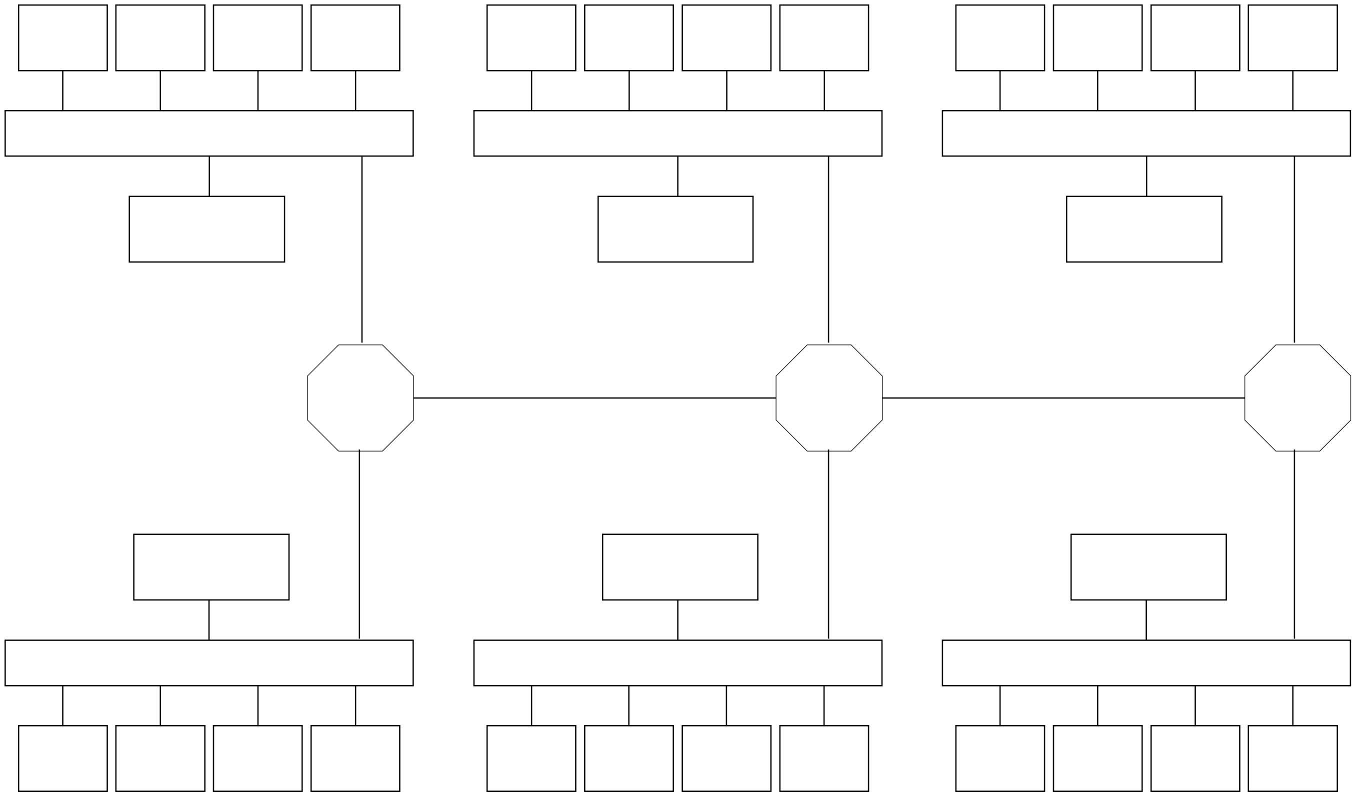
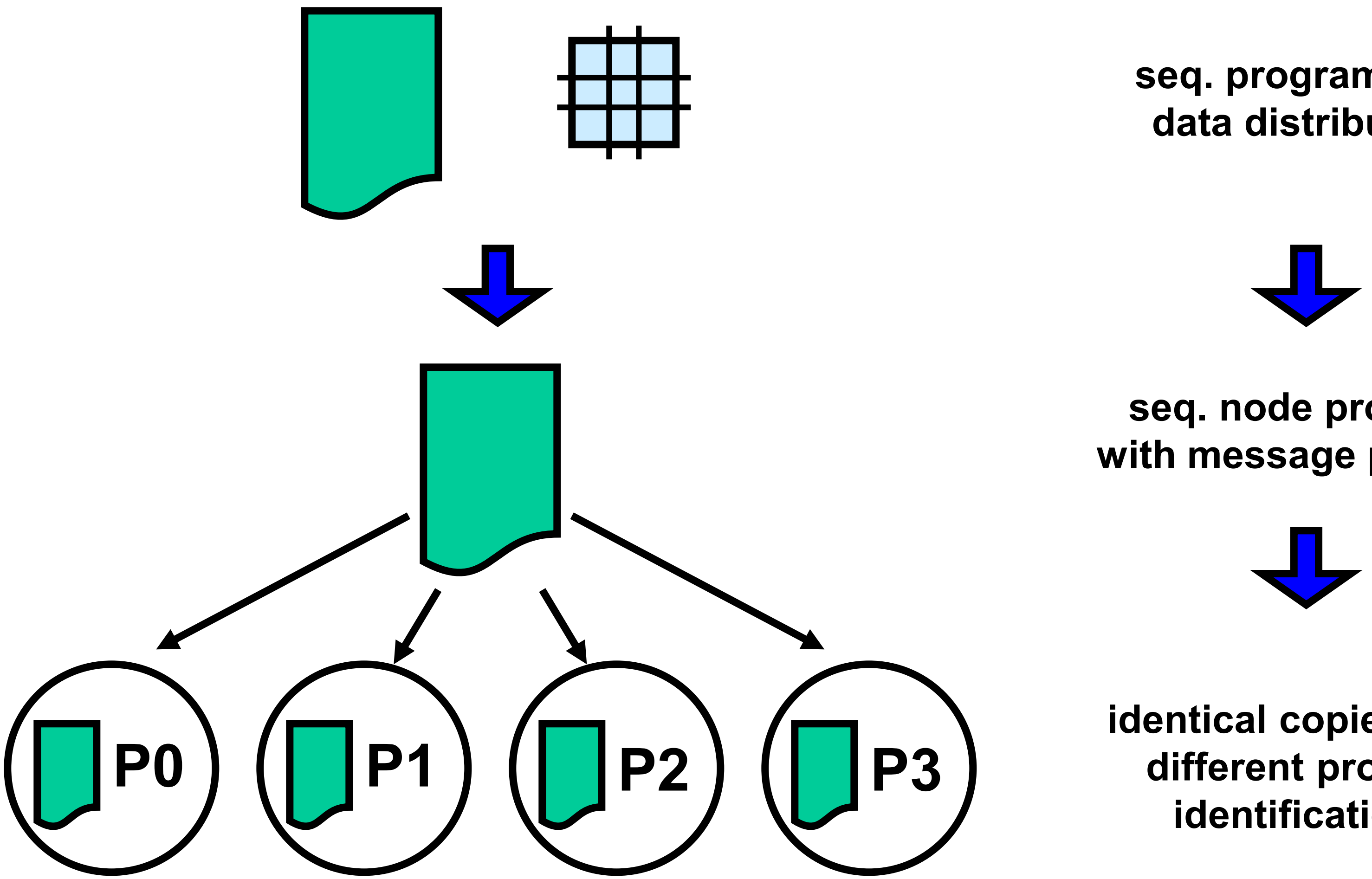
Introduction to the Parallelization with MPI

Michael Gerndt

Distributed Memory Architektur



Single Program Multiple Data (SPMD)



seq. program and data distribution

seq. node program with message passing

identical copies with different process identifications

Program Parallelization

1. Adaptation of array declarations
   - Local size of distributed arrays covers only the part of the data structure assigned to the process.
2. Index transformation
   - Global indices are mapped into a tupel of node number and of a local index.
3. Work distribution
   - Computations are executed by the process owning the assigned variable.
4. Communication
   - Accesses to array elements of other processes have to be implemented by message passing.

Scope of the Message Passing Interface

- MPI 1.2
  - Point-to-Point communication
  - Collective communication
  - Communicators
  - Process topologies
  - User-defined data types
  - Operations and properties of the execution environment
  - Profiling interface
- MPI 2.0
  - Dynamic process creation
  - One-sided communication
  - Parallel IO
- www-unix.mcs.anl.gov/mpi/

Core Routines

- MPI 1.2 has 129 functions
- It is possible to write real programs with only six functions:
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv

MPI_Init

```
int MPI_Init (int *argc, char ***argv)

        IN argc, argv:   arguments
        return:          MPI_SUCCESS or error codes
```

- This routine has to be called by each MPI process before any other MPI routine is executed

- Fortran interface
  - MPI_INIT (integer ierror)
  - The name is written in capital letters and the error code is returned via an additional argument.

MPI_Finalize

```
int MPI_Finalize ()
```

- Each process must call MPI_FINALIZE before it exits.
  - Precondition: All pending communication has to be finished.
  - One MPI_FINALIZE returns, no further MPI routines can be executed.
  - MPI_FINALIZE frees any resources.

MPI_Comm_size

```
int MPI_Comm_size (MPI_Comm comm, int *size)
```

IN comm: *Communicator*
OUT size: *Cardinality of the process group*

- Communicator
  - Identifies a process group and defines the communication context. All message tags are unique with respect to a communicator.
- MPI_COMM_WORLD
  - This is a predefined standard communicator. Its process group includes all processes of a parallel application.
- MPI_Comm_size
  - It returns the number of processes in the process group of the given communicator.

MPI_Comm_rank

int MPI_Comm_rank (MPI_Comm comm, int *rank)

IN comm: *Communicator*
OUT rank: *process number of the executing process*

- Process number
  - The process number is a unique identifier within the process group of the communicator.
  - It is the only way to distinguish processes and to implement an SPMD program.
- MPI_Comm_rank returns the process number of the executing process.

MPI_Send

```
int MPI_Send (void *buf, int count, MPI_Datatype dtype, int dest, int tag,
              MPI_Comm comm)
```

IN buf:	*Address of the send buffer*
IN count:	*Number of data to be sent*
IN dtype:	*Data type*
IN dest:	*Receiver*
IN tag:	*Message tag*
IN comm:	*Communicator*

- MPI_Send
  - Sends the data to the receiver.
  - It is a blocking operation, i.e. it terminates when the send buffer can be reused, either because the message was delivered or the data were copied to a system buffer.

MPI Data Types

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_Recv

```
int MPI_Recv (void *buf, int count, MPI_Datatype dtype, int source, int
              tag, MPI_Comm comm, MPI_Status *status)
```

OUT buf:	*Address of the receive buffer*
IN count:	*Size of receive buffer*
IN dtype:	*Data type*
IN source:	*Sender*
IN tag:	*Message tag*
IN comm:	*Communicator*
OUT status:	*Status information*

- Properties:
  - It is a blocking operation, i.e. it terminates after the message is available in the receive buffer.
  - The message must not be larger than the receive buffer.
  - The remaining part of the buffer not used for the received message will be unchanged.

Properties of MPI_Recv

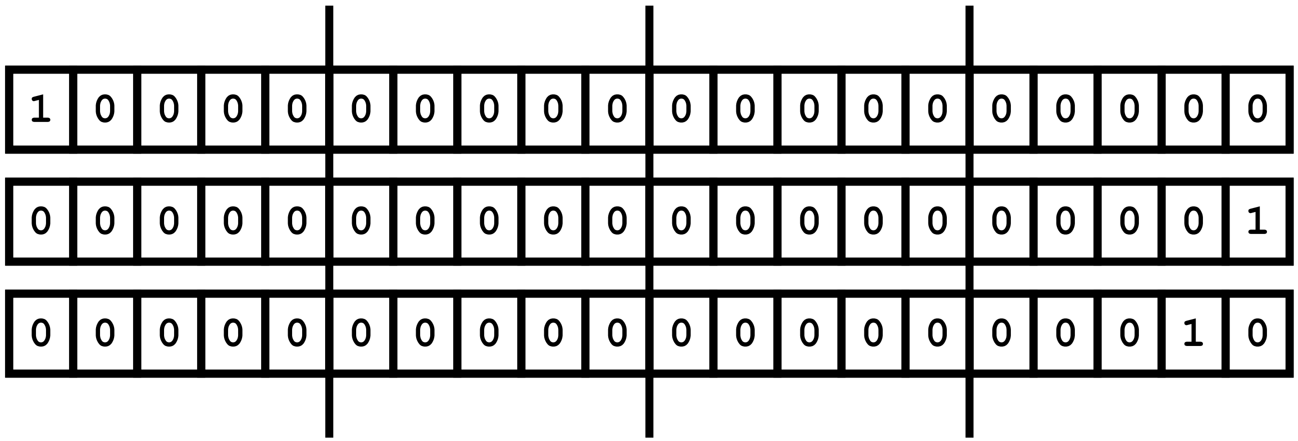
- Message selection
  - A message to be received by this function must match
    - the sender
    - the tag
    - the communicator
  - Sender and tag can be specified as wild cards
    - MPI_ANY_SOURCE and MPI_ANY_TAG
  - There is no wild card for the communicator.
- Status
  - The data structure MPI_Status includes
    - status(MPI_SOURCE): sender of the message
    - status(MPI_TAG): message tag
    - status(MPI_ERROR): error code
  - The actual length of the received message can be determined via MPI_Get_count.

Circular Left Shift Application

mpirun –np 4 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.



Shifts: Initialization

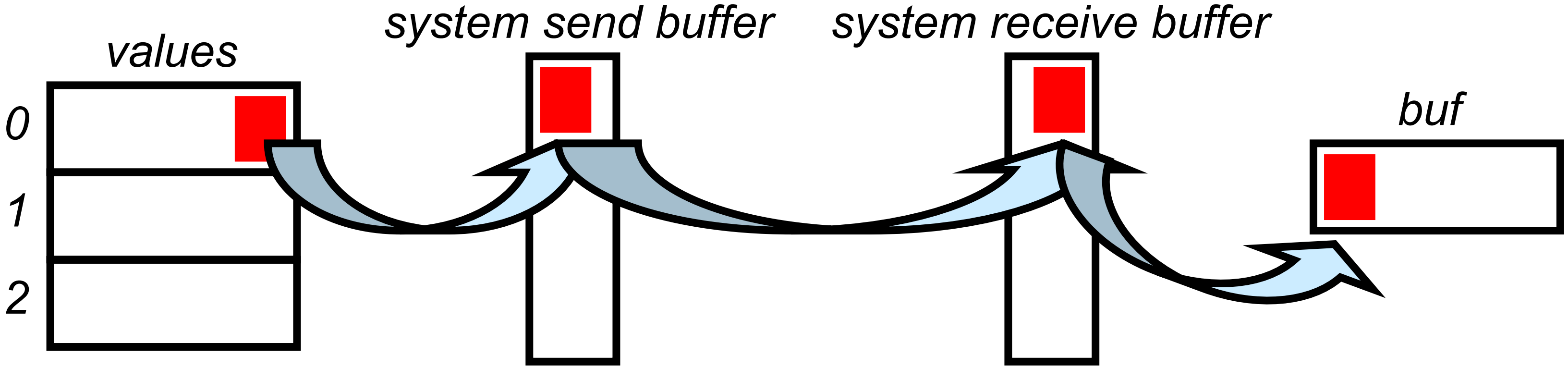
```
#include "mpi.h"

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);
```

Shifts: Definition of Neighbors

```
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;
}
```

Shifts: Shift the array

```
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;
}
```



MPI_Sendrecv

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

- Sendbuf and recvbuf have to be different
- Equivalent to the execution of MPI_Send and MPI_Receive in parallel threads.
- MPI_Sendrecv_replace:
  - Only one buffer.
  - The sent message is replaced by the received message.

Shifts: Shift the array

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

    for (j=1;j<100/np;j++){
      values[j-1]=values[j];
    }

    MPI_Sendrecv(&buf, 1, MPI_INT, lnbr, 10,
             &values[100/np-1], 1, MPI_INT, rnbr, 10
             MPI_COMM_WORLD, &status);
}
```

MPI_Wtime

```
double MPI_Wtime (void)
```

- Return value represents elapsed wall-clock time since some time in the past measured in seconds.
- The time returned is local to the node that called it.

Collective Operations

- Properties
  - Must be executed by all processes of the process group.
  - Must be executed in the same sequence.
  - All collective operations are blocking operations.
- MPI provides three classes of collective operations
  - Synchronization
    - Barrier
  - Communication
    - Broadcast
    - gather
    - scatter
  - Reduction
    - Global value returned to one or all processes.
    - Combination with subsequent scatter.
    - Parallel prefix operations

MPI_Barrier

int MPI_Barrier (MPI_Comm comm)

IN comm: *Communicator*

- This operation synchronizes all processes.

```
#include "mpi.h"

main (int argc,char *argv[]){
...
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);

MPI_Barrier(MPI_COMM_WORLD)

...
```

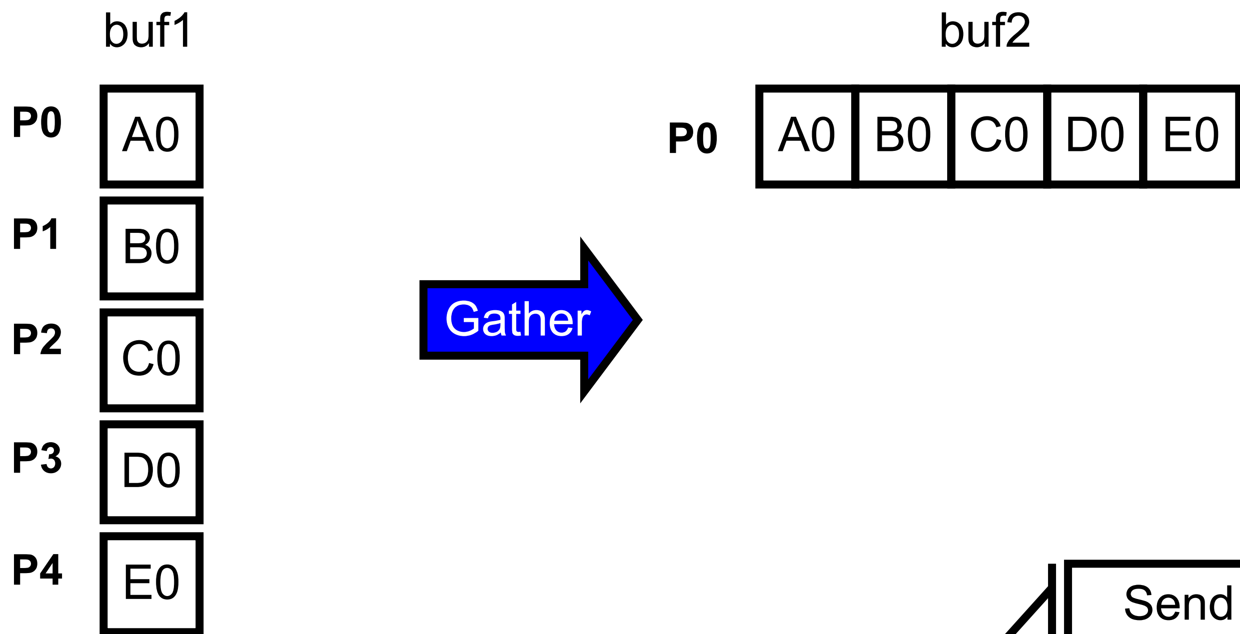
MPI_Bcast

```
int MPI_Bcast (void *buf, int count, MPI_Datatype dtype, int root,
               MPI_Comm comm)
```

IN buf:	*Adress of send/receive buffer*
IN count:	*Number of elements*
IN dtype:	*Data type*
IN root:	*Sender*
IN comm:	*Communicator*

- The contents of the send buffer is copied to all other processes.
- Collective vs P2P operations
  - Only blocking
  - No tag
  - Number of elements sent must be equal to number of elements received.
  - The routines do not necessarily synchronize processes.

Gather Operation



int MPI_Gather (buf1, 1, MPI_INT,
buf2, 1, MPI_INT,
P0,
MPI_COMM_WORLD)

Send Buffer

Receive Buffer

Root

MPI_Gather

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

IN sendbuf:	*Send buffer*
IN sendcount:	*Number of elements to be sent to the root*
IN sendtype:	*Data type*
OUT recvbuf:	*Receive buffer*
IN recvcount:	*Number of elements to be received from each process.*
IN recvtype:	*Data type*
IN root:	*Receiver*
IN comm	*Communicator*

- The root receives from all processes the data in the send buffer.
- It stores the data in the receive buffer ordered by the process number of the senders.

Reductions

- The data of the processes are combined via a specified operation, e.g. ’+’.
- There are three different variants
  - The result is only available at the root process.
  - The result is available at all processes.
  - The result is different for the processes according to a prefix operation.
- Input values at each process:
  - Scalar variable: The operations combines all variables of the processes.
  - Array: The elements of the arrays are combined in an elementwise fashion. The result is an array.

Example: Scalar 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
```

MPI_Reduce

```
int MPI_Reduce (void* sbuf, void* rbuf, int count, MPI_Datatype
                dtype,MPI_Op op, int root, MPI_Comm comm)
```

IN sbuf:	*Send buffer*
OUT rbuf:	*Receive buffer*
IN count:	*Number of elements*
IN dtype:	*Data type*
IN op:	*Operation*
IN root:	*Root process*
IN comm:	*Communicator*

- This operation combines the elements in the send buffer and delivers the result to root.
- Count, op, and root have to equal in all processes

Nonblocking Communication

- Properties
  - Nonblocking send and receive operations terminate after a communication request was created. (Posting of send or receive operation)
  - A separate test is required to ensure that the posted operation completed.
  - Only after the test it is safe to access the send and receive buffer.
- This allows:
  - Send: Overlap communication with local computation.
  - Receive: Copy a message directly into the address space of an application without blocking on the message.

MPI_Isend

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

IN buf:	*Send buffer*
IN count:	*Number of elements*
IN dtype:	*Data type*
IN dest:	*Receiver*
IN tag:	*Tag*
IN comm:	*Communicator*
OUT request:	*Reference to request*

- This operation terminates after a request was created.
- Access to send buffer is save only after the request has terminated.
- This is verified with MPI_Wait.

MPI_Irecv

int MPI_Irecv (void* buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Request* request)

IN buf:	*receive buffer*
IN count:	*number of entries in receive buffer*
IN dtype:	*data type*
IN dest:	*sender*
IN tag:	*tag*
IN comm	communicator
OUT request:	*reference to request*

MPI_Wait

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

        INOUT request:  request
        OUT status:     status of terminated operation
```

- This operation blocks until the request was executed.
- Other test operations:
  - MPI_Wait_any, MPI_Waitall, MPI_Waitsome
  - MPI_Test, MPI_Test_any, MPI_Test_some
  - MPI_Cancel

Example: Nonblocking Communication

```
int a[100], b[100]

MPI_Send(b[0], 1, MPI_INT,...)
MPI_Recv(b[50], 1, MPI_INT, ...)

for (i=0;i<50;i++){
 a[i]=a[i]+b[i+1]
}
```

The processes have to wait until:

- the send buffer is ready although it is not overwritten
- the value of the right neighbor was received although it is read only in the last iteration.

Example contd., Overlapping Communication

```
int a[100], b[100];
MPI_Request request;

MPI_Isend(b[0], 1, MPI_INT,..., &request)
MPI_Recv(b[50], 1, MPI_INT, ...)

for (i=0;i<50;i++){
 a[i]=a[i]+b[i+1]
}
MPI_Wait(&request, &status)
...
b[0]= ...
```

- The send operation need to terminate only before the assignment to *b*.

Process Topologies

- Topologies define an intuitive name space
- They allow an effective mapping of processes to a hardware topology
- MPI supports
  - multidimensional grids and tori
  - arbitrary graphs
- Virtual topology will be an attribute of a communication domain
- Processes can access the topology and the coordinates via the communicator.

Grid Topologies

- MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart)
  - comm_old : input communicator
  - ndims: number of dimensions
  - dims: array with lengths
  - periods: logical array specifying whether the grid is periodic
  - reorder: allow reordering of ranks in output communicator
  - comm_cart: output communicator

Example: Topologies

```
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);
```

Example: Topologies