Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Shared Nothing Parallelism – MPI

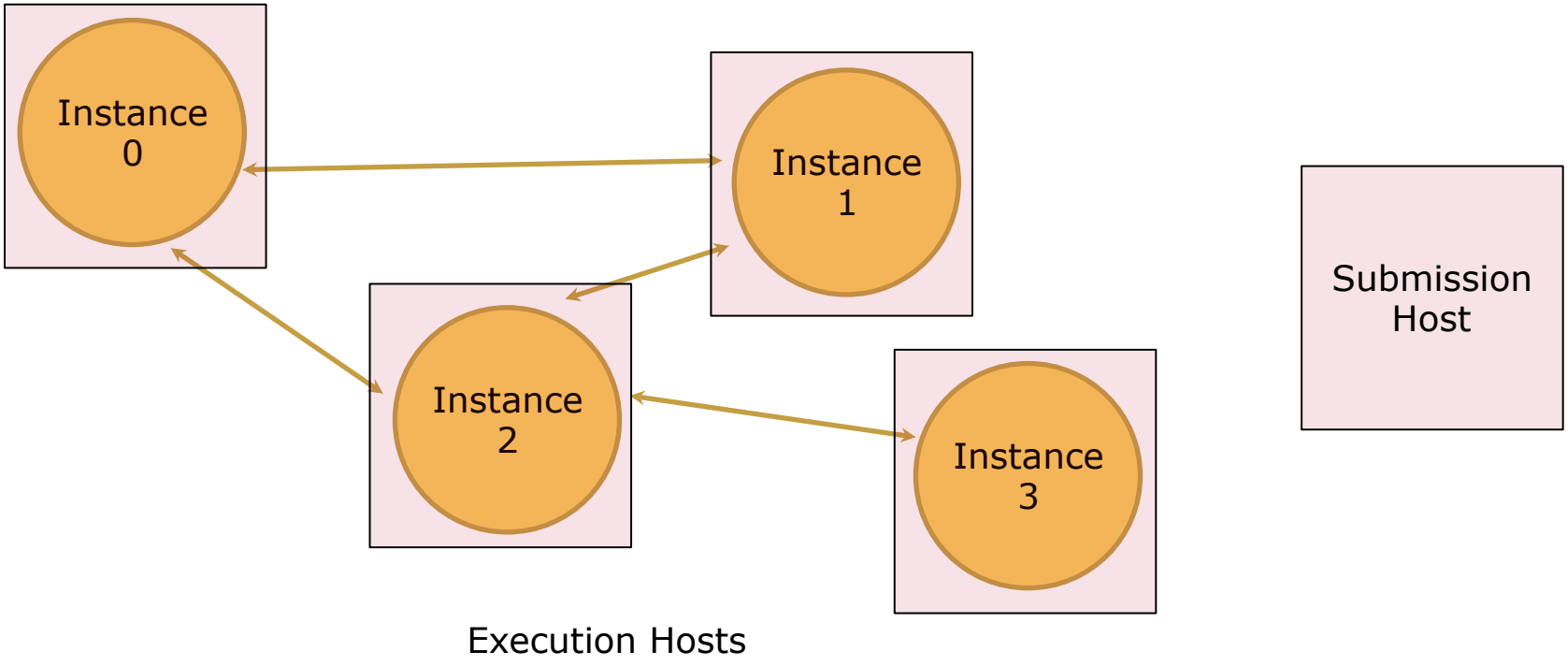
Programmierung Paralleler und Verteilter Systeme (PPV)

Sommer 2015

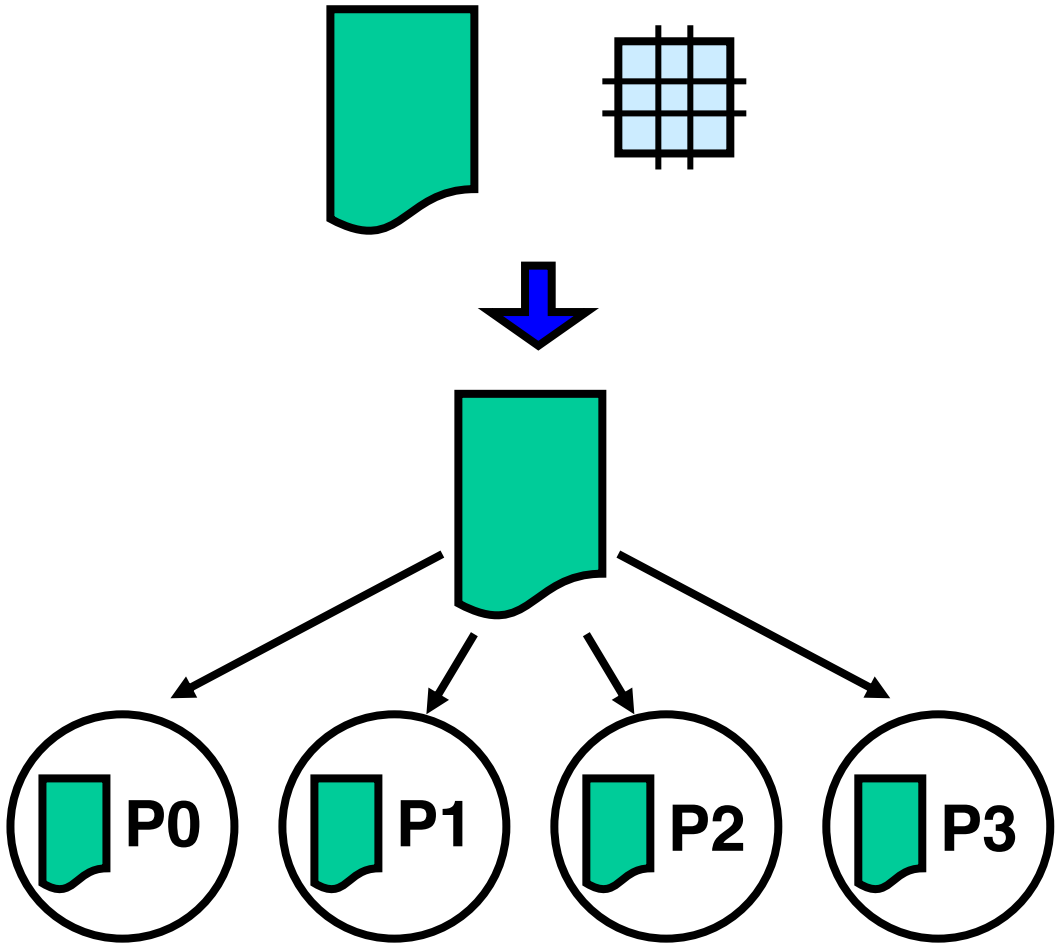
Frank Feinbube, M.Sc., Felix Eberhardt, M.Sc.,
Prof. Dr. Andreas Polze

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

The Parallel Virtual Machine (PVM)

- Developed at Oak Ridge National Laboratory (1989)
- Intended for heterogeneous environments
  - Creation of a parallel multi-computer from cheap components
  - User-configured host pool
- Integrated set of software tools and libraries
- Transparent hardware → Collection of virtual processing elements
- Unit of parallelism in PVM is a **task**
  - Process-to-processor mapping is flexible
- Explicit message-passing mode, multiprocessor support
- C, C++ and Fortran language

PVM (contd.)

- PVM tasks are identified by an integer **task identifier (TID)**
- User named groups of tasks
- Programming paradigm
  - User writes one or more sequential programs
  - Contain embedded calls to the PVM library
  - User typically starts one copy of one task manually
  - This process subsequently starts other PVM tasks
  - Tasks interact through explicit message passing
- Explicit API calls for converting transmitted data into a platform-neutral and typed representation

PVM_SPAWN


```
int numt = pvm_spawn(char *task, char **argv, int flag,
                     char *where, int ntask, int *tids )
```

- Arguments
  - *task*: Executable file name
  - *flag*: Several options for execution
    (usage of *where* parameter, debugging, tracing options)
    - If flag is 0, then *where* is ignored
  - *where*: Execution host name or type
  - *ntask*: Number of instances to be spawned
  - *tids*: Integer array with TIDs of the spawned tasks
- Returns actual number of spawned tasks

PVM Example


```
main() {     /* hello.c */
      int cc, tid, msgtag;
      char buf[100];
      printf("i'm t%x\n", pvm_mytid());  //print id
      cc = pvm_spawn("hello_other",
                     (char**)0, 0, "", 1, &tid);
      if (cc == 1) {
            msgtag = 1;
            pvm_recv(tid, msgtag);    // blocking
            pvm_upkstr(buf);          // read msg content
            printf("from t%x: %s\n", tid, buf);
      } else
            printf("can't start it\n");
      pvm_exit();
}
```

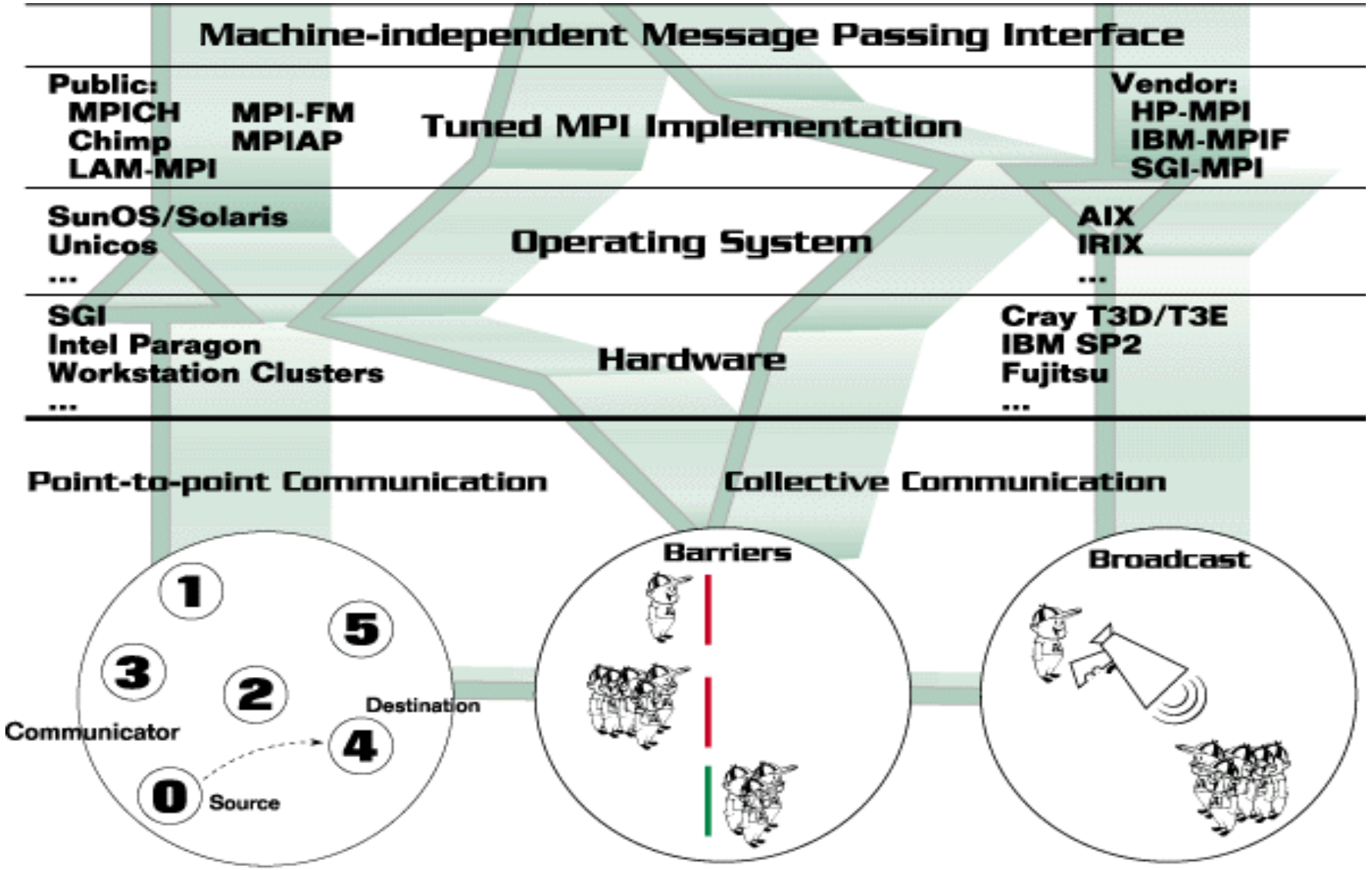
PVM Example (contd.)


```
main() {      /* hello_other.c */
      int ptid, msgtag;
      char buf[100];
      ptid = pvm_parent(); // get master id
      strcpy(buf, "hello from ");
      gethostname(buf+strlen(buf), 64); msgtag = 1;
      // initialize send buffer
      pvm_initsend(PvmDataDefault);
      // place a string
      pvm_pkstr(buf);
      // send with msgtag to ptid
      pvm_send(ptid, msgtag); pvm_exit();
}
```

Message Passing Interface (MPI)

- Large number of different message passing libraries (PVM, NX, Express, PARMACS, P4, …)
- Need for standardized API solution: *Message Passing Interface*
  - Communication library for SPMD programs
  - Definition of syntax and semantics for source code portability
  - Ensure implementation freedom on messaging hardware - shared memory, IP, Myrinet, proprietary …
  - MPI 1.0 (1994), 2.0 (1997), 3.0 (2012) – developed by MPI Forum for Fortran and C
- Fixed number of processes, determined on startup
  - Point-to-point and collective communication
  - Focus on efficiency of communication and memory usage, not interoperability

MPI Concepts



MPI Data Types

Hasso Plattner Institut

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 Communicators

- Each application process instance has a **rank**, starting at zero
- Communicator: Handle for a group of processes with a rank space

  `MPI_COMM_SIZE` `(IN comm, OUT size),`
  `MPI_COMM_RANK` `(IN comm, OUT pid)`
- Default communicator `MPI_COMM_WORLD` per application
- Point-to-point communication between ranks

  `MPI_SEND` `(IN buf, IN count, IN datatype, IN destPid, IN msgTag, IN comm)`
  `MPI_RECV` `(IN buf, IN count, IN datatype, IN srcPid, IN msgTag, IN comm, OUT status)`
  - Send and receive functions need a matching partner
  - Source / destination identified by *[tag, rank, communicator]*
  - Constants: `MPI_ANY_TAG, MPI_ANY_SOURCE, MPI_ANY_DEST`

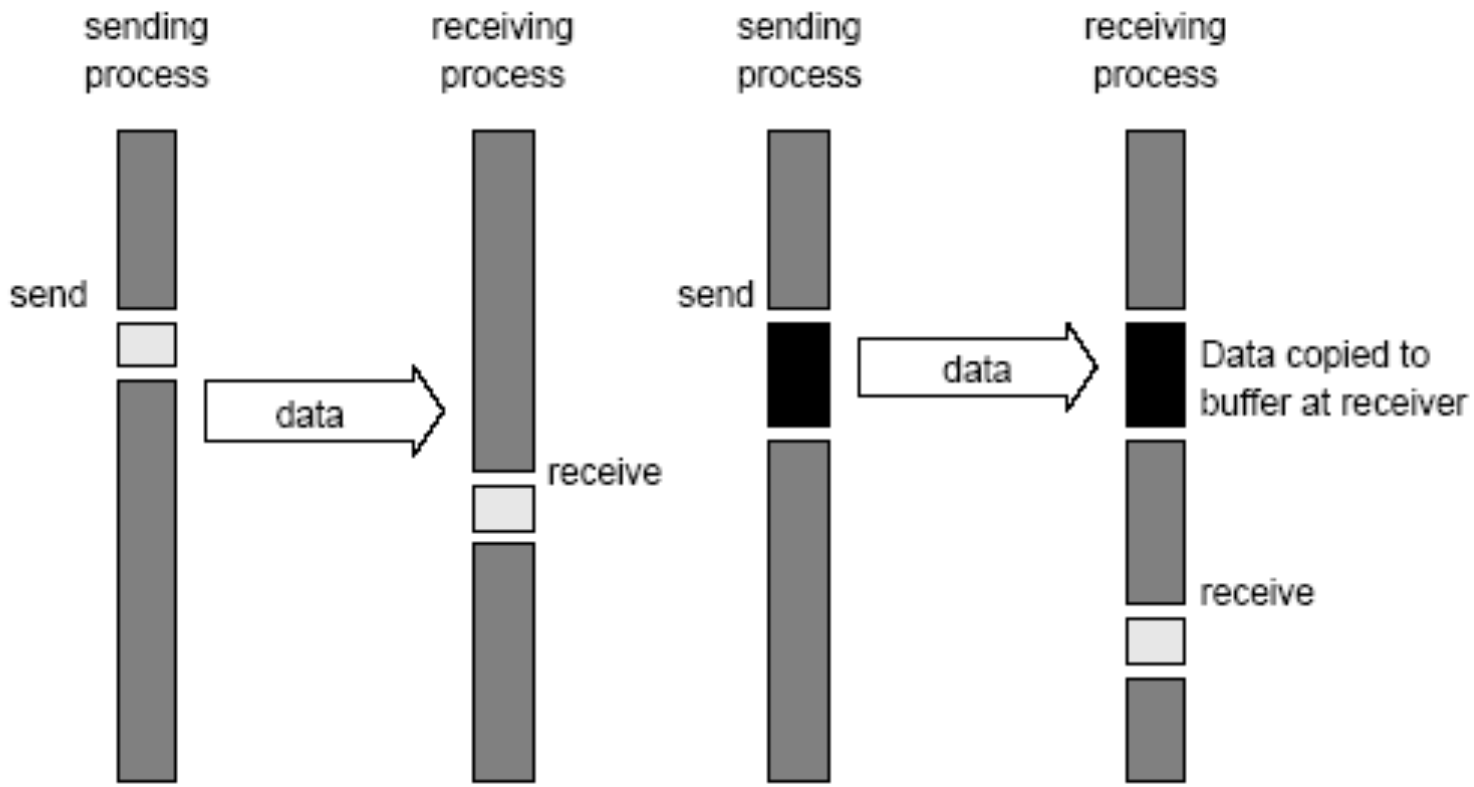
Blocking communication

- 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 behavior: **`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_SEND`
  - Without receiver match, outcome is undefined
  - Can omit a handshake-operation on some systems

Blocking Buffered Send



Blocking Buffered Send

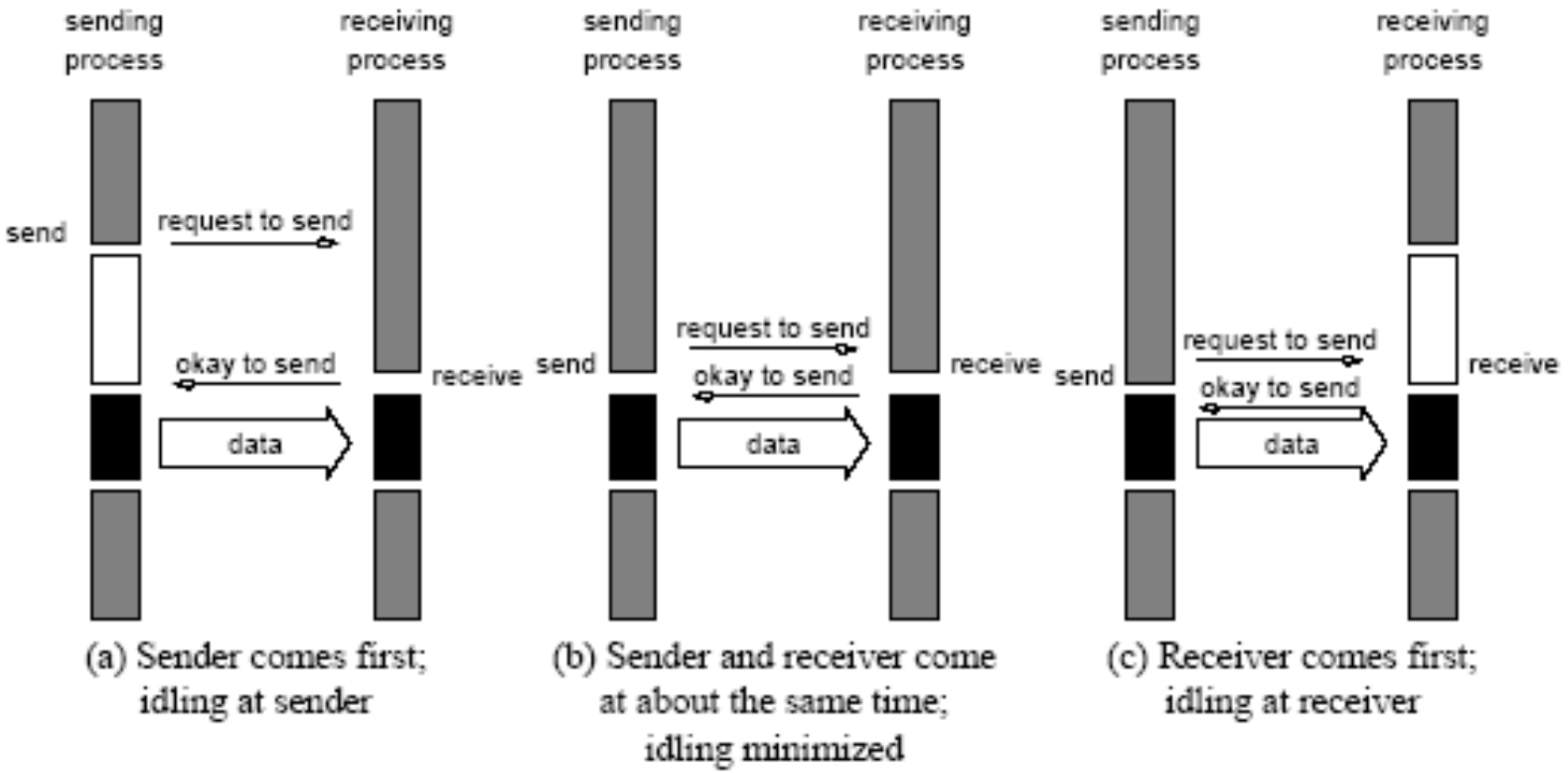
Bounded buffer sizes can have significant impact on performance.

```
P0
for (i = 0; i < 1000; i++){
    produce_data(&a);
    send(&a, 1, 1);
 }
```

```
P1
 for (i = 0; i < 1000; i++){
    receive(&a, 1, 0);
    consume_data(&a);
}
```

What if consumer was much slower than producer?

Blocking Non-Buffered Send



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

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.“

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_BSEND (buf1, count, MPI_REAL, 1, tag, comm, ierr)
   CALL MPI_BSEND (buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE   ! rank.EQ.1
   CALL MPI_RECV (buf1, count, MPI_REAL, 0,
                  MPI_ANY_TAG, comm, status, ierr)
   CALL MPI_RECV (buf2, count, MPI_REAL, 0,
                  tag, comm, status, ierr)
END IF
```

Deadlocks

Consider:

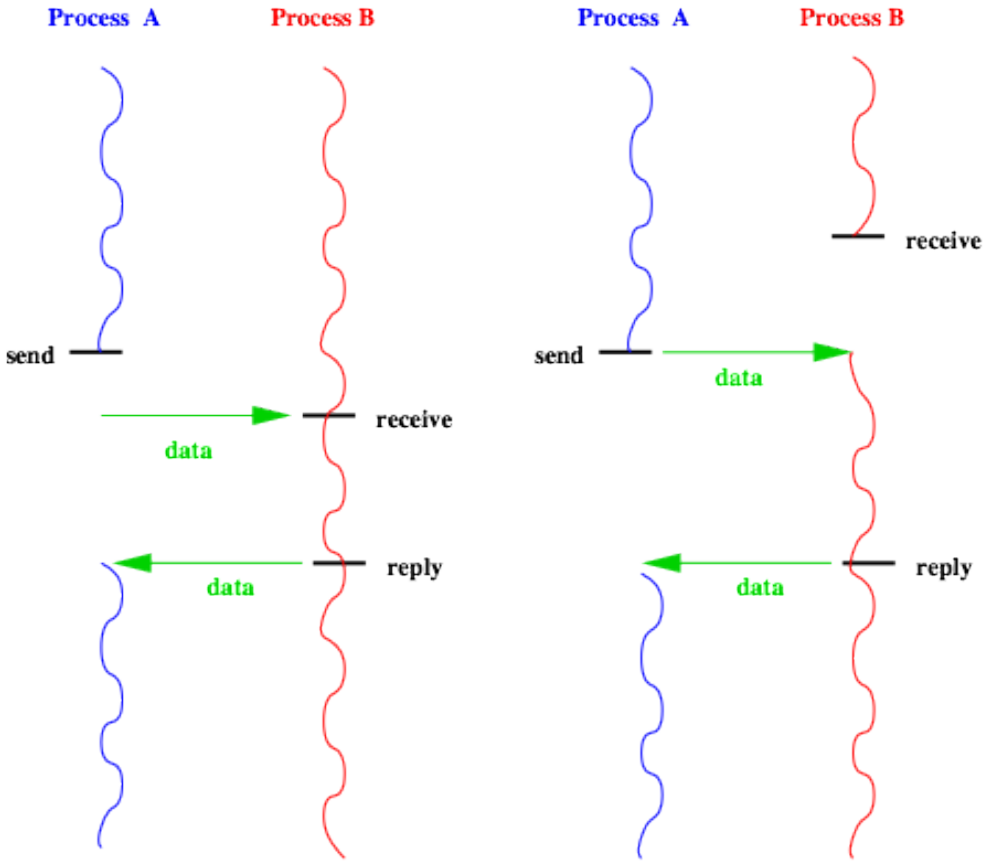
```
int MPI_Send(void* buf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm com);
```

```
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 blocking, there is a deadlock.

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

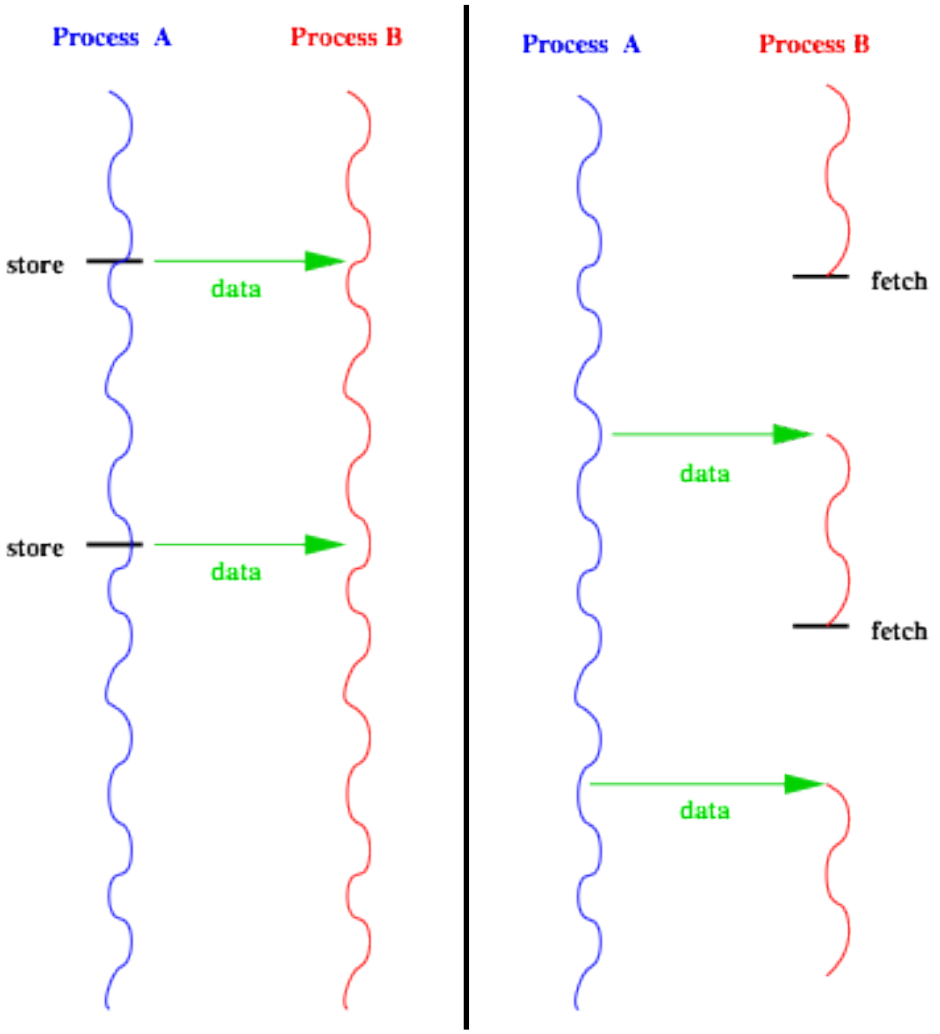



```
int MPI_Sendrecv( void* sbuf, int scount, MPI_Datatype stype,
int dest, int stag, void* rbuf, int rcount, MPI_Datatype rtype,
int src, int rtag, MPI_Comm com, MPI_Status* status);
```

One-Sided Communication

- No explicit receive operation, but synchronous remote memory access

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



One-Sided Communication

Hasso Plattner Institut

(C) http://mpi.deino.net

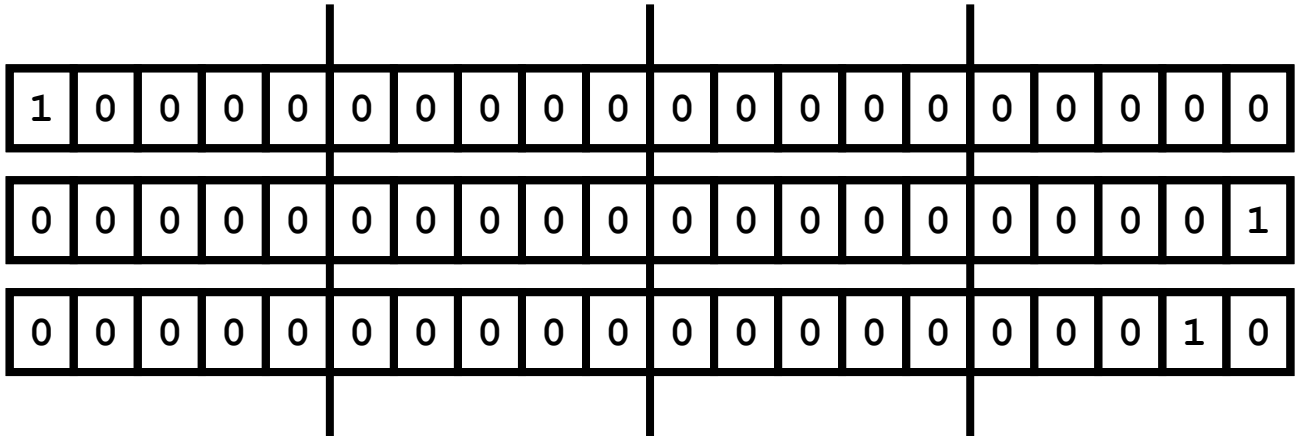
```
#include "mpi.h"
#include "stdio.h"
#define SIZE1 100
#define SIZE2 200
int main(int argc, char *argv[]) {
    int rank, destrank, nprocs, *A, *B, i, errs=0;
    MPI_Group comm_group, group; MPI_Win win;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs); MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Alloc_mem(SIZE2 * sizeof(int), MPI_INFO_NULL, &A);
    MPI_Alloc_mem(SIZE2 * sizeof(int), MPI_INFO_NULL, &B);
    MPI_Comm_group(MPI_COMM_WORLD, &comm_group);
    if (rank == 0) {
        for (i=0; i<SIZE2; i++) A[i] = B[i] = i;
        MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &win);
        destrank = 1;
        MPI_Group_incl(comm_group, 1, &destrank, &group);
        MPI_Win_start(group, 0, win);
        for (i=0; i<SIZE1; i++)
            MPI_Put(A+i, 1, MPI_INT, 1, i, 1, MPI_INT, win);
         for (i=0; i<SIZE1; i++)
            MPI_Get(B+i, 1, MPI_INT, 1, SIZE1+i, 1, MPI_INT, win);
        MPI_Win_complete(win);
    } else { /* rank=1 */
        for (i=0; i<SIZE2; i++) B[i] = (-4)*i;
        MPI_Win_create(B, SIZE2*sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &win);
        destrank = 0;
        MPI_Group_incl(comm_group, 1, &destrank, &group);
        MPI_Win_post(group, 0, win);      // matches to MPI_Win_start
        MPI_Win_wait(win);                // matches to MPI_Win_complete
    }
...}
```

Circular Left Shift Example

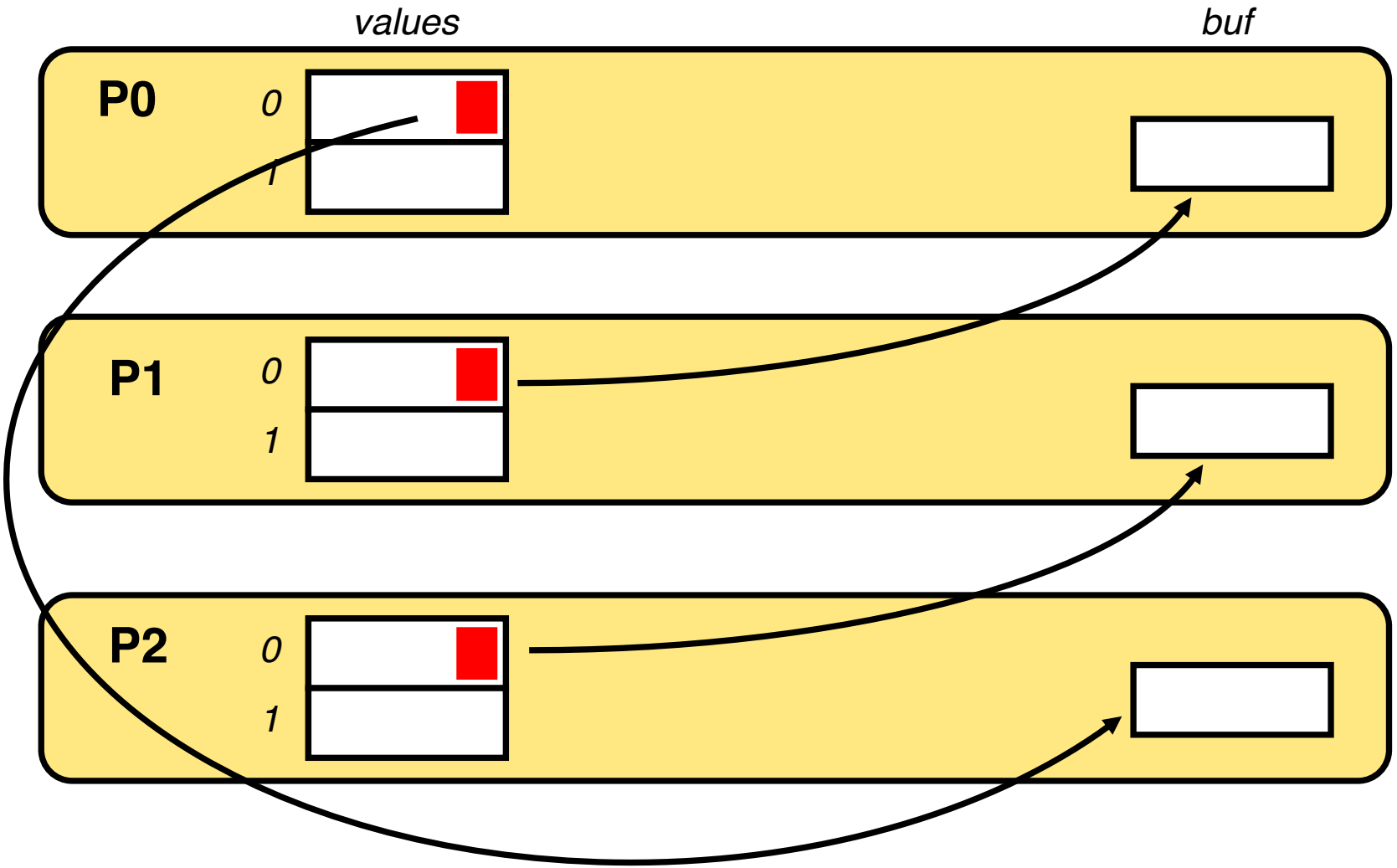
shifts <number of positions>

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



Circular Left Shift Example

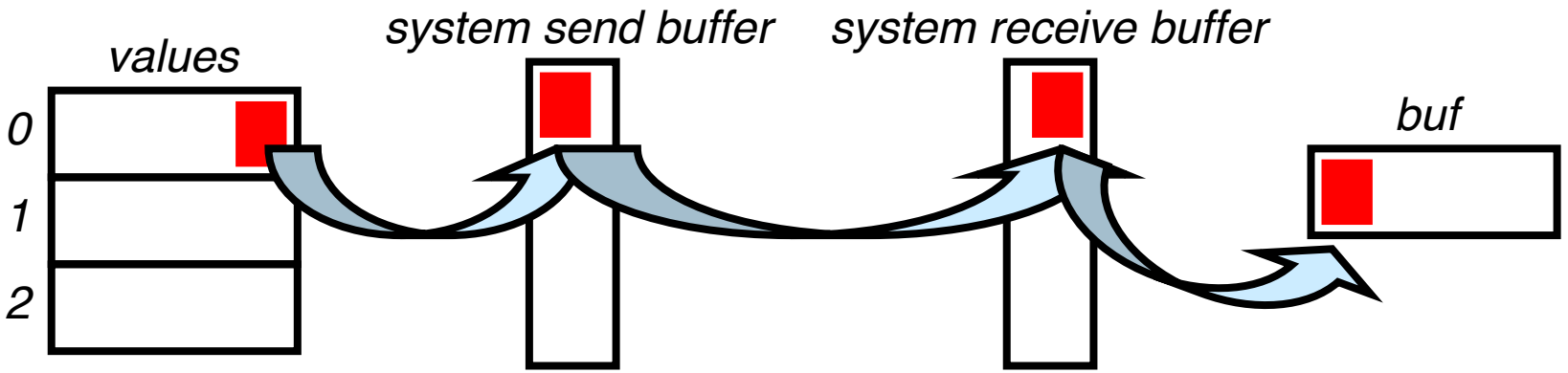

```
#include "mpi.h"

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){
```

Process 0

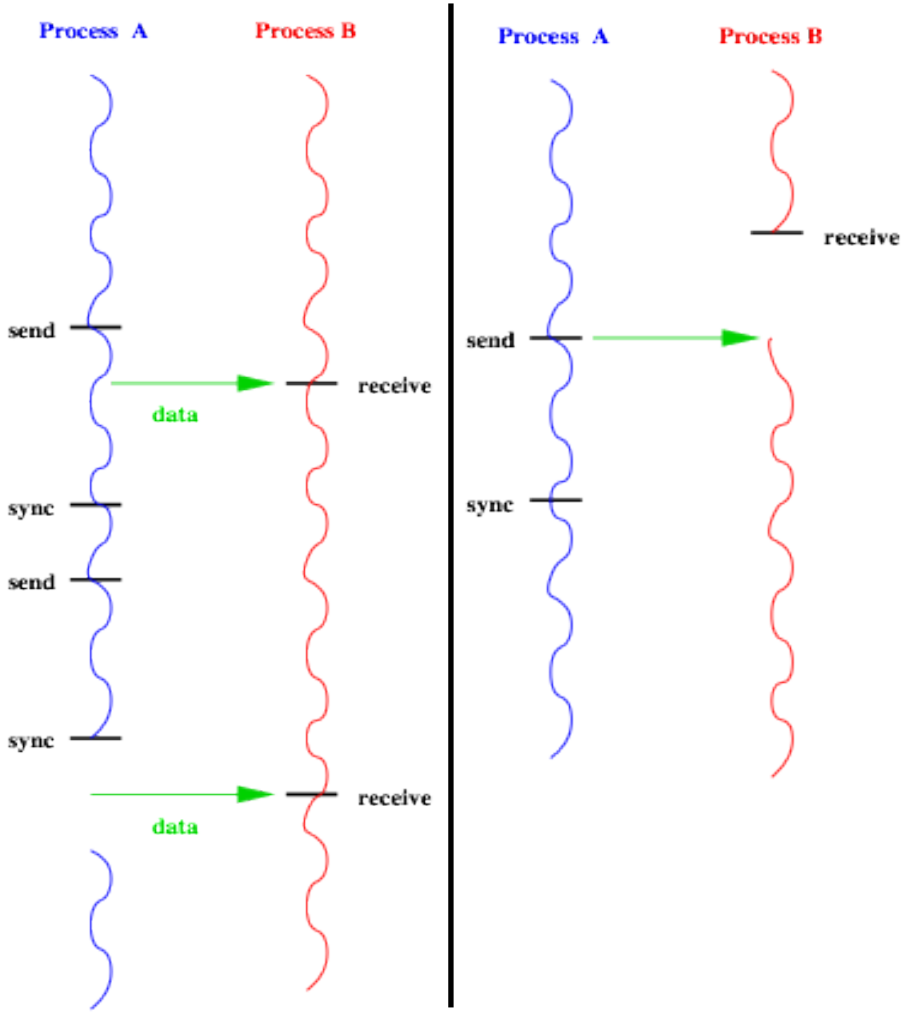
```
    MPI_Send(&values[0], 1, MPI_INT, lnbr, 10,
             MPI_COMM_WORLD);
    for (j=1;j<100/np;j++){
      values[j-1]=values[j];
    }
    MPI_Recv(&values[100/np-1], 1, MPI_INT, rnbr,
             10, MPI_COMM_WORLD, &status);
  }else{
```

Other Processes

```
    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);
  }
}
```

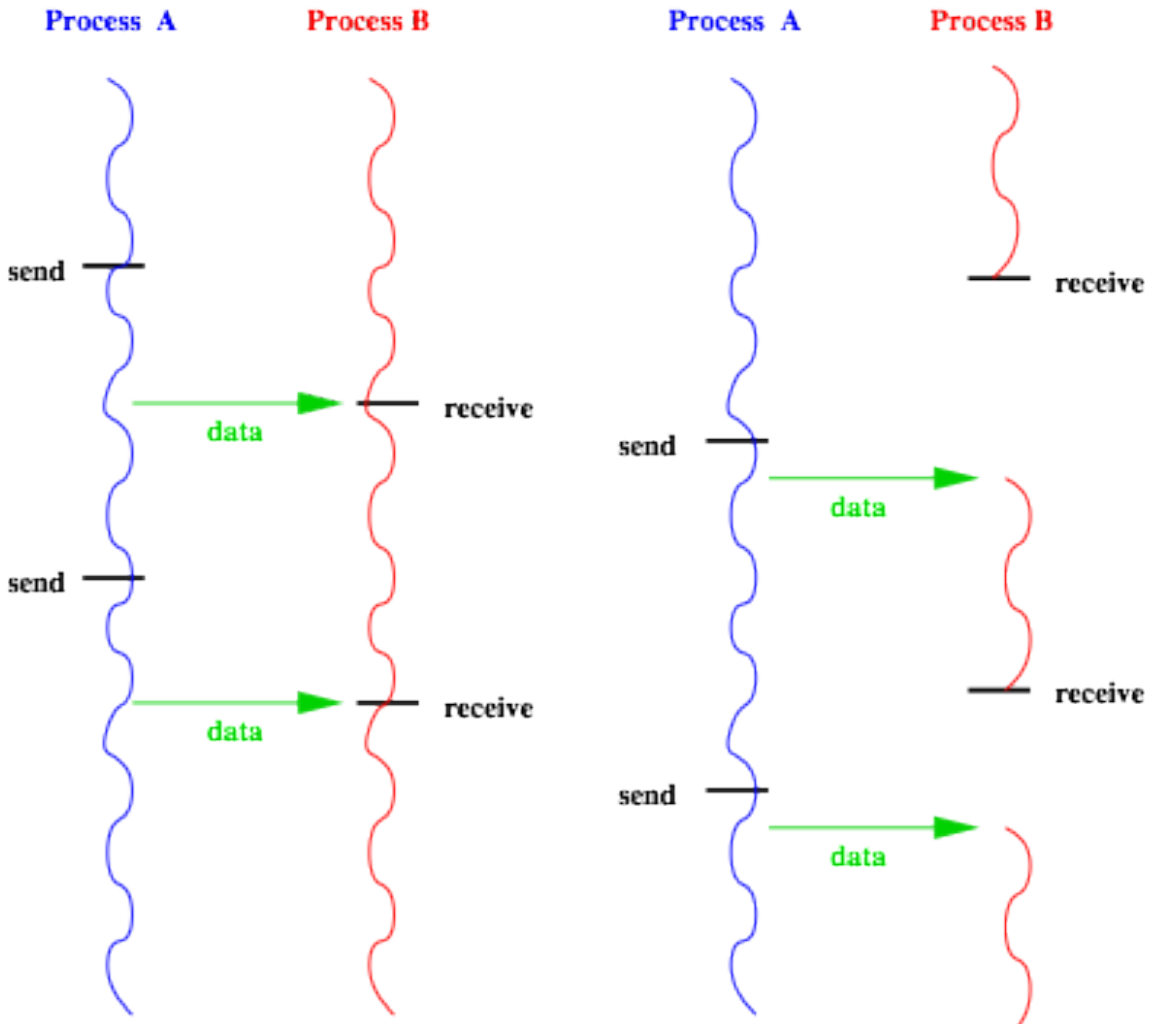
Non-Blocking Communication

- 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: Buffering on Send

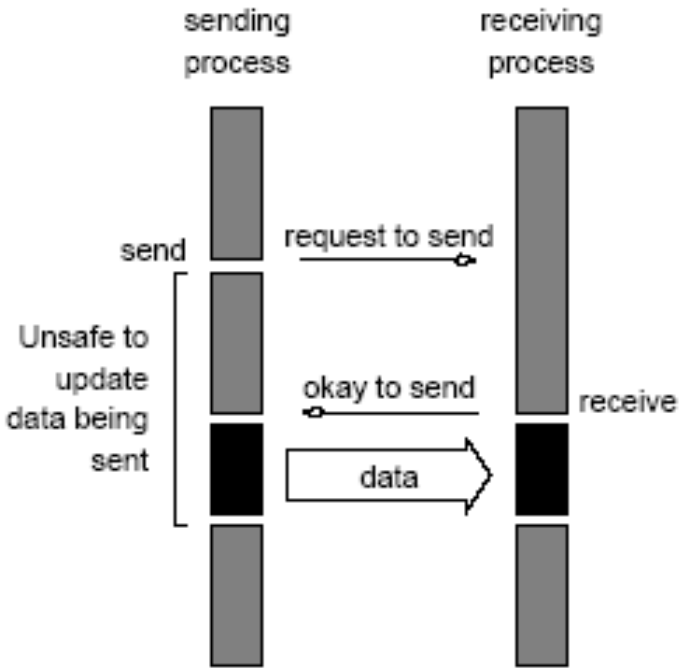
- Data is stored in the communication stack of the sender side
- Receiver only gets notification about available data
- Receiver triggers data transfer by reaction
- Additional communication overhead, useful only for few large transfers



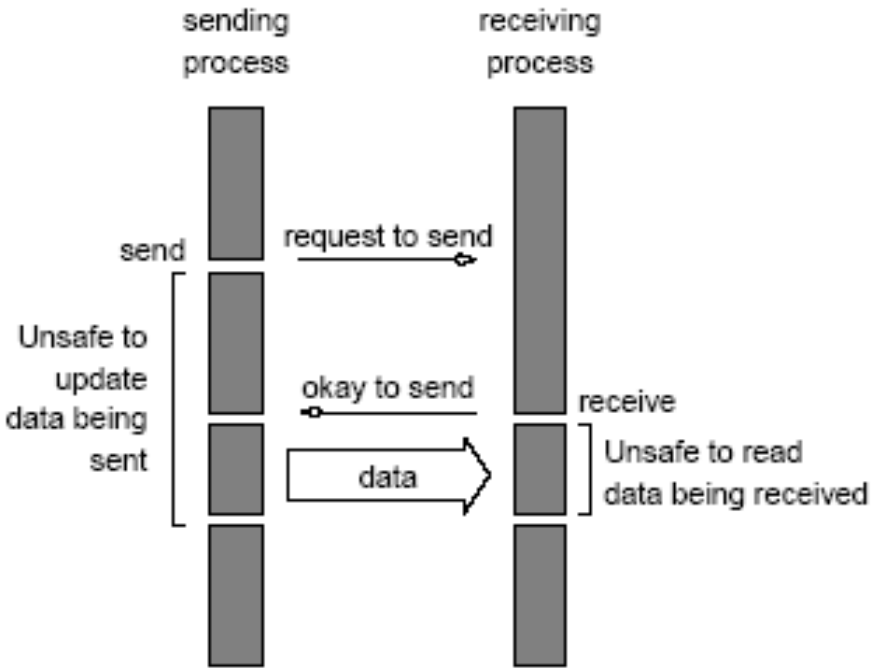
MPI Non-Blocking Communication

- Send/receive start and send/receive completion calls with additional request handle
- ‚Immediate send' calls:
  `MPI_ISEND, MPI_IBSEND, MPI_ISSEND, MPI_IRSEND`
- Completion calls
  - `MPI_WAIT, MPI_TEST, MPI_WAITANY, MPI_TESTANY, MPI_WAITSOME, ...`
- `MPI_IBSEND:` Always immediate return of the completion call
- `MPI_ISSEND:` Return of the completion call on receiver start
- …
- Sending side cleanup: `MPI_REQUEST_FREE`

Non-Blocking Non-Buffered Send



(a) Without hardware support



(b) With hardware support

Non-Blocking Communication Without Buffering

Hasso Plattner Institut

```
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

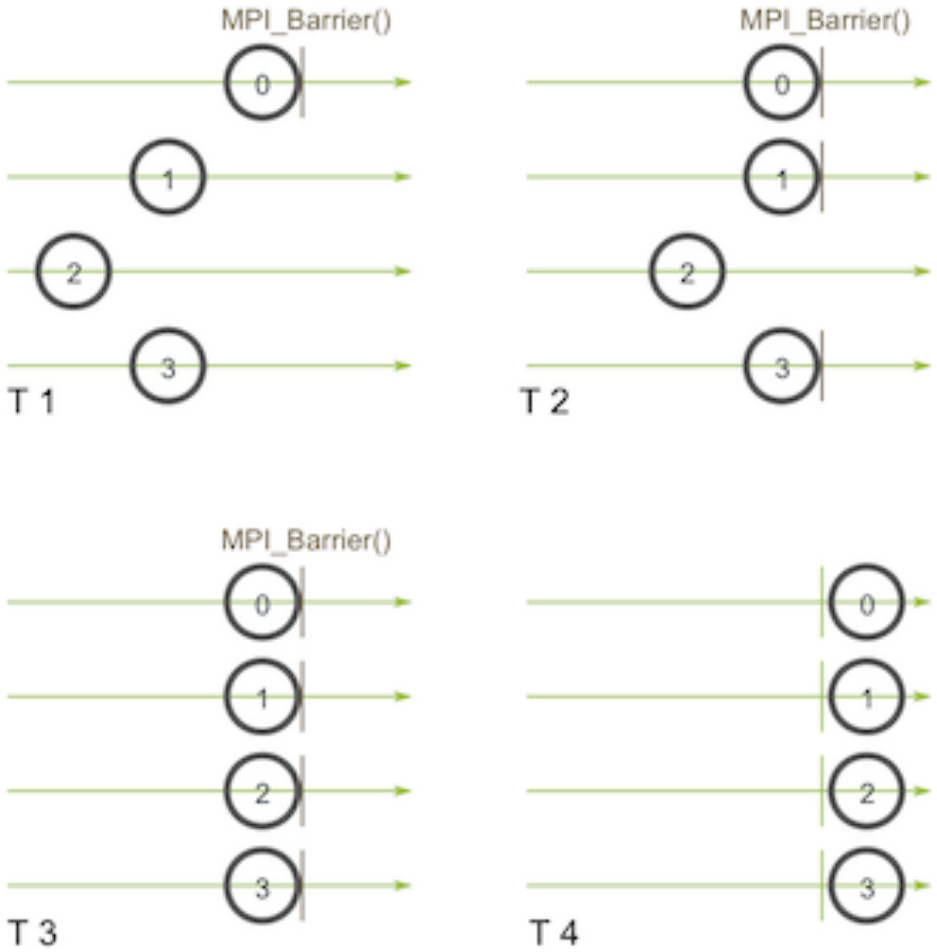
	Blocking	Non-Blocking
Buffered	Send call returns after data has been buffered *MPI_BSend*	Send call returns after initiating DMA transfer to the buffer *MPI_IBSend*
Non-Buffered	Send call returns after matching receive is Available *MPI_SSend*	No semantics promised. *MPI_ISSend*

Collective Communication

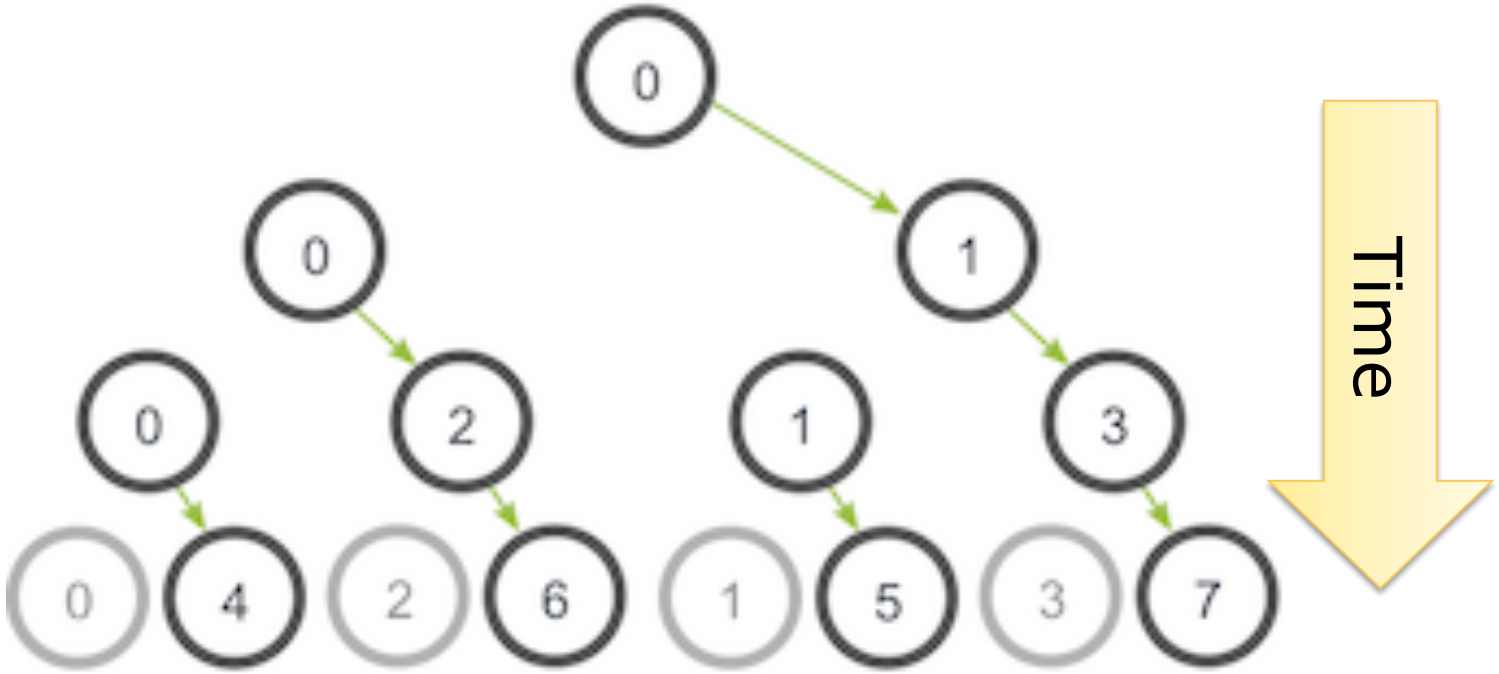
- Point-to-point communication vs. collective communication
- Use case: Synchronization, communication, reduction
- All communication of processes belonging to a group
  - One sender with multiple receivers (‚one-to-all‘)
  - Multiple senders with one receiver (‚all-to-one‘)
  - Multiple senders and 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 of a group are blocked until everybody reached the barrier

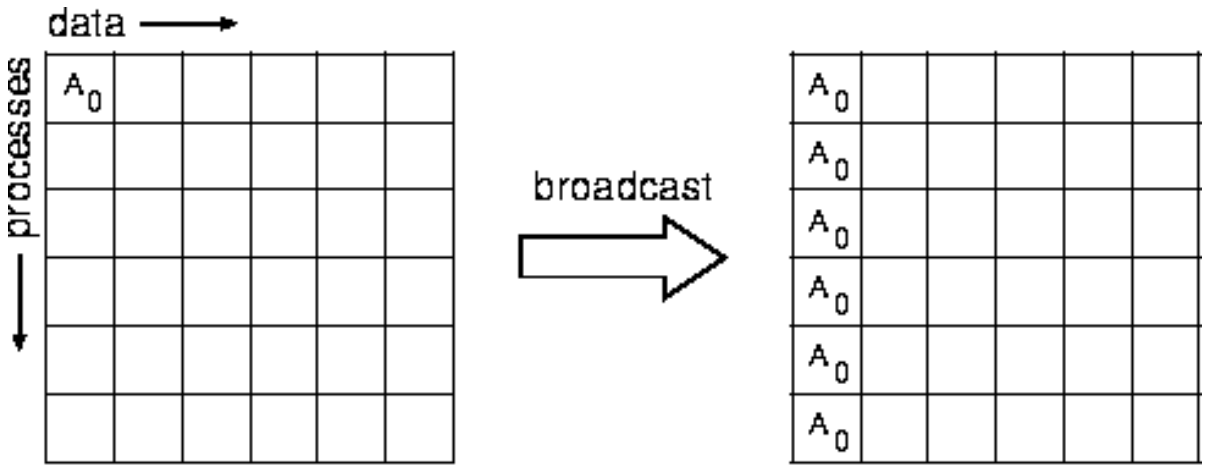


Efficient Barrier Implementation

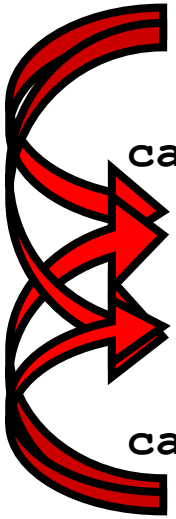


Collective Communication

- `MPI_BCAST (INOUT buffer, IN count, IN datatype, IN rootPid, IN 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



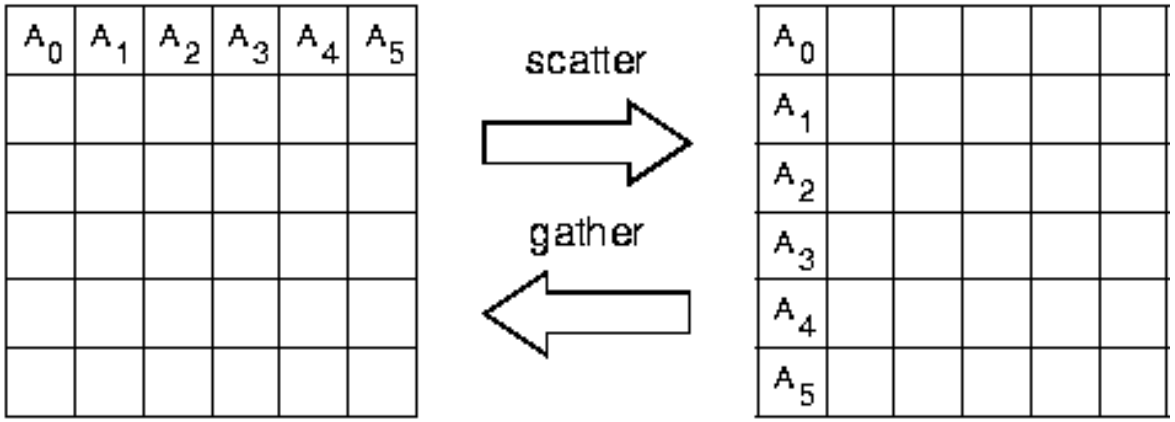
MPI Broadcast



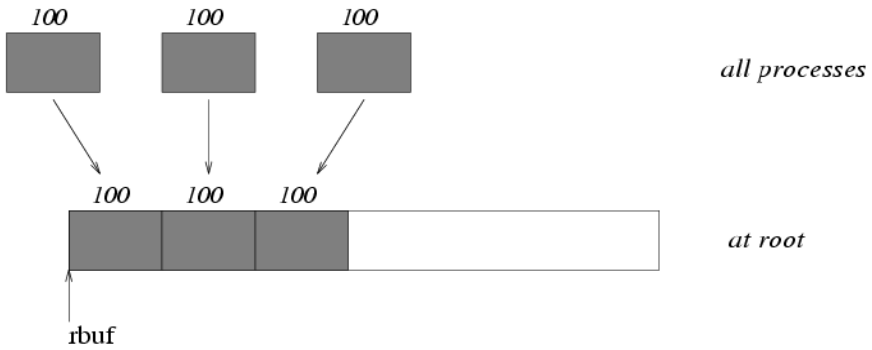
```
switch (rank) {
case 0:
  MPI_Bcast (buf1, ct, tp, 0, comm);
  MPI_Send (buf2, ct, tp, 1, tag, comm);
  break;
case 1:
  MPI_Recv (buf2, ct, tp, MPI_ANY_SOURCE, tag, ...);
  MPI_Bcast (buf1, ct, tp, 0, comm);
  MPI_Recv (buf2, ct, tp,MPI_ANY_SOURCE, tag, ...);
  break;
case 2:
  MPI_Send (buf2, ct, tp, 1, tag, comm);
  MPI_Bcast (buf1, ct, tp, 0, comm);
  break;
}
```

Gather

- `MPI_GATHER ( IN sendbuf, IN sendcount, IN sendtype, OUT recvbuf, IN recvcount, IN recvtype, IN root, IN 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
  - `MPI_GATHERV` allows varying count of data to be received
  - Returns if the buffer is re-usable (no finishing promised)



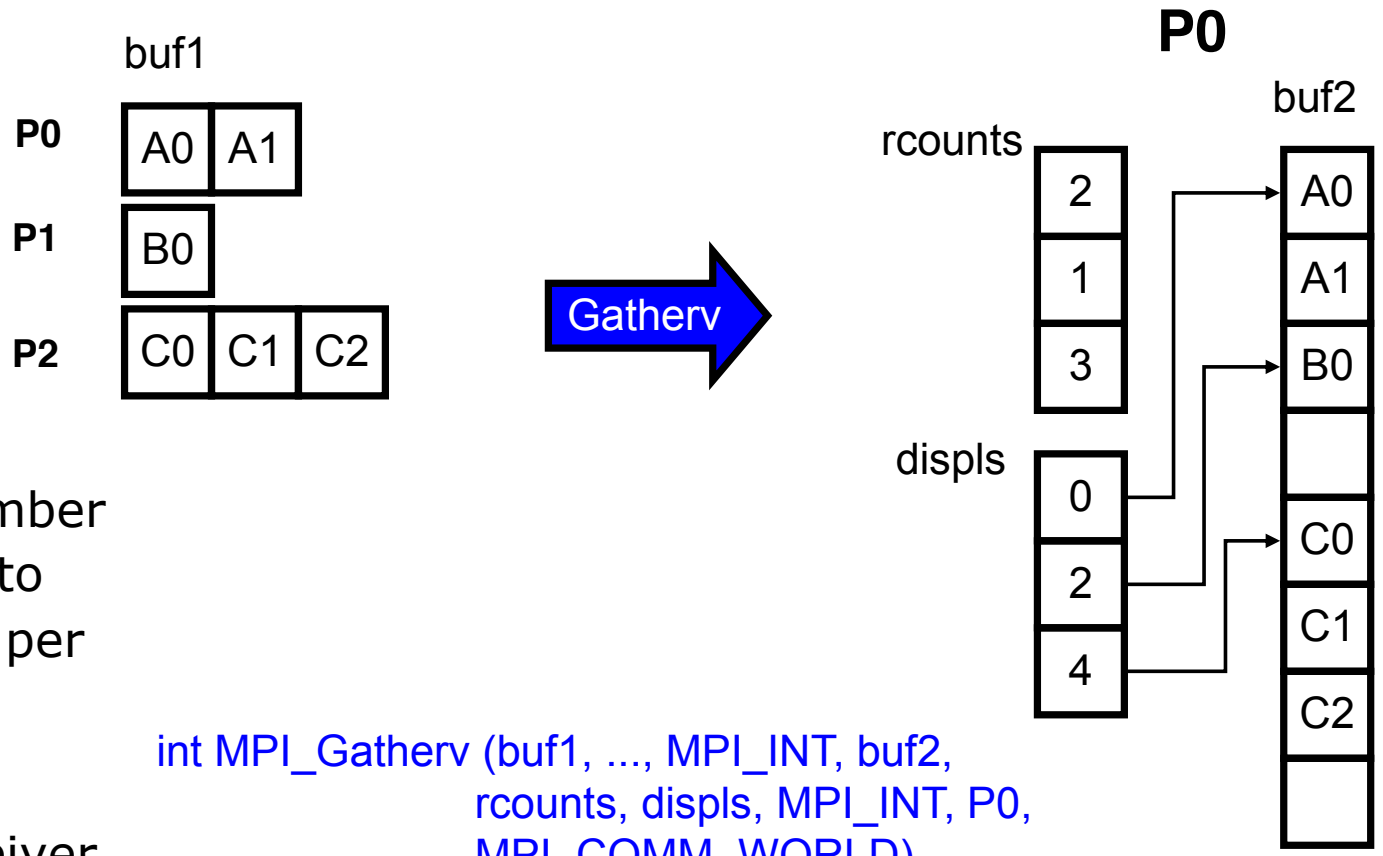
MPI Gather Example




```
MPI_Comm comm;
int gsize,sendarray[100];
int root, myrank, *rbuf;
... [compute sendarray]
MPI_Comm_rank( comm, myrank);
if ( myrank == root) {
  MPI_Comm_size( comm, &gsize);
  rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather ( sendarray, 100, MPI_INT, rbuf, 100,
            MPI_INT, root, comm );
```

Ignored on all non-root

MPI_Gatherv



- *rcounts*: Number of elements to be retrieved per process
- *displs*: First index in receiver buffer per peer process

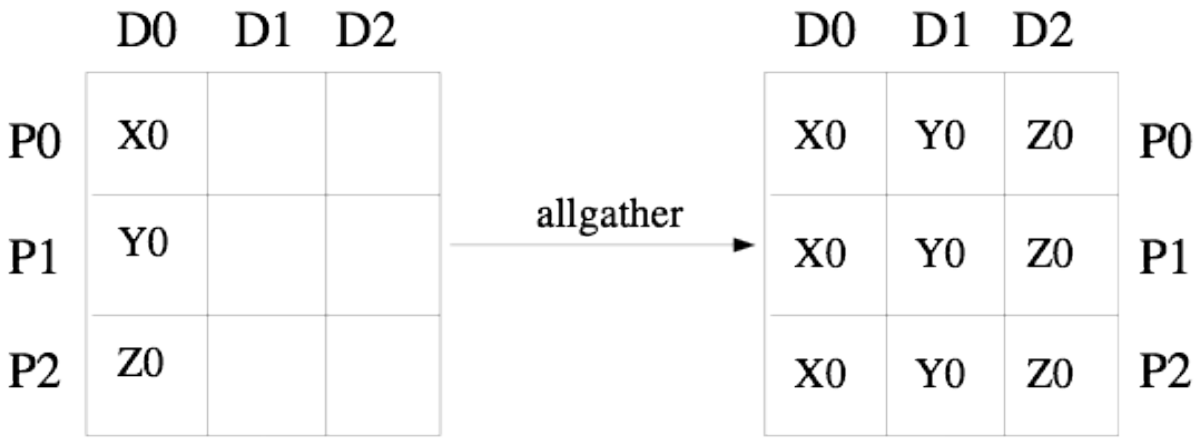
```
int MPI_Gatherv (buf1, ..., MPI_INT, buf2,
                 rcounts, displs, MPI_INT, P0,
                 MPI_COMM_WORLD)
```

Scatter

- `MPI_SCATTER ( IN sendbuf, IN sendcount, IN sendtype, OUT recvbuf, IN recvcount, IN recvtype, IN root, IN 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
  - `MPI_SCATTERV` allows varying count of data to be send to each process
  - Returns if data buffer is re-usable, not necessarily finished

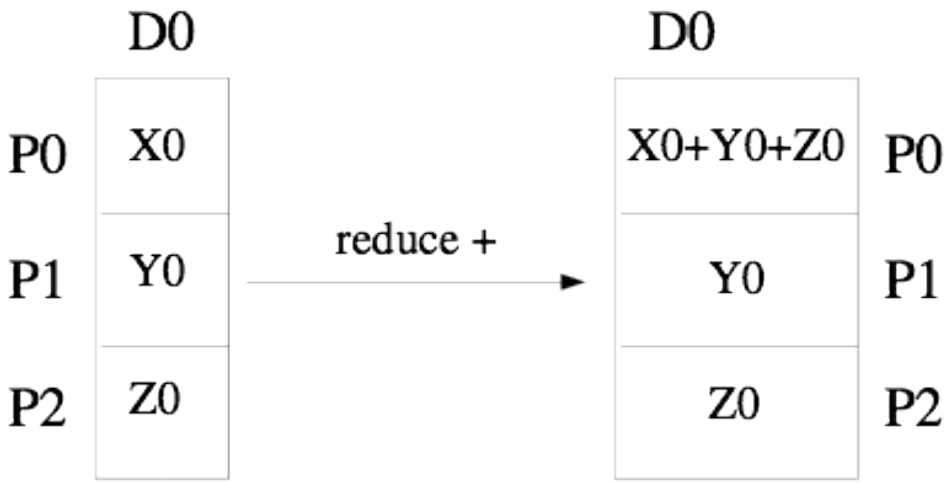
Allgather

- 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



Reduction

- Similar to gather operation, all group members participate with their data
- Partial results are accumulated by reduction operation
  - Typical example: Global sum / product
  - Mostly only commutative or associative operations
  - Reduction can be performed in parallel to the communication



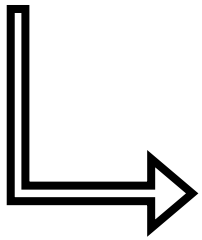
Reduction


```
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
```

Reduction


```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    b[i]=b[i]+a[i][j]
```



```
for (i=0; i<n; i++)
 for (j=0;j<local_n; j++)
  b[i]=b[i]+a[i][j]

  MPI_Reduce(b, b1, n,
             MPI_INT, MPI_SUM, P0,
             MPI_COMM_WORLD)
```

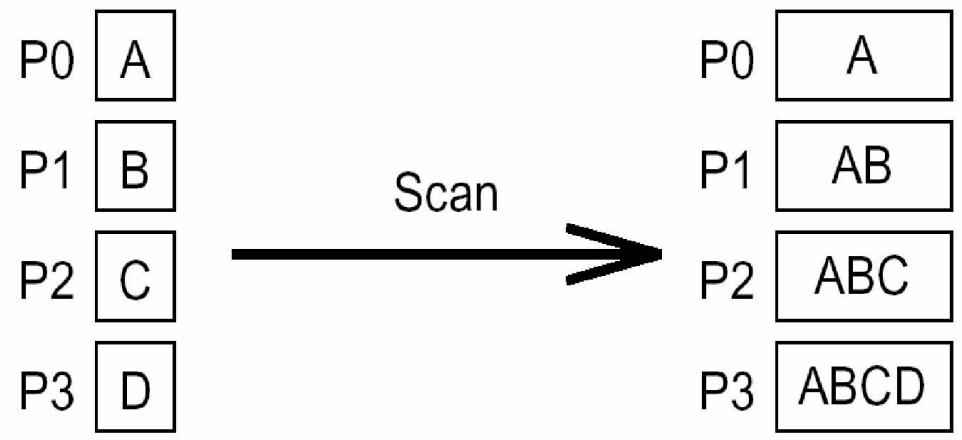
Predefined Reduction Operators

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 Prefix Scan


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

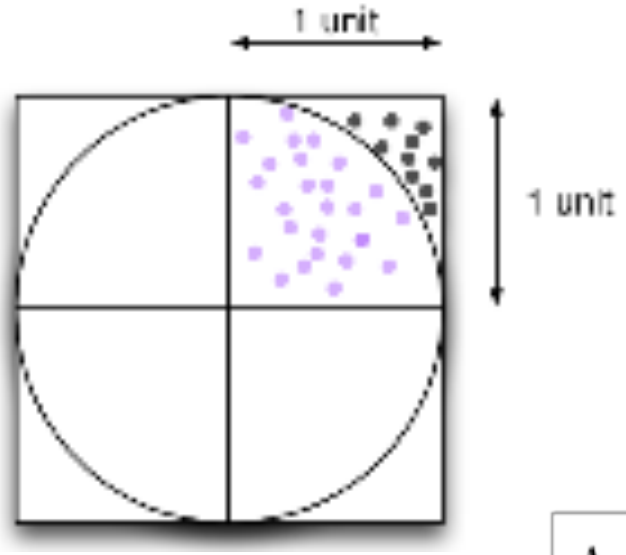
- Computes the inclusive reduction result of the send buffer
- Each result buffer element *i* holds the reduction until rank *i*
- Operations and constraints are the same as with *MPI_Reduce*



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;
}
```

Example: Estimating PI [Chen et al.]



$$\frac{\text{Area of quarter unit circle}}{\text{Area of quarter of square}} = \frac{\frac{(\pi)(1\ \text{unit})^2}{4}}{(1\ \text{unit})^2}$$

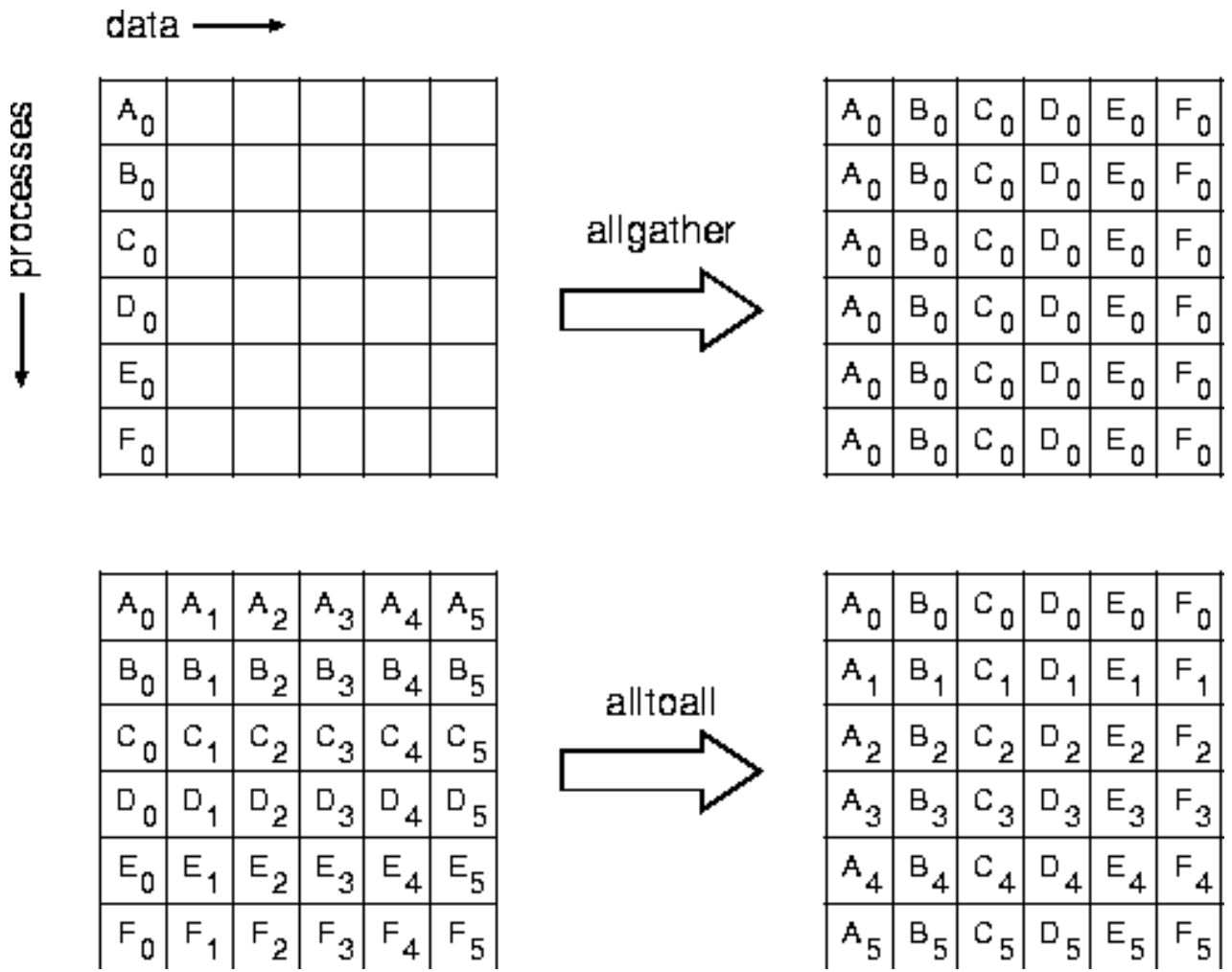
$$\approx \frac{\text{Points in circle}}{\text{Total points in first quadrant}}$$

$$\Rightarrow \pi \approx 4 * \frac{\text{Points in circle}}{\text{Total points in first quadrant}}$$

```
17 int count_points_in_circle(int* stream_id){
18   int i, my_count = 0;
19   double x, y, distance_squared;
20   for(i = 0; i < ITERATIONS; i++) {
21     x = (double)sprng(stream_id);
22     y = (double)sprng(stream_id);
23     distance_squared = x*x + y*y;
24     if(distance_squared <= RADIUS) my_count++;
25   }
26   return my_count;
27 }
28
29 int main(int argc, char* argv[]) {
30
31   int my_id;
32   int number_of_processors;
33
34   //
35   // Initialize MPI and set up SPMD programs
36   //
37   MPI_Init(&argc,&argv);
38   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
39   MPI_Comm_size(MPI_COMM_WORLD, &number_of_processors);
40
41   //
42   // Initialize pseudo parallel random number generator
43   //
44   generate_seed();
45   int* stream_id = init_sprng(SPRNG_LFG, my_id, number_of_processors, rand(),
46                               SPRNG_DEFAULT);
47
48   int my_count = count_points_in_circle(stream_id);
49
50   //
51   // Combine the results from different UEs
52   //
53   int total_count;
54   MPI_Reduce(&my_count, &total_count, 1, MPI_INT, MPI_SUM, MASTER_NODE,
55     MPI_COMM_WORLD);
56
57   if(my_id == MASTER_NODE) {
58     double estimated_pi = (double)total_count /
59                                 (ITERATIONS * number_of_processors) * 4;
60     printf("The estimate of pi is %g\n", estimated_pi);
61   }
62
63   MPI_Finalize();
64
65   return 0;
66 }
```

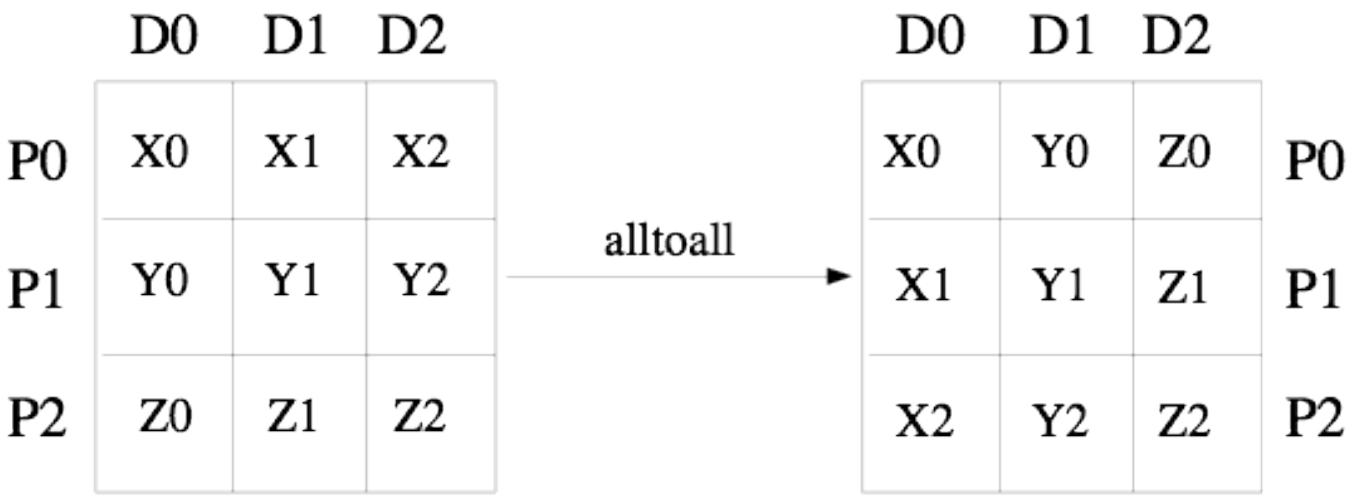
HPI Hasso Plattner Institut

MPI_Allgather



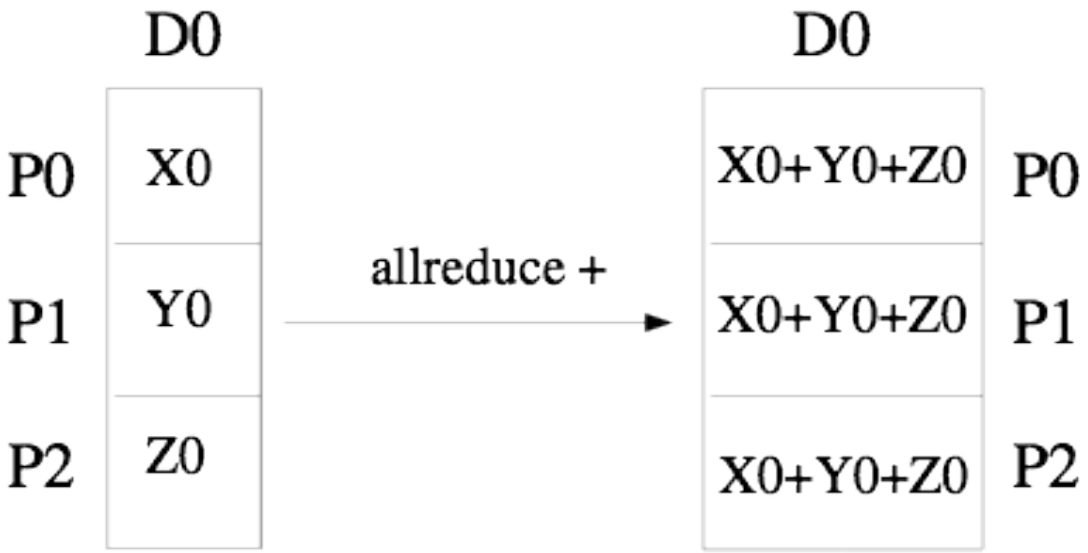
MPI_Alltoall

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



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

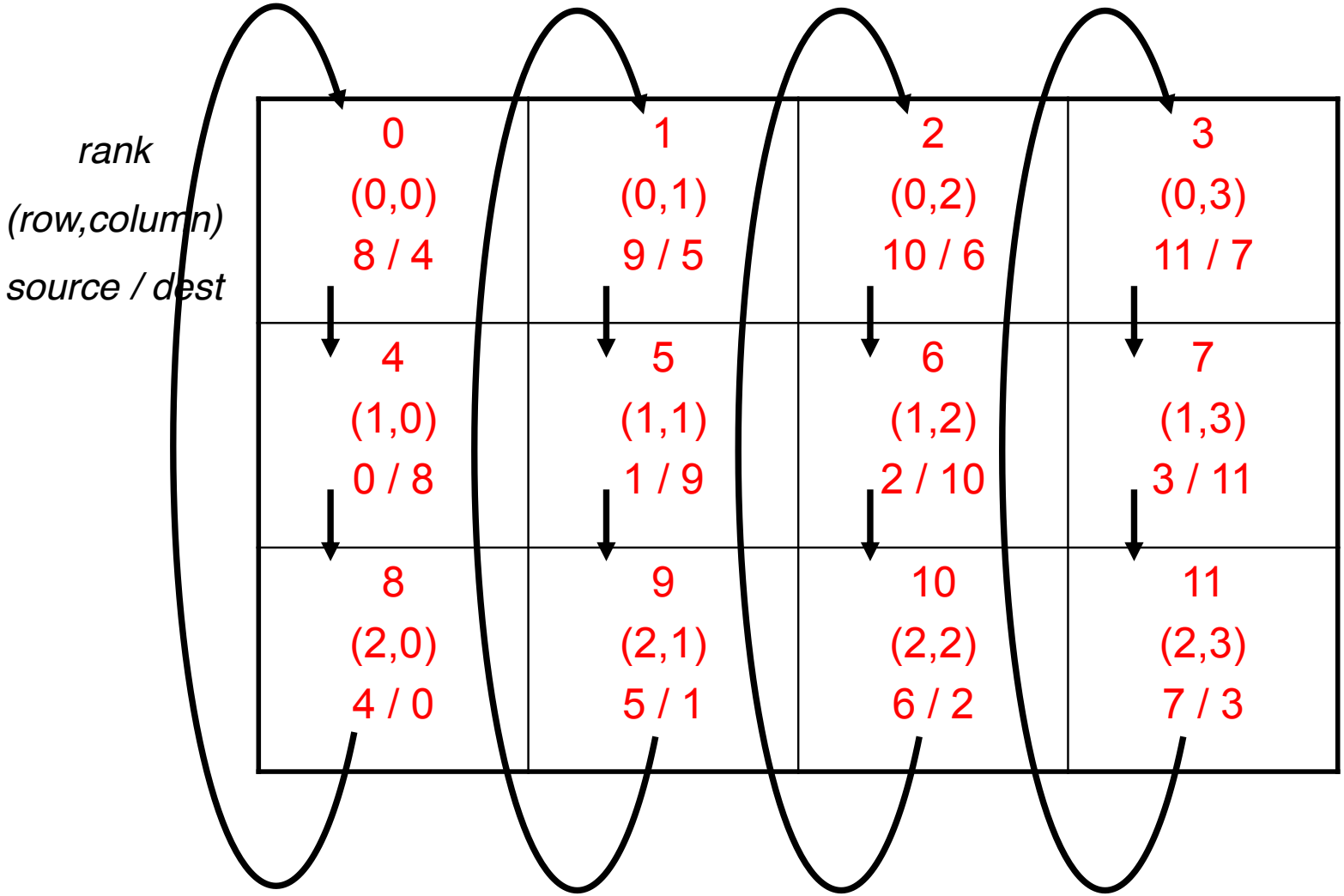
- Topologies help to define a virtual name space structuring
  - Effective mapping of processes to nodes
  - Optimizations for interconnection networks (grids, tori, ...)
- Access through a newly defined communicator
  - *MPI_Cart_create(oldcomm, ndims, dims, periods, reorder, new_comm)*
  - Define structure 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 neigbour

MPI Process Topologies



What Else

- Complex data types
- Packing / Unpacking (sprintf / sscanf)
- Group / Communicator Management
- Error Handling
- Profiling Interface
- Several implementations available
  - MPICH - Argonne National Laboratory; Shared memory or networking
  - OpenMPI - Consortium of Universities and Industry
  - ...