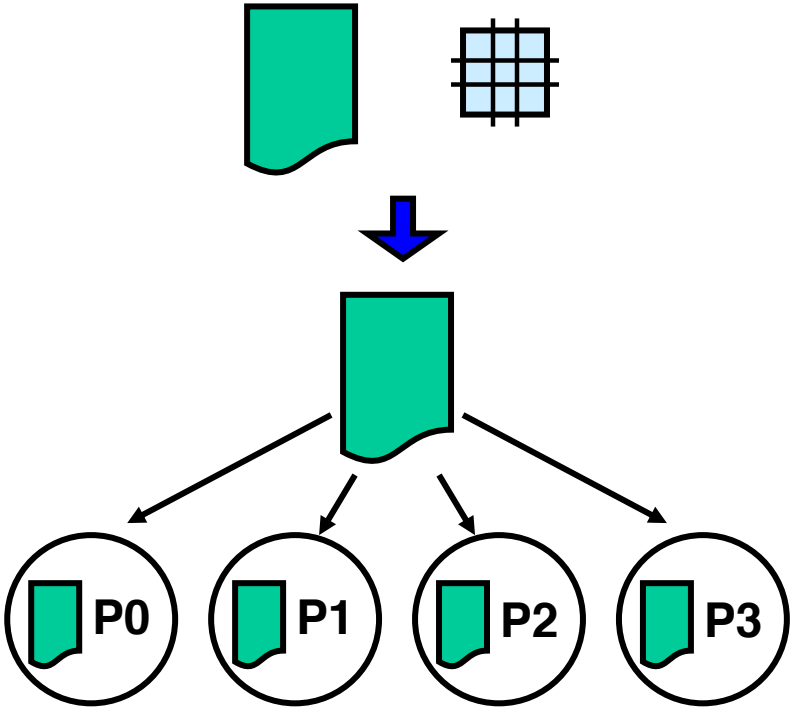
1 Message Passing

Message Passing

- Programming paradigm targeting shared-nothing infrastructures
  - Implementations for shared memory available, but typically not the best-possible approach
- Multiple instances of the same application on a set of nodes (SPMD)



Single Program Multiple Data (SPMD)



seq. program and data distribution

seq. node program with message passing

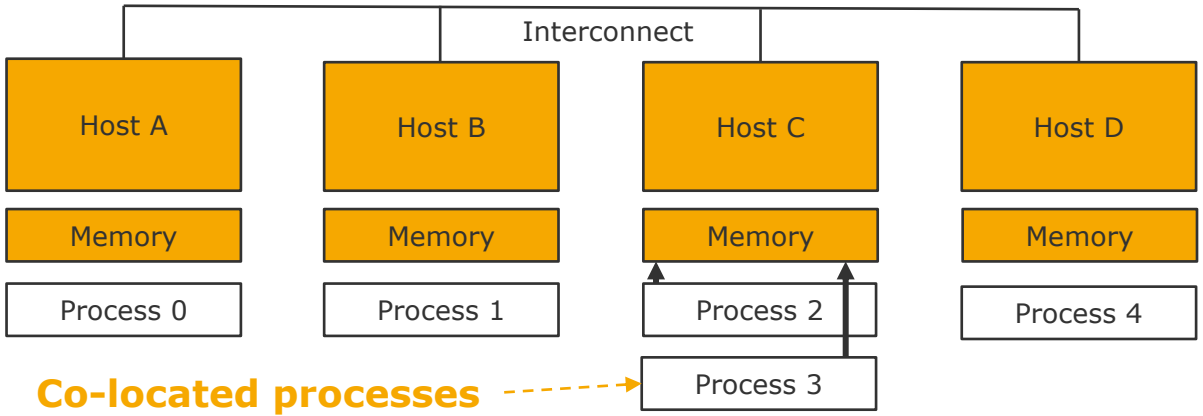
identical copies with different process identifications

Search For Standardization

Situation in the late 1980s to 1990s:
Multiple competing standards and libraries (NX, Express, PARMACS, P4, PVM)

Design goals for new standard:

- Abstraction of interconnect technology
- Enabling vendor-specific optimization while keeping API stable
- Nice to have: Shared-memory execution, with local data exchange (IPC)



Message Passing Interface (MPI)

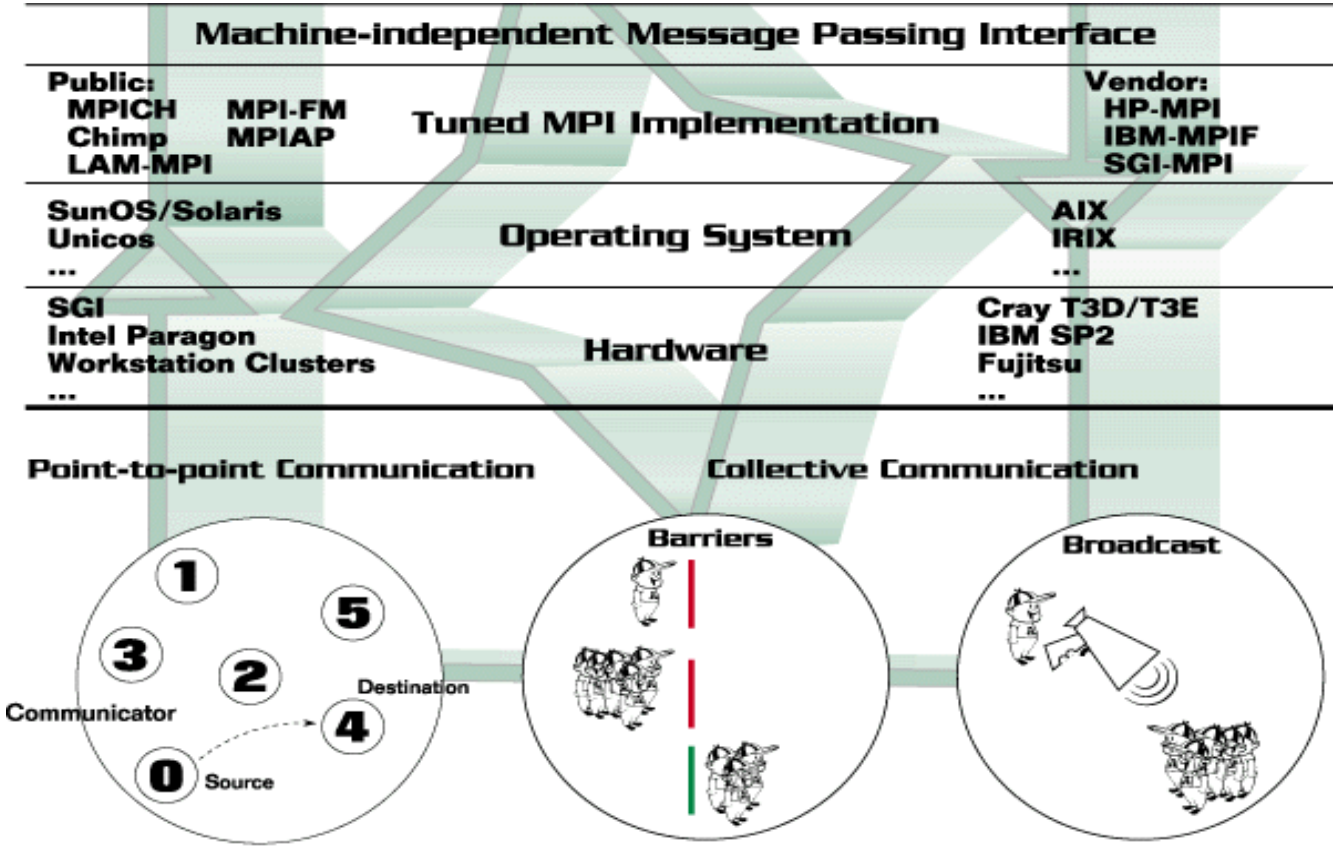
Message Passing Interface (MPI)

Message Passing Interface: standardized API for communication libraries

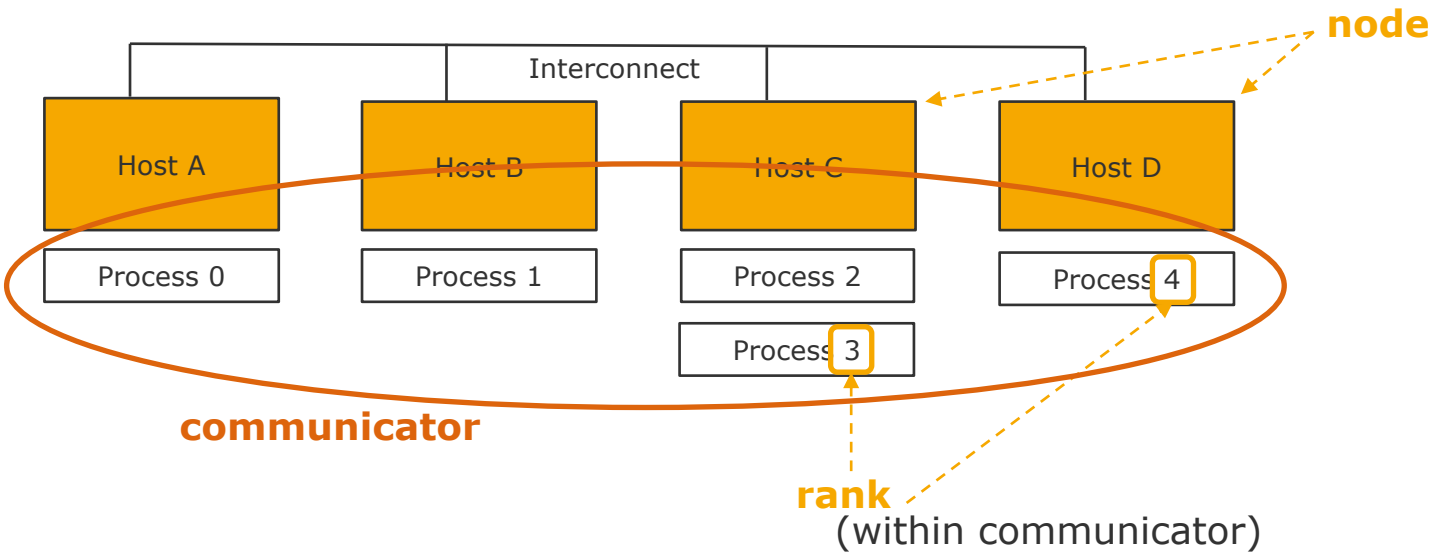
- Developed by the MPI Forum for Fortran and C (other bindings exist)
- Defines Syntax and semantics for source code portability.
- Point-to-point and collective communication
- The same executable runs in several instances (SPMD)
- Ensure implementation freedom on messaging hardware (shared memory, IP, Myrinet, proprietary …)
- Focus on efficiency of communication and memory usage by specific vendor implementations (non-interoperable):
  - MPICH (Argonne National Laboratory)
  - OpenMPI (Different universities and industry)
  - …

Revisions: MPI 1.0 (1994), 2.0 (1997), 3.0 (2012), 3.1 (2015)

MPI Concepts



MPI Communication Terminology



Communicator: handle for group of processes (MPI_COMM_WORLD = all)

Size: Number of processes in a communicator

Hello World

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int size, rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello from rank = %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Get **your own** rank

mandatory

Hello World: Compile and Execute (Single Machine)

```
$ mpicc hello.c -o hello
```

Compile with drop-in CC replacement

```
$ ./hello
Hello from rank = 0 of 1
```

Local execution (serial program)

```
$ mpirun -np 4 ./hello
Hello from rank = 2 of 4
Hello from rank = 3 of 4
Hello from rank = 0 of 4
Hello from rank = 1 of 4
```

Local execution (4 parallel processes)

It is the same executable!
Beware of side effects (fopen, sockets, …)!

Organization

```
static const int SERVER_RANK = 0;

/* Get rank */

if (rank == SERVER_RANK) {
        /* read, distribute, and gather data */
} else {
        /* compute data on node and send back result */
}
```

Often process rank 0 coordinates the other processes and handles I/O.

MPI Data Types

You can send/receive buffers (arrays) of:

C	
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	
MPI_UNSIGNED_INT	
...	
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

FORTRAN	
MPI_INTEGER	integer
MPI_REAL	real
MPI_DOUBLE_PRECISION	double precision
MPI_COMPLEX	complex
MPI_LOGICAL	logical
MPI_CHARACTER	character(1)
MPI_BYTE	
MPI_PACKED	

MPI Point-To-Point Communication

Point-to-point communication between ranks:

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
        int destPid, int tag, MPI_Comm comm);

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
        int srcPid, int tag, MPI_Comm comm,
        MPI_Status *status);
```

Send and receive functions need a matching partner

- Source / destination identified by tuple *(tag, rank, communicator)*
- Constants: `MPI_ANY_TAG`, `MPI_ANY_SOURCE`, `MPI_ANY_DEST`

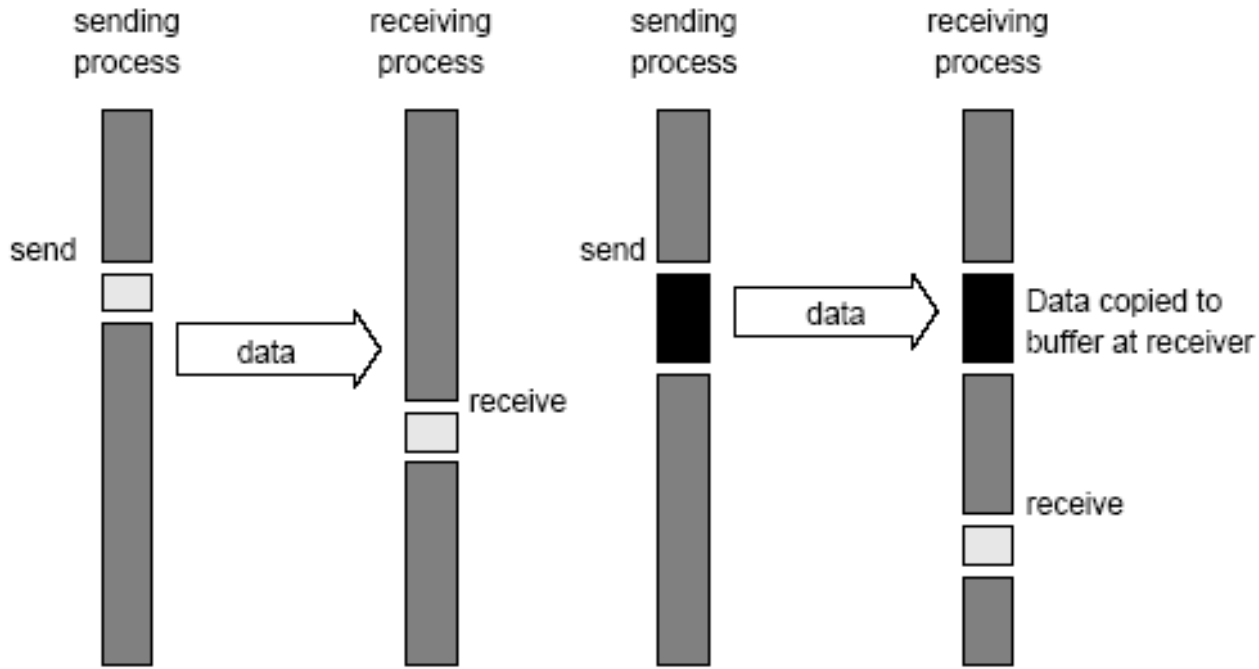
Blocking communication

- Synchronous / blocking communication
  - *„Do not return until the message data and envelope have been stored away"*
  - Send and receive operations run synchronously
  - Buffering may or may not happen
  - Sender and receiver application-side buffers are in a defined state afterwards
- Default mode: **`MPI_Send`**
  - Blocks until the message is received by the target process
  - MPI decides whether outgoing messages are buffered
  - Call will not return until you can re-use the send buffer

Blocking communication

- Buffered mode: **`MPI_Bsend`**
  - User provides self-created buffer (**`MPI_Buffer_attach`**)
  - Returns even if no matching receive is currently available
  - Send buffer not promised to be immediately re-usable
- Synchronous mode: **`MPI_Ssend`**
  - Returns if the receiver started to receive
  - Send buffer not promised to be immediately re-usable
  - Recommendation for most cases, can (!) avoid buffering at all
- Ready mode: **`MPI_Rsend`**
  - Sender application takes care of calling `MPI_Rsend` only if the matching `MPI_Recv` is promised to be available
  - Beside that, same semantics as `MPI_Ssend`
  - Without receiver match, outcome is undefined
  - Can omit a handshake-operation on some systems

Blocking Buffered Send



Non-Overtaking Message Order

"If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending."

```
if (rank == SERVER_RANK) {
    MPI_Bsend(buf1, count, MPI_REAL, 1, tag, comm);
    MPI_Bsend(buf2, count, MPI_REAL, 1, tag, comm);
} else if (rank == 1) {
    MPI_Recv(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, &status);
    MPI_Recv(buf2, count, MPI_REAL, 0,         tag, comm, &status);
}
```

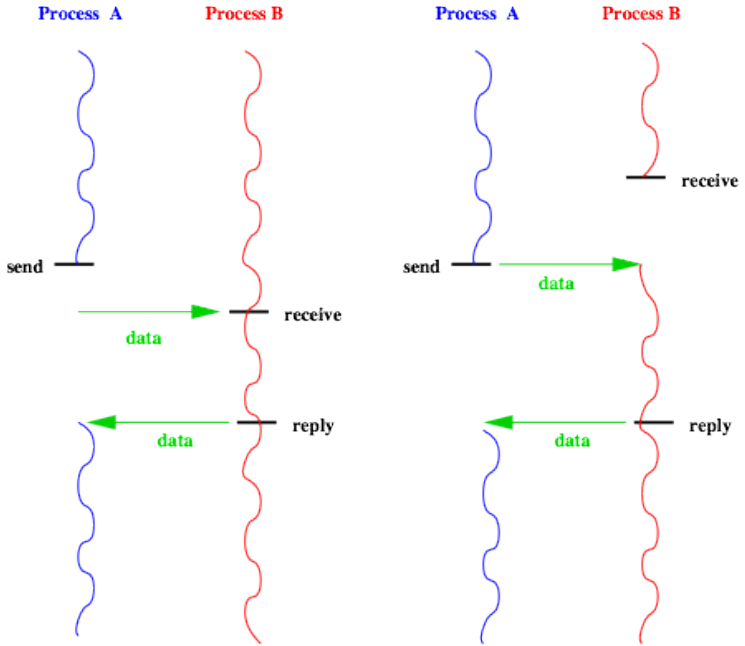
Distributed Deadlock Example

Your responsibility to ensure correct receive order, otherwise deadlocks can occur with circular waits in blocking send/receives:

```
if (rank == SERVER_RANK) {
    MPI_Send(buf1, count, MPI_REAL, 1, 1, comm);
    MPI_Send(buf2, count, MPI_REAL, 1, 2, comm);
} else if (rank == 1) {
    MPI_Recv(buf2, count, MPI_REAL, 0, 2, comm, &status);
    MPI_Recv(buf1, count, MPI_REAL, 0, 1, comm, &status);
}
```

Rendezvous

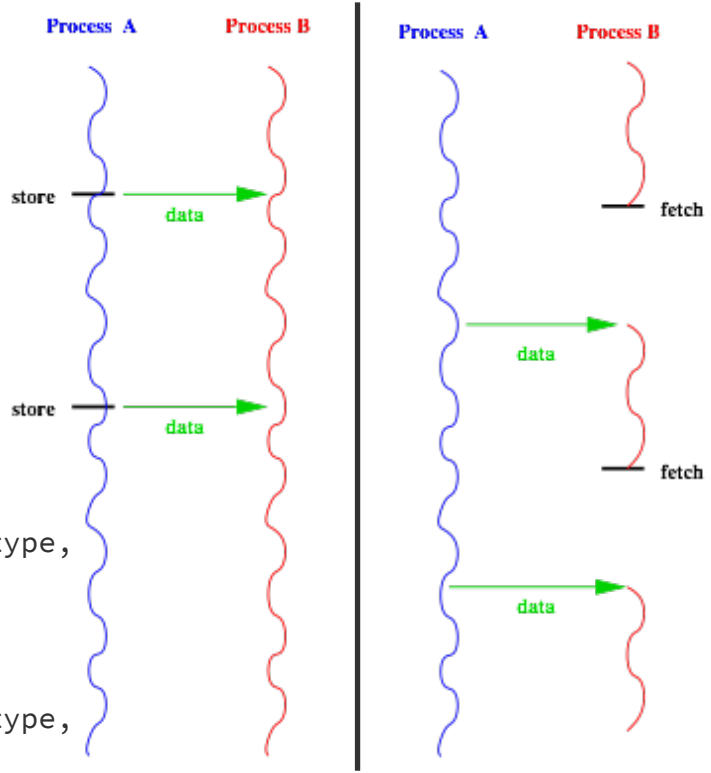
- Special case with rendezvous communication
- Sender retrieves reply message for it's request
- Control flow on sender side only continues after this reply message
- Typical RPC problem
- Ordering problem should be solved by the library




```
int MPI_Sendrecv(
        void* sbuf, int scount, MPI_Datatype stype, int dest, int stag,
        void* rbuf, int rcount, MPI_Datatype rtype, int src, int rtag,
        MPI_Comm com, MPI_Status* status);
```

One-Sided Communication

- No explicit receive operation, but synchronous remote memory access
- Requires explicit “Window object”
- Subsequent synchronization on window object needed to ensure operation is complete




```
int MPI_Put(
  void *src, int srccount, MPI_Datatype srctype,
  int dest, void *destoffset, int destcount,
  MPI_Datatype desttype, MPI_Win win);

int MPI_Get(
  void *dst, int dstcount, MPI_Datatype dsttype,
  int src, void *srcoffset, int srccount,
  MPI_Datatype srctype, MPI_Win win);
```

Hasso Plattner Institut

3

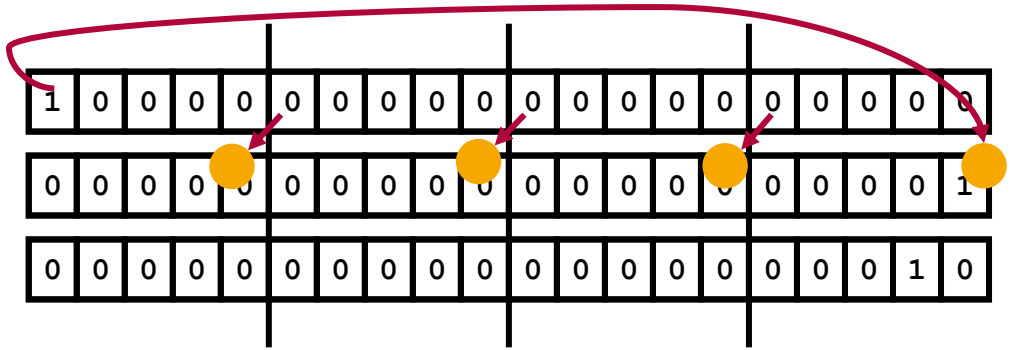
Distributed Deadlocks & Non-Blocking Communication

Circular Left Shift Example

shifts <number of positions>

Description

- Position 0 of an array with 100 entries is initialized to 1. The array is distributed among all processes in a blockwise fashion.
- A number of circular left shift operations is executed.
- The number is specified via a command line parameter.



Circular Left Shift Example

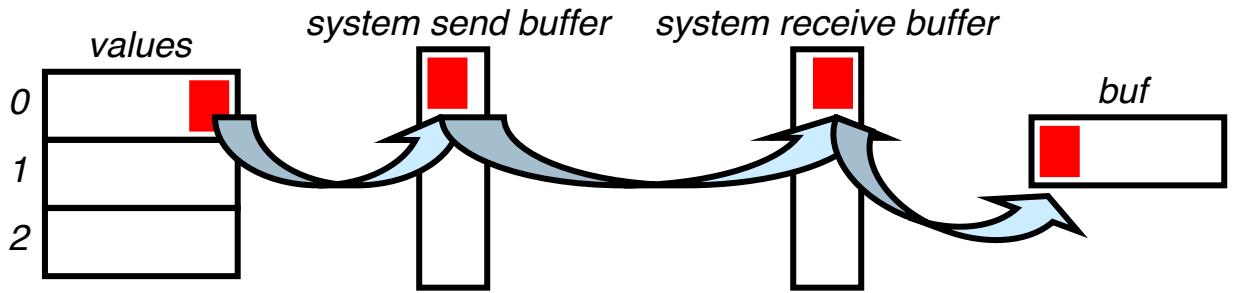
```
#include "mpi.h"

main (int argc,char *argv[]){
int myid, np, ierr, lnbr, rnbr, shifts, i, j;
int *values;
MPI_Status status;

ierr = MPI_Init (&argc, &argv);
if (ierr != MPI_SUCCESS){
  ...
}

MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

Circular Left Shift Example

```
if (myid==0){
  lnbr=np-1; rnbr=myid+1;
}
else if (myid==np-1){
  lnbr=myid-1; rnbr=0;
}
else{
  lnbr=myid-1; rnbr=myid+1;
}

if (myid==0) shifts=atoi(argv[1]);
MPI_Bcast (&shifts, 1, MPI_INT, 0, MPI_COMM_WORLD);

values= (int *) calloc(100/np,sizeof(int));
if (myid==0){
  values[0]=1;
}
```

Circular Left Shift Example

```
for (i=0;i<shifts;i++){
  int buf;

  MPI_Send(&values[0],1,MPI_INT,lnbr,10,MPI_COMM_WORLD);
  MPI_Recv(&buf, 1, MPI_INT,rnbr,10,
                                  MPI_COMM_WORLD, &status);
  for (j=1;j<100/np;j++){
    values[j-1]=values[j];
  }
  values[100/np-1]=buf;
}
```



Circular Left Shift Example

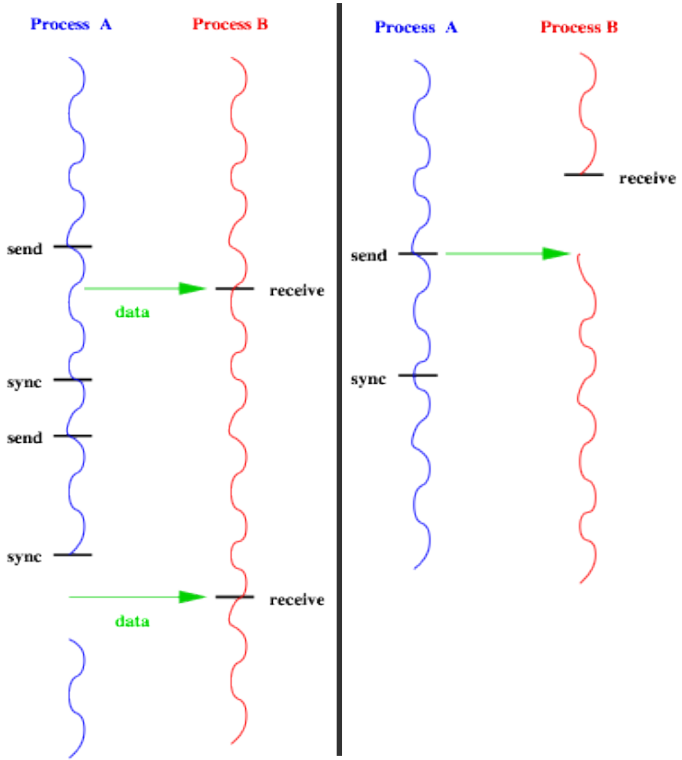
```
for (i=0;i<shifts;i++){
  if (myid==0){
    MPI_Send(&values[0], 1, MPI_INT, lnbr, 10,
             MPI_COMM_WORLD);
    for (j=1;j<100/np;j++){
      values[j-1]=values[j];
    }
    MPI_Recv(&values[100/np-1], 1, MPI_INT, rnbr,
             10, MPI_COMM_WORLD, &status);
  }else{
    int buf=values[0];
    for (j=1;j<100/np;j++){
      values[j-1]=values[j];
    }
    MPI_Recv(&values[100/np-1], 1, MPI_INT, rnbr,
             10, MPI_COMM_WORLD, &status);
    MPI_Send(&buf, 1, MPI_INT, lnbr, 10,
             MPI_COMM_WORLD);
  }
}
```

Process 0

Other Processes

Non-Blocking Communication

- Control flows of sender and receiver are decoupled
- Typical approach:
  Blocking receiver with non-blocking sender
  - Implicit buffering on sender side
  - Demands consideration of additional resource consumption:
    application responsibility vs. communication library responsibility

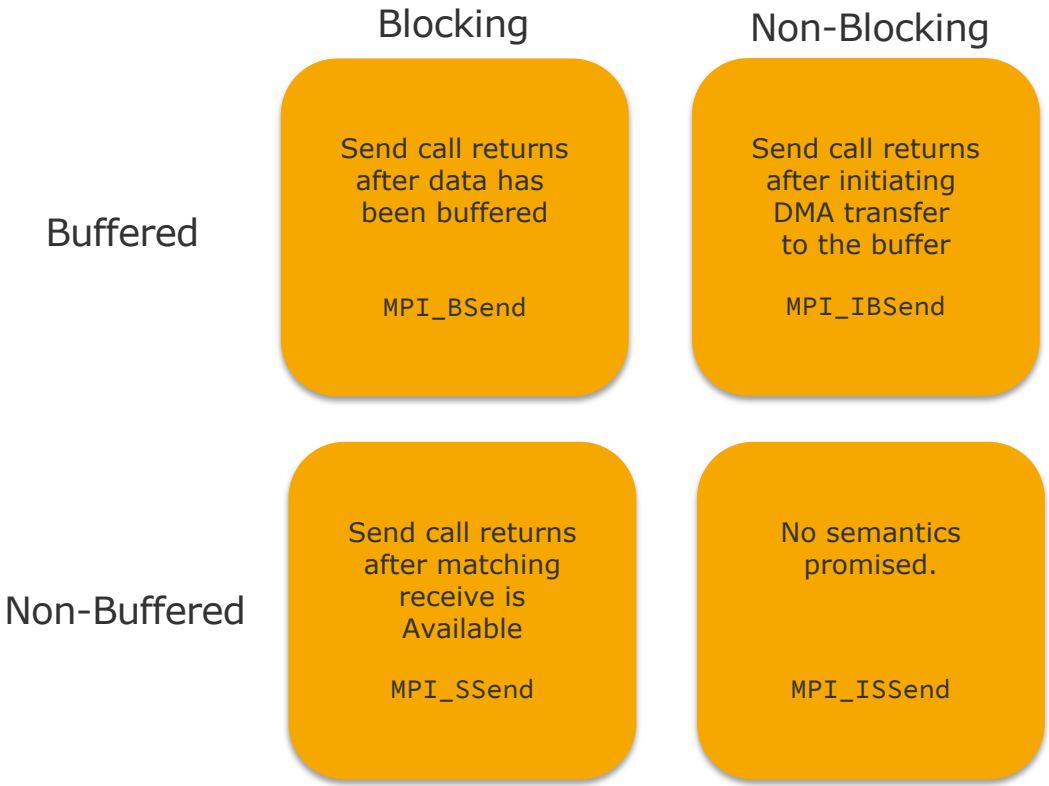


Non-Blocking Communication: Without Buffering

```
int MPI_Issend(void* buf, int count, MPI_Datatype type,
               int dest, int tag, MPI_Comm com,
               MPI_Request* handle);

int MPI_Wait(MPI_Request* handle, MPI_Status* status);
```

- Completion call returns if matching receive has started
- Most efficient non-blocking send method
  - No buffering of data in the communication layer needed
  - Application has to responsibility of not touching the send buffer until the operation is finalized
  - High potential for unintended data corruption
  - Buffering problem is relayed to the application layer

Send and Receive Protocols



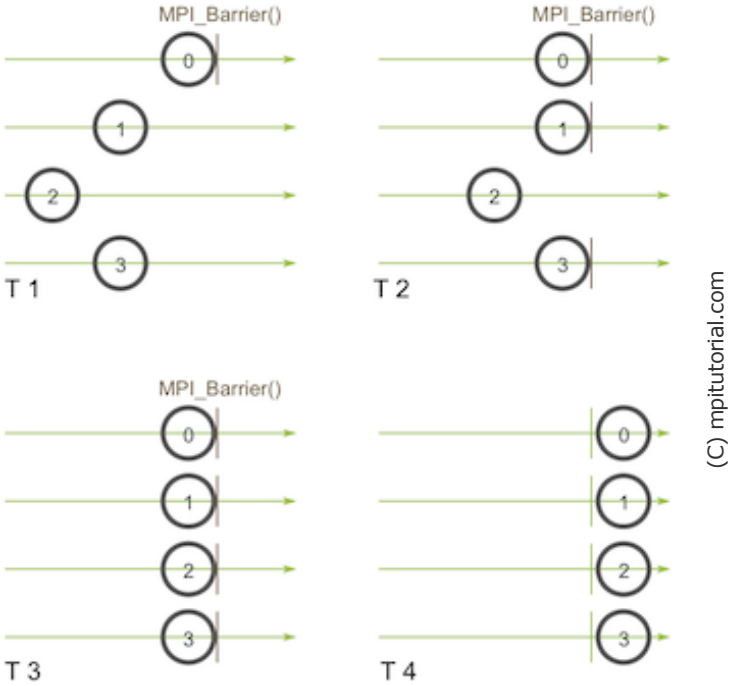
Collective Communication

Collective Communication

- Use case: Synchronization, communication, reduction
- All communication of processes belonging to a group
  - One sender with multiple receivers (‚one-to-all‘)
  - Multiple senders with one receiver (‚all-to-one‘)
  - Multiple senders and multiple receivers (‚all-to-all‘)
- Typical pattern in high-performance computing
- Also nice for data-parallel applications on SIMD hardware
- Participants continue their execution if their send / receive communication with the group is finished
  - Always blocking operation
  - Must be executed by all processes in the group
  - No assumptions on the state of other participants on return

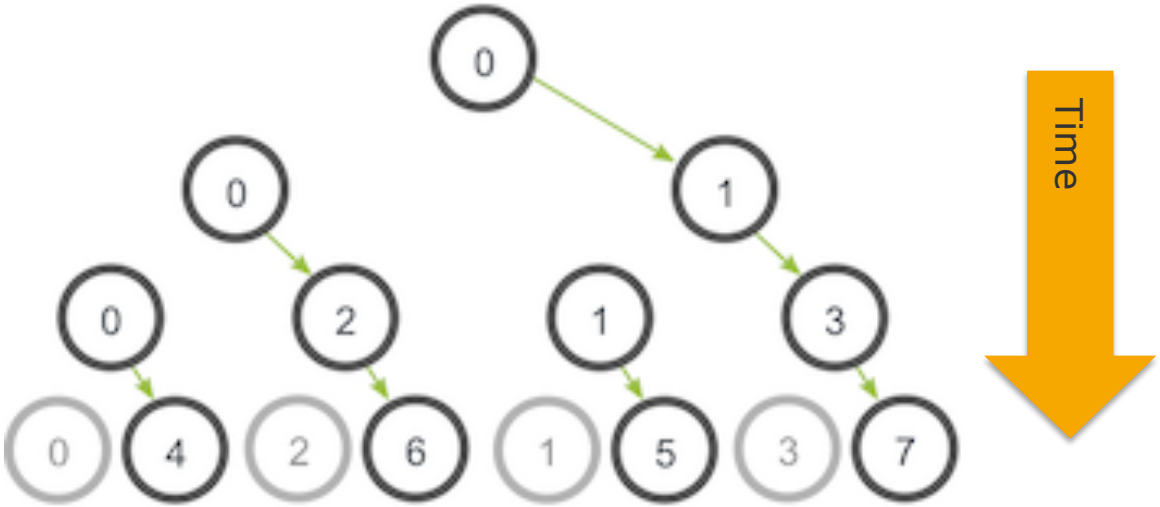
Barrier

Processes in communicator are blocked until everybody reached the barrier




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

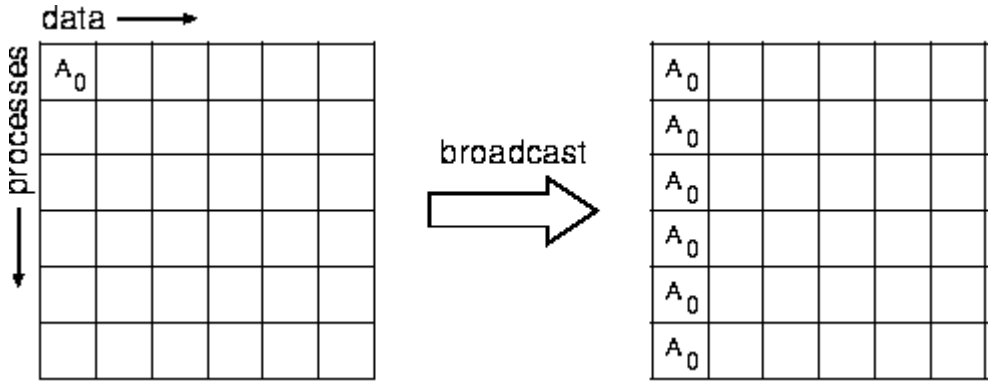
Efficient Barrier Implementation



Broadcast

```
int MPI_Bcast(void *buffer, int count,
        MPI_Datatype datatype, int root, MPI_Comm comm);
```

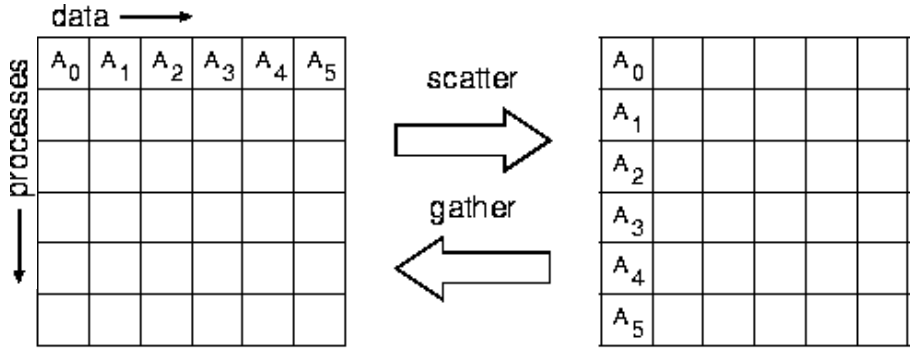
- Root process broadcasts to all group members, itself included
- All group members use the same communicator and the same root as parameter
- On return, all processes have a copy of root's send buffer



Gather

```
int MPI_Gather(
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm)
```

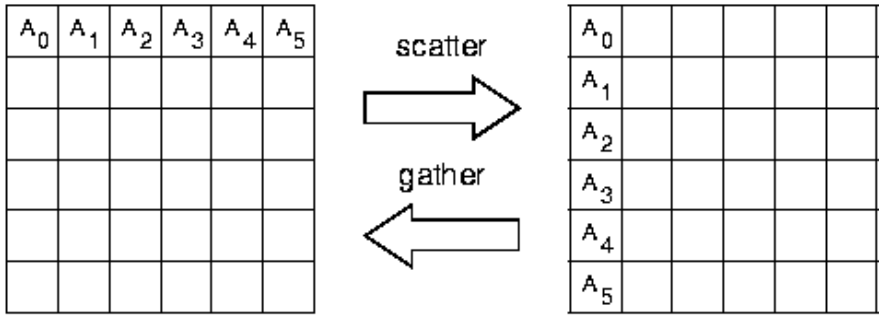
- Each process sends its buffer to the root process, including root
- Incoming messages are stored in rank order
- Receive buffer is ignored for all non-root processes
- Returns if the buffer is re-usable (no finishing promised)



Scatter

```
int MPI_Scatter(
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm)
```

- Sliced buffer of root process is send to all other processes (including the root process itself)
- Send buffer is ignored for all non-root processes
- Returns if data buffer is re-usable, not necessarily finished
- MPI_SCATTERV allows varying count of data to be send to each process

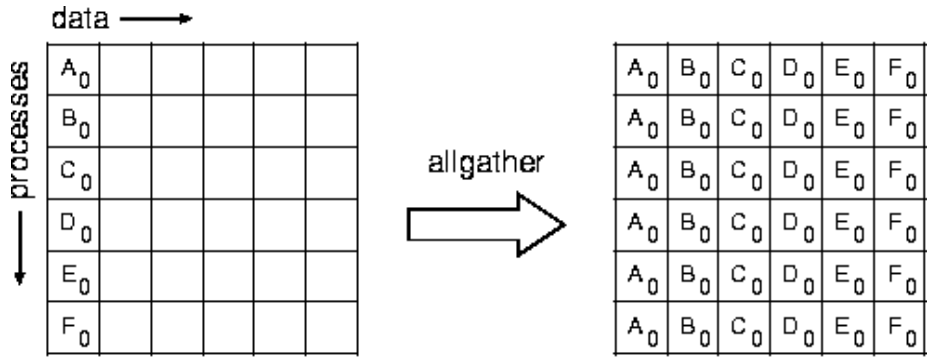


Example: MPI_Scatter + MPI_Reduce

```
/* -- E. van den Berg                                    07/10/2001 -- */
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[]) {
   int data[] = {1, 2, 3, 4, 5, 6, 7}; // Size must be >= #processors
   int rank, i = -1, j = -1;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);

   MPI_Scatter ((void *)data, 1, MPI_INT, (void *)&i  ,
                1, MPI_INT, 0, MPI_COMM_WORLD);
   printf ("[%d] Received i = %d\n", rank, i);

   MPI_Reduce ((void *)&i, (void *)&j, 1, MPI_INT, MPI_PROD,
                0, MPI_COMM_WORLD);

   printf ("[%d] j = %d\n", rank, j);
   MPI_Finalize();
   return 0;
}
```

Allgather

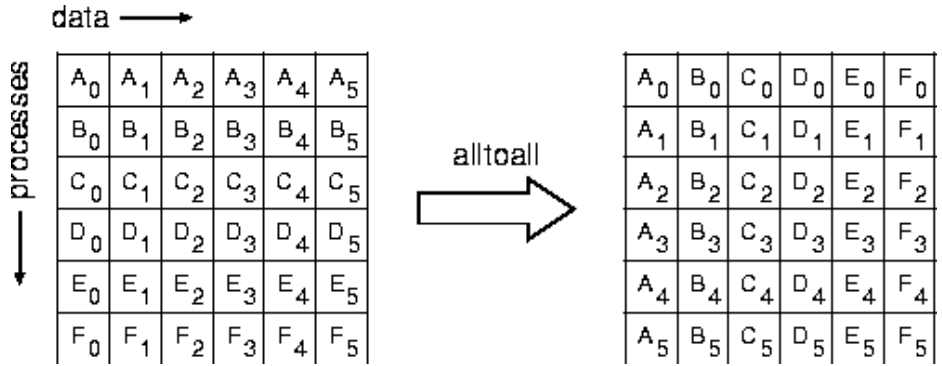
- Distributes the data of all group members to all group members
  - Everbody sends its data together with the own rank
  - Data received is ordered according to the originating rank
- Can be mapped to gather / multicast
  - First collect all data, then distribute everything



Alltoall

```
int MPI_Alltoall(
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
     MPI_Comm comm)
```

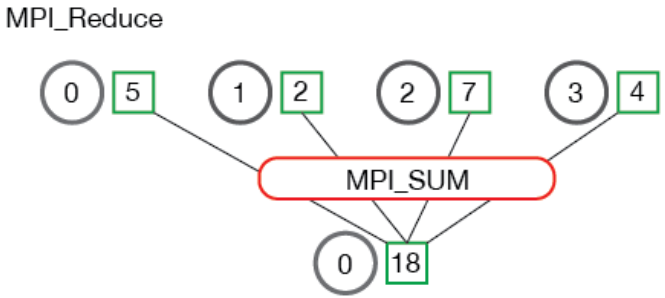
- Global exchange of ‚rows' and ‚colums'
  - All processes execute a logical scatter operation
  - Everybody sends as much as he receives

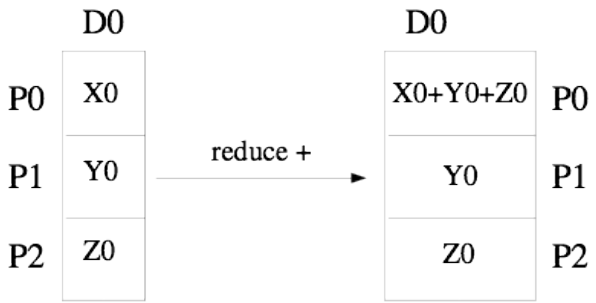


Reduce

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

- Similar to the gather operation, all group members participate with their data, but an **operation** is applied before storing on root rank
- Typical example: Global sum / product
- Mostly only commutative or associative operations
- Reduction can be performed in parallel to the communication





Reduction

Example: Simple parallel sum

```
s=0
for (i=1; i<n; i++)
  s=s+a[i]
```

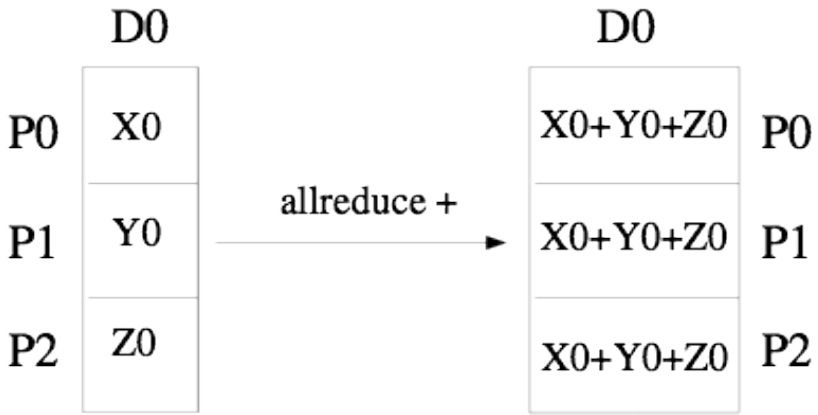
```
s=0
for (i=0; i<local_n; i++){
  s=s+a[i]
}
MPI_Reduce(s, s1, 1,
           MPI_INT, MPI_SUM, P0,
           MPI_COMM_WORLD);
s=s1
```

Predefined Reduction Operators

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-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

MPI_Allreduce

- Everbody sends its data to everybody
  - Reduction result is then available for all participants
  - Can be mapped to reduction and multicast



MPI Process Topologies

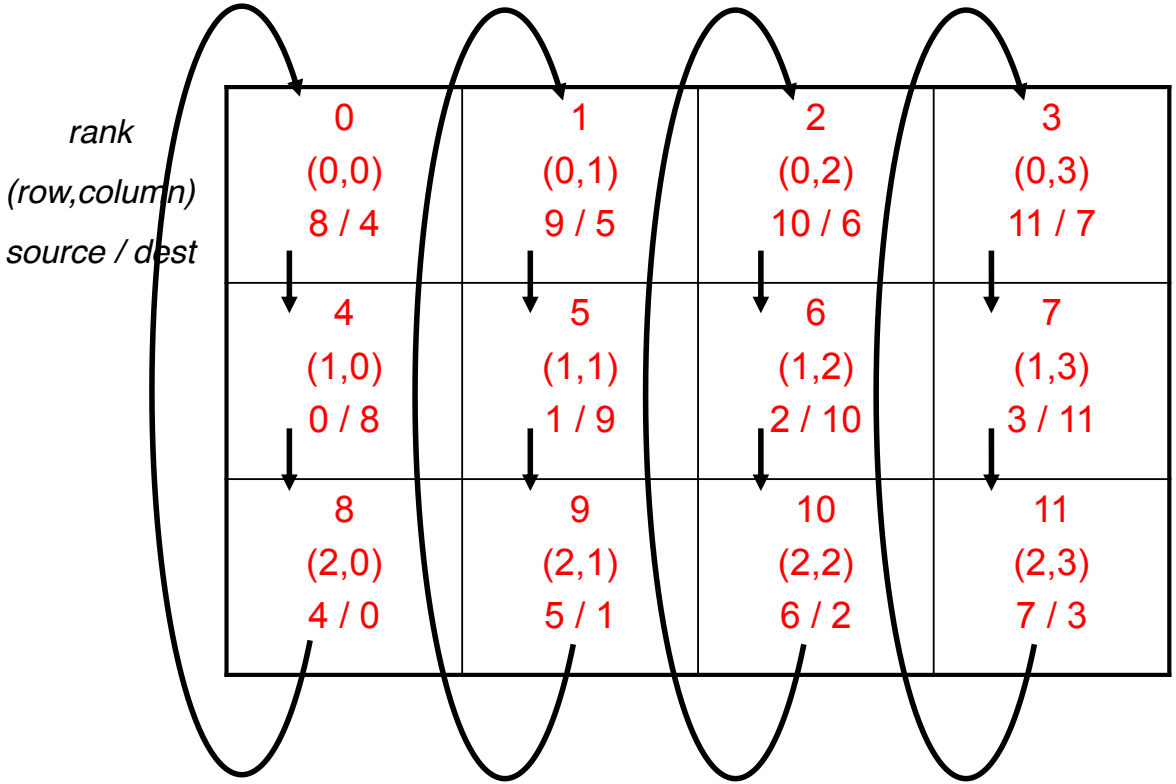
- Topologies help to define a virtual name space structuring
  - Effective mapping of processes to nodes
  - Optimizations for interconnection networks (grids, tori, ...)
- Access through a newly defined communicator
  - *MPI_Cart_create(oldcomm, ndims, dims, periods, reorder, new_comm)*
  - Define structure by
    - number of dimensions (*ndims*)
    - number of processes per dimension (*dims*)
    - periodicity per dimension (*periods*)
- Rank → Coordinates: *MPI_Cart_Coords*
- Coordinates → Rank: *MPI_Cart_Rank*
- Determine target ranks on coordinate shift: *MPI_Cart_Shift*

Example

```
a=rank;
b=-1;

dims[0]=3; dims[1]=4;
periods[0]=true; periods[1]=true;
reorder=false;

MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder
                &comm_2d);
MPI_Cart_coords(comm_2d, rank, 2, &coords);
MPI_Cart_shift(comm_2d, 0, 1, &source, &dest);

MPI_Sendrecv(a, 1, MPI_REAL, dest, 13, b, 1, MPI_REAL,
             source, 13, comm_2d, &status);
```

- Send the own rank number in dimension 0 to the next higher neigbor

MPI Process Topologies




end