Parallele Programmierung I

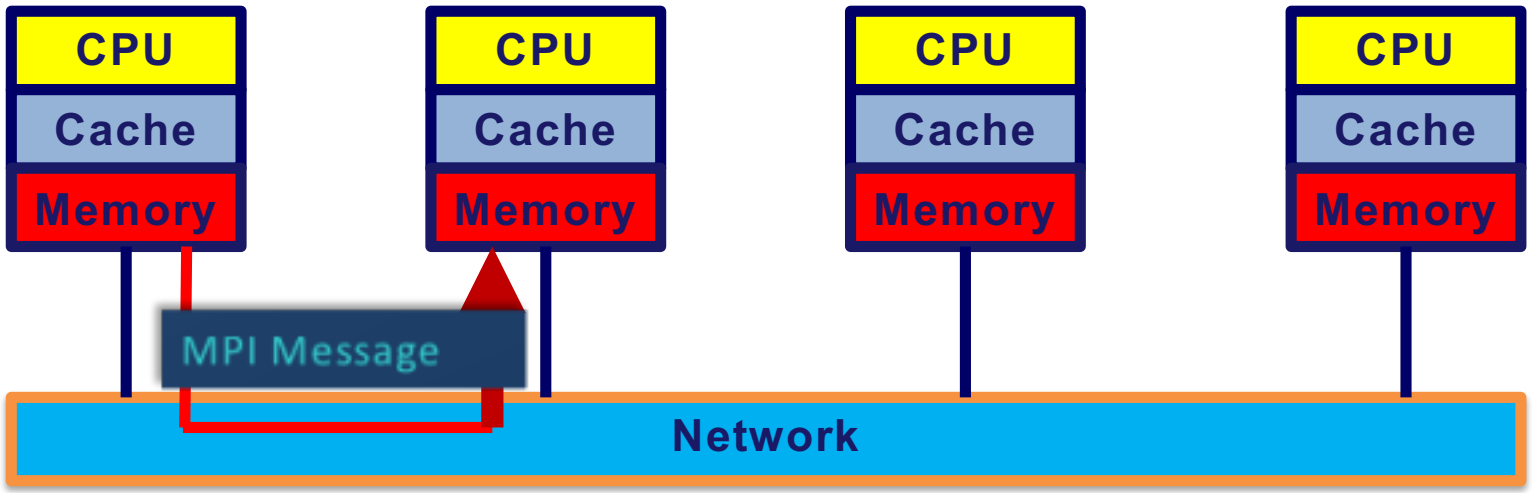
Prof. Dr. Volker Gülzow
Dr. Yves Kemp
Dr. Sergey Yakubov

SS 2018

We continue with MPI

MPI (recap from the previous lecture)

– Standardized interface to a communication library
– Hides hardware/software communication mechanisms from a user
– Implements internode communications via messages



MPI Forum and MPI Standards

- Beginning (1992-1994) – Introduction of the first message passing standard MPI 1.0

- MPI forum (www.mpi-forum.org) defines the standards
  - MPI-1 (1994) , MPI-1.3 (2008)
  - MPI-2 (1997), MPI2-2 (2009)
  - MPI-3 (2012), MPI-3.1 (2015), now working at 4.0

- Contributions from approx. 60 members
  - Academical organizations
  - HPC centres
  - Computer vendors

MPI Forum and MPI Standards

- Independent implementations
  - OpenMPI, MPICH
- Vendor‘s implementations
  - Cray, HP, Intel, Microsoft
- All implementations are portable
  - Architecture and hardware independent
  - Can also be used on shared memory machines
- Language support
  - Basic languages - C/C++ and Fortran
  - Bindings to Perl, Python, Java, etc.

Example

```
#include <stdio.h>
#include <mpi.h>
main (int argc, char* argv[]) {
        int myrank; int size;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        printf("Hello World. I'm process %d of %d  processes.",
myrank, size);
        MPI_Finalize();
        exit(0);
}
```

Compilation

- There are two ways to compile an MPI program
  - Using wrappers (mpicc, mpif77, mpif90)
    - $ mpicc prog.c
    - $ mpif90 prog.f90
  - Using regular compiler and setting compiler/linker flags
    - $ gcc prog.c –I<path to mpi.h> -L<path to libmpi> -lmpi
- Usually several MPI implementations/versions and several compilers are installed – one should be careful

Execution

- Depending on MPI version execution command may differ
  - Standard commands
    - $ mpirun –np 4 prog_name
    - $ mpiexec -n 4 prog_name
  - Other versions
    - $ poe -procs 4 prog_name # IBM
    - $ aprun -n 4 prog_name # Cray
- See man pages for more info
- Usually queuing system is used on HPC machines to submit parallel jobs (future lectures)

Example

```
#include <stdio.h>
#include <mpi.h>
main (int argc, char* argv[]) {
        int myrank, size;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        printf("Hello World. I'm process %d of %d  processes.",
myrank, size);
        MPI_Finalize();
        exit(0);
}
```

Try to compile and execute it,
use MPI_WTIME to measure execution time and print it only once

Basic functions - MPI_Send

```
int MPI_Send(void* buffer, // message sending buffer
             int count, // number of elements to send
             MPI_Datatype datatype, // element type
             int destination, // rank of destination process
             int tag, // message tag
             MPI_Comm comm); // communicator
```

- The function is blocking
  - Buffer may be reused after return
  - That doesn‘t mean that message was recieved!
  - May block until the message is received by the destination process – implementation/message size depending

Basic functions - MPI_Receive

```
MPI_Receive(void* buffer, // message receive buffer
            int count, // max. number of elements to receive
            MPI_Datatype datatype, // element type
            int source, // rank of source process
            int tag, // message tag
            MPI_Comm comm, // communicator
            MPI_Status* status); // receive status
```

- The function is blocking
  - Message has been successfully received
  - Buffer size can be larger than message size
  - Number of elements in the message can be less than count

Basic functions - MPI_Receive

MPI_Receive(buffer, count, datatype, source, tag, comm, status)

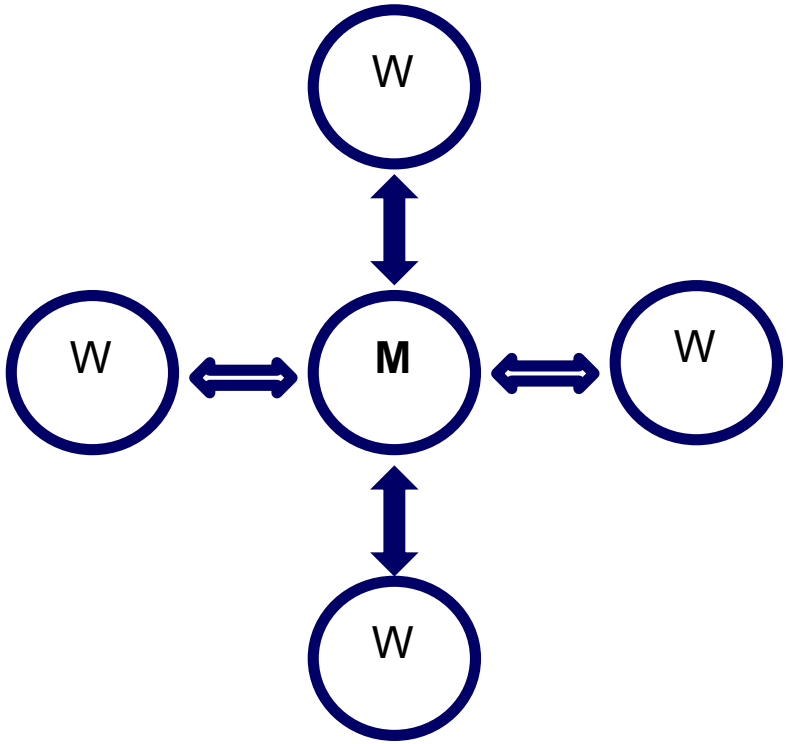
- Source can be MPI_ANY_SOURCE
- Tag can be MPI_ANY_TAG
- MPI_Status* status (integer status(MPI_STATUS_SIZE)) has info
  - status(MPI_SOURCE) – rank of source process
  - status(MPI_TAG) – tag of the message
- Number of received elements can be obtained via
  - MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int *count)

Blocking point-to-point communication mode

MPI routine	Return condition	Send status on return	Receive status on return
MPI_BSEND	Immediately	Unknown	Unknown
MPI_SSEND	Receive has started	Started	Started
MPI_SEND	Either MPI_SSEND or MPI_BSEND		
MPI_RECV	When message was obtained	Complete	Complete

Master-Worker Approach

- Classical approach to parallel programming
  - One process is a master
  - The other processes are workers
  - Master collects results from workers
- Uses only MPI_SEND and MPI_RECEIVE
- Point-to-point communication pattern



Parallel calculation of $\int_a^b f(x)dx$

```
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
a=0.d0 ; b=2.d0 ; res=0.d0 ; h = 10.d0;
mya = a + rank*(b-a)/size
myb = mya + (b-a)/size
psum = (myb-mya)*h
if(rank.eq.0) then
 res=psum
 do i=1,size-1
  call MPI_Recv(tmp,1,MPI_DOUBLE_PRECISION,i, 0,MPI_COMM_WORLD,status,ierror)
  res=res+tmp
 enddo
 write(*,*) 'Result: ',res
else
 call MPI_Send(psum,1,MPI_DOUBLE_PRECISION,0,0, MPI_COMM_WORLD,ierror)
endif
```

do it in C, measure execution time for different number of processes

Deadlock

```
#include <mpi.h>
#include <stdio.h>
#include "malloc.h"
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size = 100000;
    int* numbers = malloc(sizeof(int)*size);
    int to, from;
    int other = rank == 0?1:0;
    MPI_Send(numbers, size, MPI_INT, other, 0, MPI_COMM_WORLD);
    MPI_Recv(numbers, size, MPI_INT, other, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Finalize();
    printf("%d got %d from %d\n",rank,numbers[0],other);
    free(numbers);
    return 0;
}
```

Compile and run with two processes.
Try to make it work

Non-blocking Communications

- Motivation
  - Avoids deadlocks
  - Simplifies programming
  - Reduces syncronization
  - May allow to overlap communication and computation
- Requires additional request handle
  - Created for each nonblocking communication call
  - Used to check the communication state (wait/test operation)

MPI_Isend

```
int MPI_Isend(void* buffer, // message sending buffer
              int count, // number of elements to send
              MPI_Datatype datatype, // element type
              int destination, // rank of destination process
              int tag, // message tag
              MPI_Comm comm, // communicator
              MPI_Request *request);    // request handle
```

- The function is non-blocking
  - Buffer cannot be reused after return
  - Request handle is unique for each operation
  - Test/wait operation must follow the non-blocking call

MPI_Receive

```
MPI_Receive(void* buffer, // message receive buffer
            int count, // max. number of elements to receive
            MPI_Datatype datatype, // element type
            int source, // rank of source process
            int tag, // message tag
            MPI_Comm comm, // communicator
            MPI_Request* request); // request handle
```

- The function is non-blocking
  - Message has not been received yet
  - No status argument
  - Test/wait operation must follow (status will be provided there)

MPI_Wait, Waitall

```
int MPI_Waitall(int count,                    // arrays length
                MPI_Request requests[],  // array of request handles
                MPI_Status *statuses);     //  array of status objects
```

- Blocks until all communication operations associated with active handles in the list complete
- Returns the statuses of all these operations

Deadlock

```
#include <mpi.h>
#include <stdio.h>
#include "malloc.h"
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size = 100000;
    int* numbers = malloc(sizeof(int)*size);
    int to, from;
    int other = rank == 0?1:0;
    numbers[0]=rank;
    MPI_Send(numbers, size, MPI_INT, other, 0, MPI_COMM_WORLD);
    MPI_Recv(numbers, size, MPI_INT, other, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Finalize();
    printf("%d got %d from %d\n",rank,numbers[0],other);
    free(numbers);
    return 0;
}
```

Rewrite with non-blocking send/receive calls

Collective Communications

- Three types:
  - Synchronization (Barrier)
  - Data Movement (Scatter, Gather, Alltoall, Allgather)
  - Reductions (Reduce, Allreduce, Reduce_scatter)
- Usually blocking (MPI 3.0+ allows non-blocking)
- Receive buffers size must be exactly the size of the message
- Can be implemented with point-to-point calls
- Collective implementation is usually better optimized (tree-based algorithms, etc)

MPI_Barrier

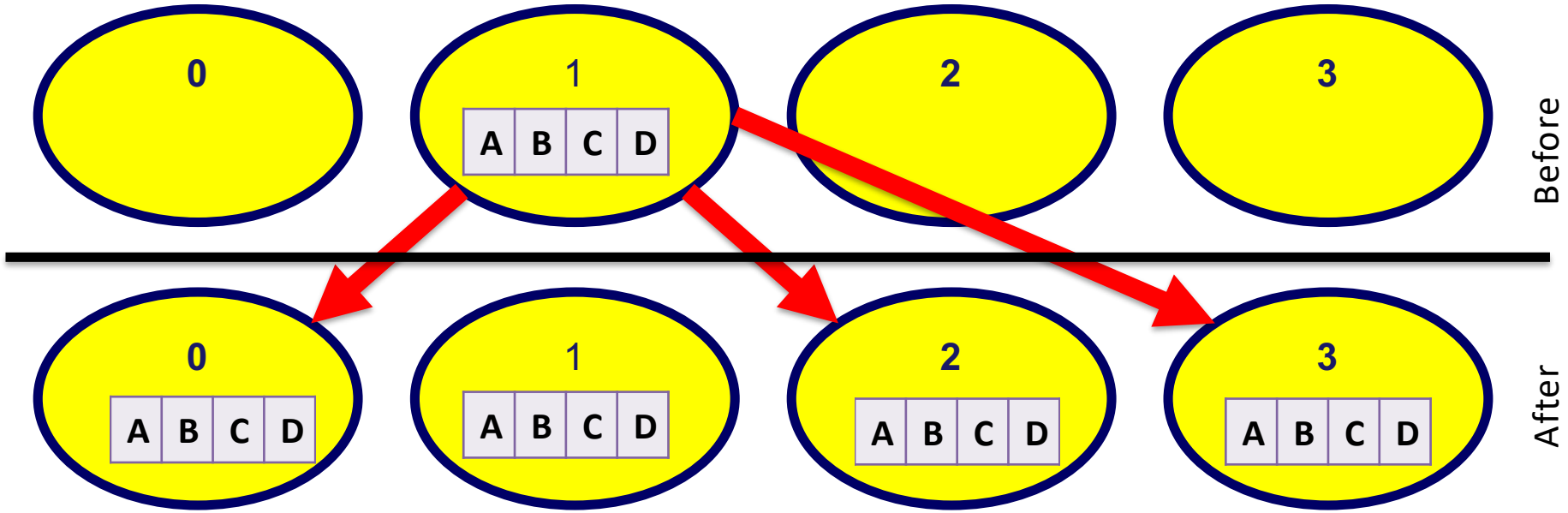
int **MPI_Barrier**(MPI_Comm comm);

- Explicit syncronization
- Block the process until all processes in the communicator called it
- Usually not needed
  - Syncronization is done implicitly by other communication calls
  - Can be used for debugging, profiling, etc.

MPI_Bcast

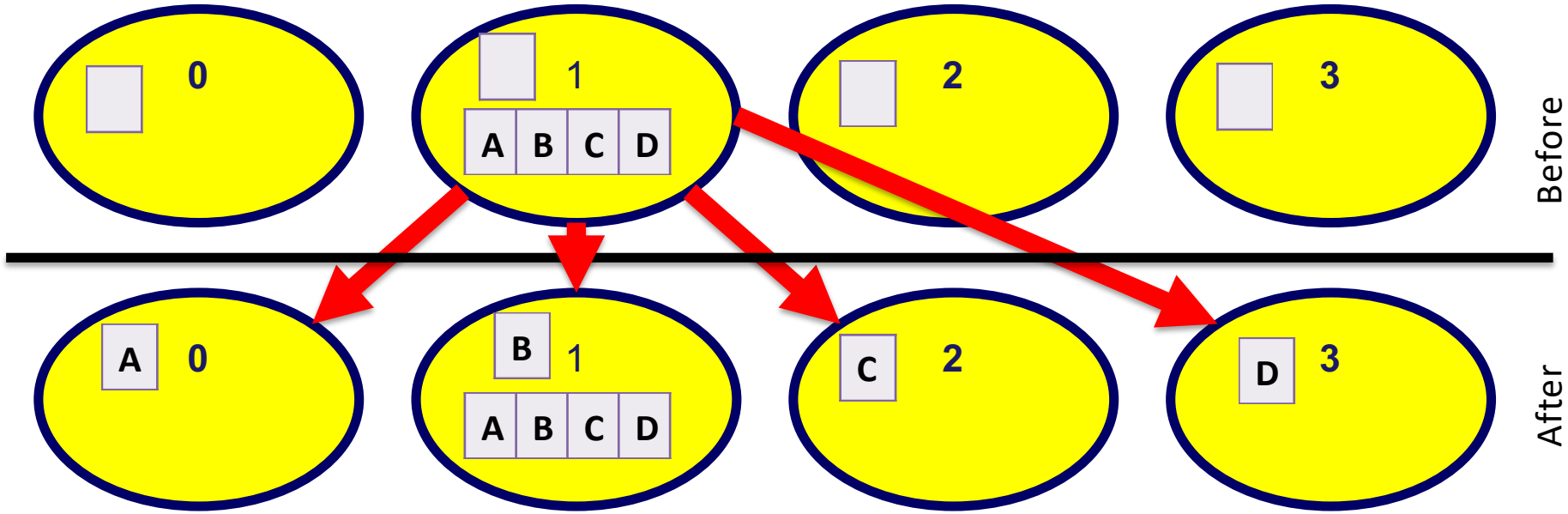
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

- One process (root) sends data to all others



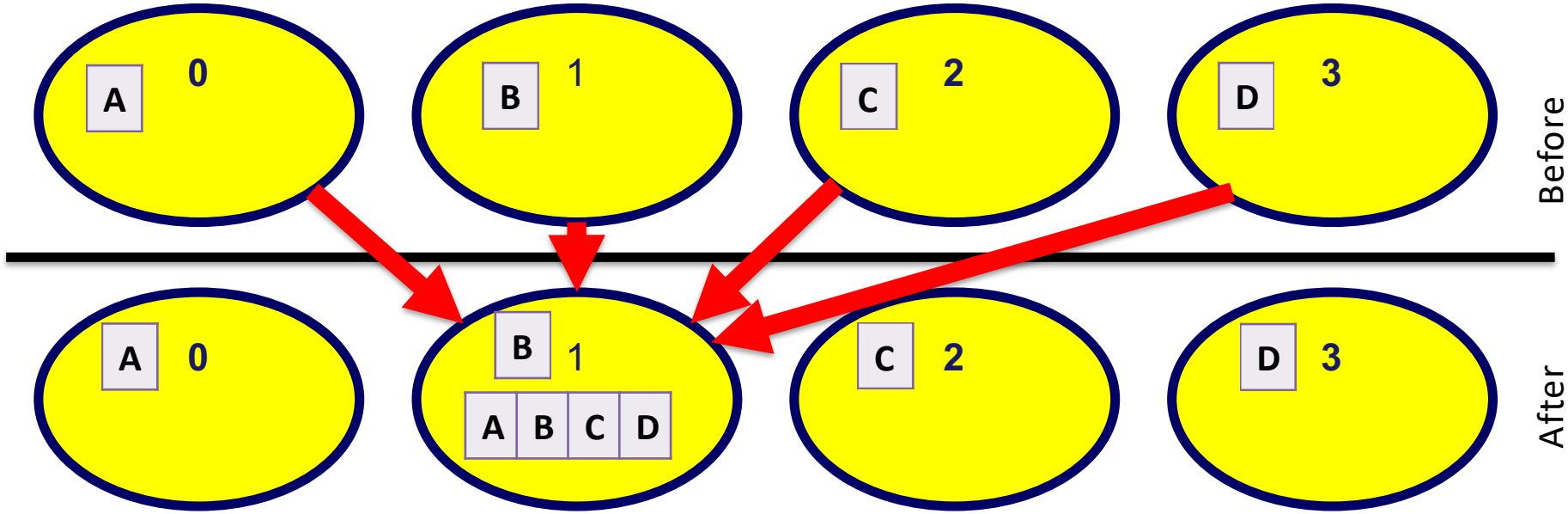
MPI_Scatter

- One process scatters data to all others (including itself)



MPI_Gather

- One process gathers data from all others (including itself)



MPI_reduce

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

- Reduces values on all processes to a single value
- Can work with arrays (count >1)
- MPI_Op - reduce operation
  - standard (MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD, …)
  - used defined (MPI_Op_create, MPI_Op_free)
- All operations are assumed to be associative
  - different results with floating point datatypes are possible
- MPI_Allreduce = MPI_Reduce+MPI_BCast

Parallel calculation of $\int_a^b f(x)dx$

```
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
a=0.d0 ; b=2.d0 ; res=0.d0 ; h = 10.d0;
mya = a + rank*(b-a)/size
myb = mya + (b-a)/size
psum = (myb-mya)*h
if(rank.eq.0) then
 res=psum
 do i=1,size-1
  call MPI_Recv(tmp,1,MPI_DOUBLE_PRECISION,i, 0,MPI_COMM_WORLD,status,ierror)
  res=res+tmp
 enddo
 write(*,*) 'Result: ',res
else
 call MPI_Send(psum,1,MPI_DOUBLE_PRECISION,0,0, MPI_COMM_WORLD,ierror)
endif
```

do it with MPI_reduce

Summary

	Point-to-Point	Collective
Blocking	MPI_SEND MPI_SSEND MPI_BSEND MPI_RECV MPI_SENDRECV MPI_WAIT/MPI_WAITALL MPI_WAITANY/MPI_WAITSOME	MPI_BARRIER MPI_BCAST MPI_SCATTER MPI_GATHER MPI_ALLTOALL MPI_REDUCE
Non-blocking	MPI_ISEND MPI_IBSEND MPI_ISSEND MPI_IRECV MPI-TEST MPI_TEST/MPI_TESTALL MPI_TESTANY/MPI_TESTSOME	MPI_IREDUCE, etc. MPI 3.0 standard only