Concepts and Models of Parallel and Data-centric Programming

Message-Passing

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

i12 High Performance Computing | RWTH AACHEN UNIVERSITY

Outline

0. Organization
1. Foundations
2. Shared Memory
3. GPU Programming
4. Bulk-Synchronous Parallelism
5. **Message Passing**
6. Distributed Shared Memory
7. Parallel Algorithms
8. Parallel I/O
9. MapReduce
10. Apache Spark

a. MPI Foundations
b. Point-to-Point Communication
c. Collective Operations

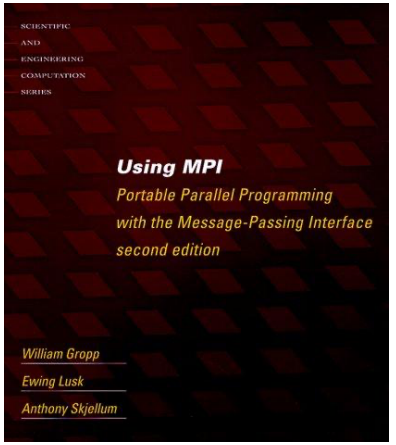
MPI

- Message Passing Interface
  - The de-facto standard API for explicit message passing nowadays
  - A moderately large standard (v3.1 is a 868 pages long)
  - Maintained by the non-profit Message Passing Interface Forum http://www.mpi-forum.org/
- Many concrete implementations of the MPI standard
  - Open MPI, MPICH, Intel MPI, MVAPICH, MS-MPI, etc.
- MPI is used to express the explicit interaction (communication) in programs for computers with distributed memory
- MPI provides source level portability of parallel applications between different implementations and hardware platforms

Literature about MPI

- Using MPI

  by William Gropp, Ewing Lusk, Anthony Skjellum
  The MIT Press, Cambridge/London, 1999

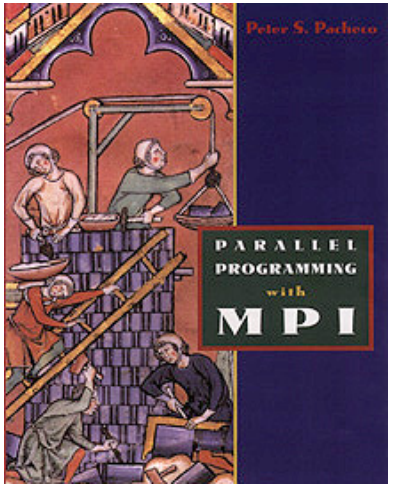


- Using MPI-2

  by William Gropp, Ewing Lusk, Rajeev Thakur
  The MIT Press, Cambridge/London, 2000

- Parallel Programming with MPI

  by Peter Pacheco
  Morgan Kaufmann Publishers, 1996



MPI Foundations

SPMD – Data Exchange

- Serial program

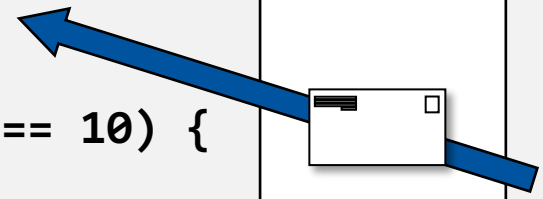
```
a[0..9] = a[100..109];
```

- SPMD program
  - process 0: **aa** holds **a[0..9]**

```
array aa[10];

if (my_id == 0) {
  recv(aa, 10);
}
else if (my_id == 10) {
  send(aa, 0);
}
```

process 10: **aa** holds **a[100..109]**

```
array aa[10];

if (my_id == 0) {
  recv(aa, 10);
}
else if (my_id == 10) {
  send(aa, 0);
}
```



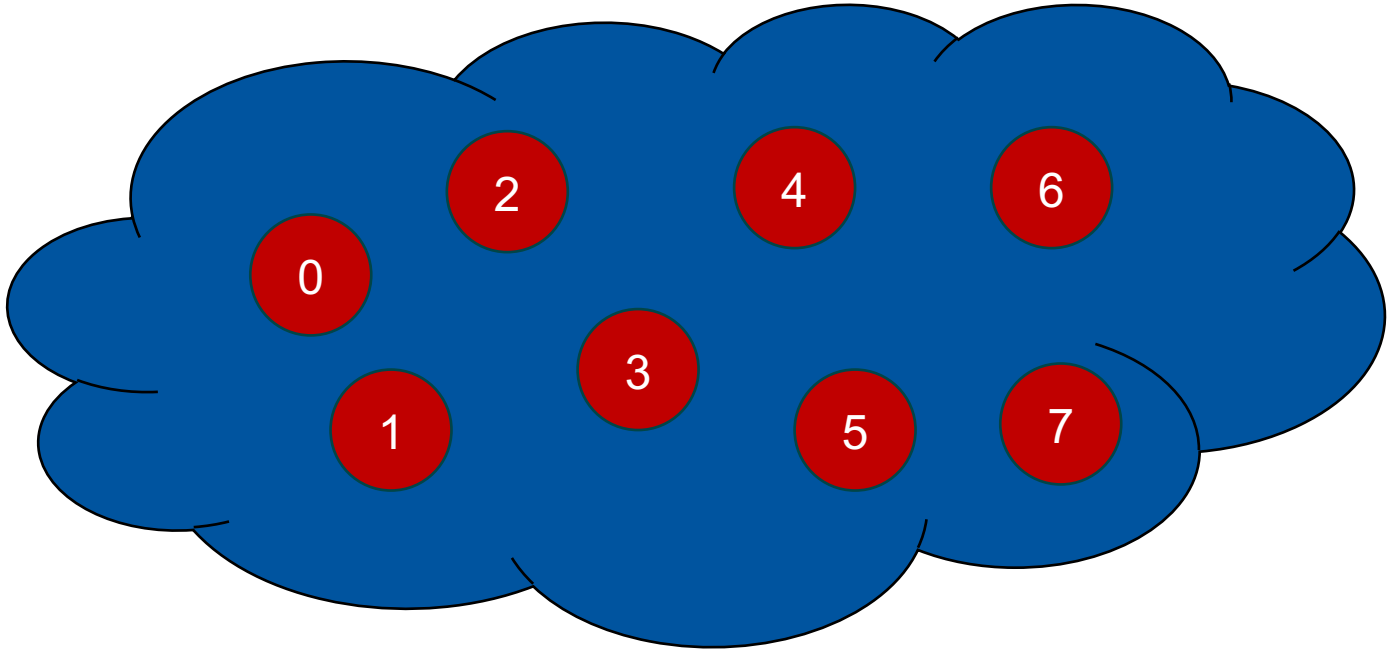
Hello, MPI!

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, nprocs;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  printf("Hello, MPI! I am %d of %d\n",
         rank, nprocs);

  MPI_Finalize();
  return 0;
}
```

1. Header file inclusion – makes available prototypes of all MPI functions
2. MPI library initialisation – must be called before other MPI operations are called
3. MPI operations – more on that later
4. Text output – MPI programs also can print to the standard output
5. MPI library clean-up – no other MPI calls after this one allowed

Communicators

- Each MPI communication happens within a context called communicator
  - Logical communication domains
  - A group of participating processes + context
  - Each MPI process has a unique numeric ID in the group – rank



Query operations on communicators

- How many processes are there in a given communicator?

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

  - Returns the total number of MPI processes when called on **MPI_COMM_WORLD**
  - Returns 1 when called on **MPI_COMM_SELF**

- What is the rank of the calling process in a given communicator?

```
MPI_Comm_rank (MPI_Comm comm, int *rank)
```

  - Returned rank will differ in each calling process given the same communicator
  - Ranks values are in **[0, size-1]** (always **0** for **MPI_COMM_SELF**)

Point-to-Point Communication

Messages

- MPI passes data around in the form of messages
- Two components
  - Message content (user data)
  - Envelope

Field	Meaning
Sender rank	Who sent the message
Receiver rank	To whom the message is addressed to
Tag	Additional message identifier
Communicator	Communication context

- MPI retains the logical order in which messages between any two ranks are sent (FIFO)
  - But the receiver has the option to peek further down the queue

Sending messages

- Messages are sent using the MPI_SEND family of operations

```
MPI_Send (buf, count, datatype, dest, tag, comm)
```

message content envelope

Parameter	Meaning
buf	Location of data in memory
count	Number of consecutive data elements to send
datatype	MPI data type handle
dest	Rank of the receiver
tag	Message tag
comm	Communicator handle

MPI data types

- MPI has a type system which tells it how to access memory content while constructing and deconstructing messages
- Complex data types can be created by combining simpler types
- Predefined MPI data type handles – C

MPI data type handle	C data type
MPI_SHORT	short
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	char
…	…
MPI_BYTE	-

MPI data types

- MPI is a library – it cannot infer the supplied buffer elements' type in runtime and hence has to be told what the type is

- MPI supports heterogeneous environments and implementations can convert between internal type representation on different architectures
  - **`MPI_BYTE`** is used to send/receive data as-is without any conversion

- MPI data type must match the language type of the data in the array

- Underlying data types must match in both communicating processes

Receiving messages

- Messages are received using the MPI_RECV operation

```
MPI_Recv (buf, count, datatype, src, tag, comm, status)
```

message content envelope

Parameter	Meaning
buf	Location in memory where to place data
count	Number of consecutive data elements that **buf** can hold
datatype	MPI data type handle
src	Rank of the sender
tag	Message tag
comm	Communicator handle
status	Status of the receive operation

Receiving messages

- The next message with matching envelope is received
  - Wildcard specifiers possible
    - **`MPI_ANY_SOURCE`** – matches messages from any rank
    - **`MPI_ANY_TAG`** – matches messages with any tag
  - Examine **`status`** to find out the actual values matched by the wildcard(s)

- Status argument – C
  - Structure of type **`MPI_Status`** with the following fields
    - **`status.MPI_SOURCE`** – source rank
    - **`status.MPI_TAG`** – message tag
    - **`status.MPI_ERROR`** – error code of the receive operation

Receiving messages

- Inquiry about the number of elements received

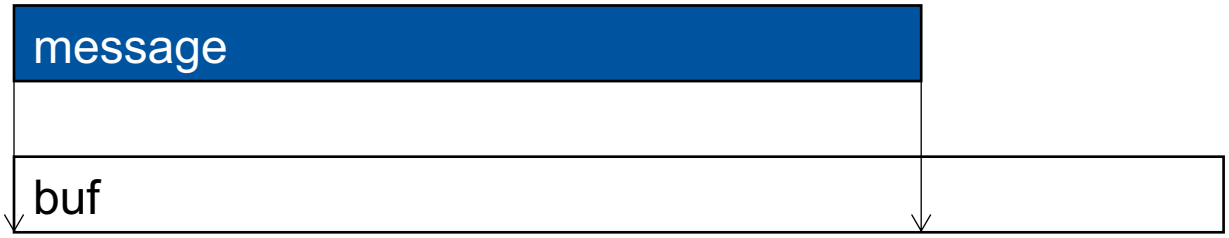
```
MPI_Get_count (status, datatype, count)
```

  - **count** is set to number of elements of type **datatype** that can be formed by the content of the message or to **MPI_UNDEFINED** if the number of elements is not integral
  - **datatype** should match the **datatype** argument of **MPI_RECV**

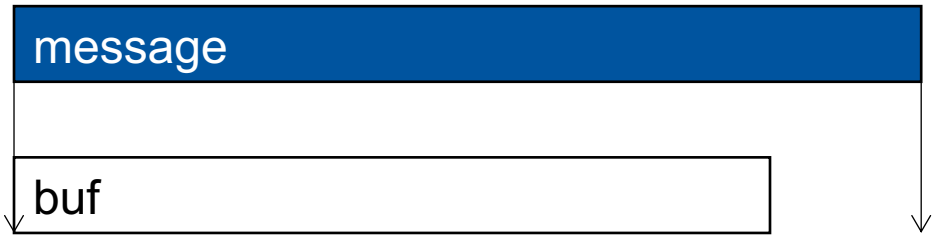
- If the receive status is of no interest – MPI_STATUS_IGNORE

Receiving messages

- buf/count must be large enough to hold the received message
  - OK



  - Not OK – truncation error in **`MPI_RECV`**

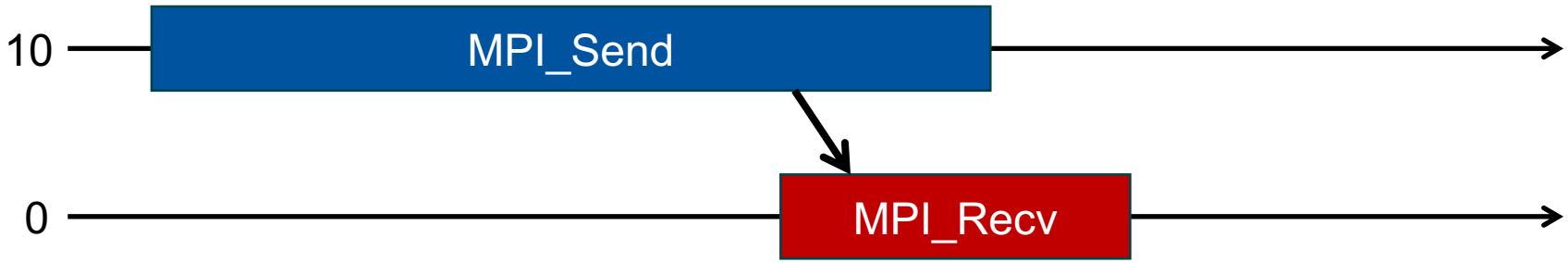


- Probe for matching message before receiving it
  - **`MPI_Probe(src, tag, comm, status)`**

Example

- Back to our earlier SPMD example

```
int aa[10];
MPI_Status status;

if (rank == 0) {
  MPI_Recv(aa, 10, MPI_INT, 10, 0, MPI_COMM_WORLD, &status);
}
else if (rank == 10) {
  MPI_Send(aa, 10, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```



Collective Operations

SPMD – Data Exchange

- Common data exchange operations
  - **`send(data, dst)`** – sends *data* to another process with ID of *dst*
  - **`recv(data, src)`** – receives *data* from another process with ID of *src*
    - wildcard sources usually possible, i.e. receive from any process
  - **`bcast(data, root)`** – broadcasts *data* from *root* to all other processes
  - **`scatter(data, subdata, root)`** – distributes *data* from *root* into *subdata* in all processes
  - **`gather(subdata, data, root)`** – gathers *subdata* from all processes into *data* in process *root*
  - **`reduce(data, res, op, root)`** – computes *op* over *data* from all processes and place the result in *res* in process *root*

Collective communication operations

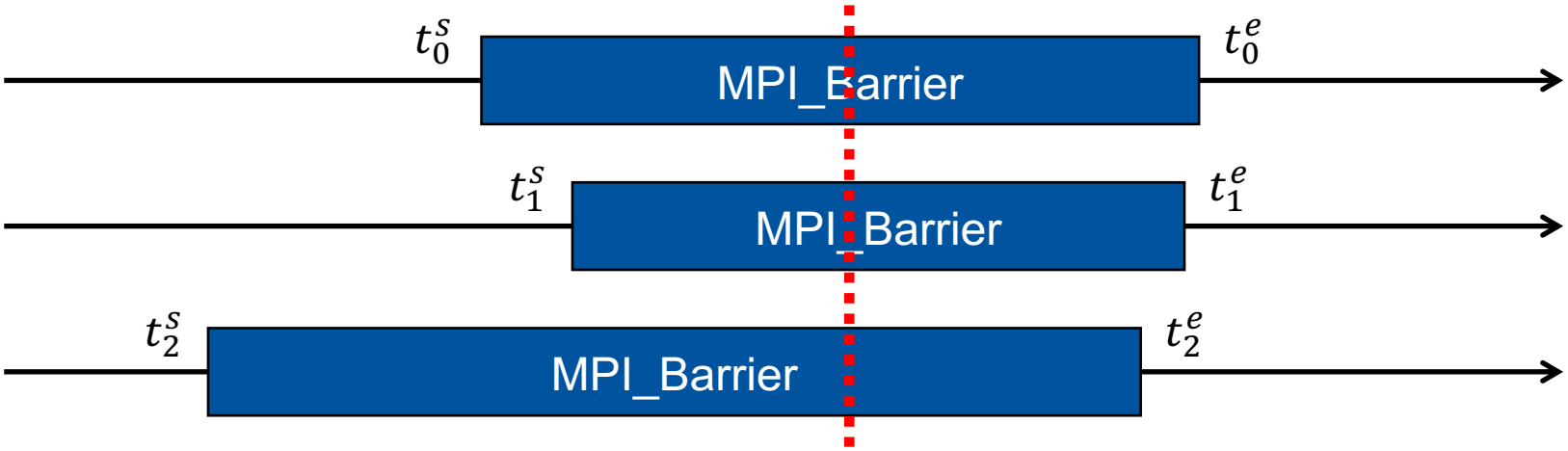
- An MPI collective communication operation is one which involves all processes in a given communicator
- All processes must call the same MPI function and provide the same value for the “root” process rank (if required by the operation)
- Collective operations can be globally synchronous
- The scope of a collective operation is a single communicator
- Any collective operation can be replaced with a set of point-to-point communication operations (although generally not recommended)

because the modern network architecture can exploit the topology to be more clever than calling a set of P2P communication operations

Barrier synchronisation

- Block until all processes in `comm` have entered the barrier

```
MPI_Barrier (comm)
```



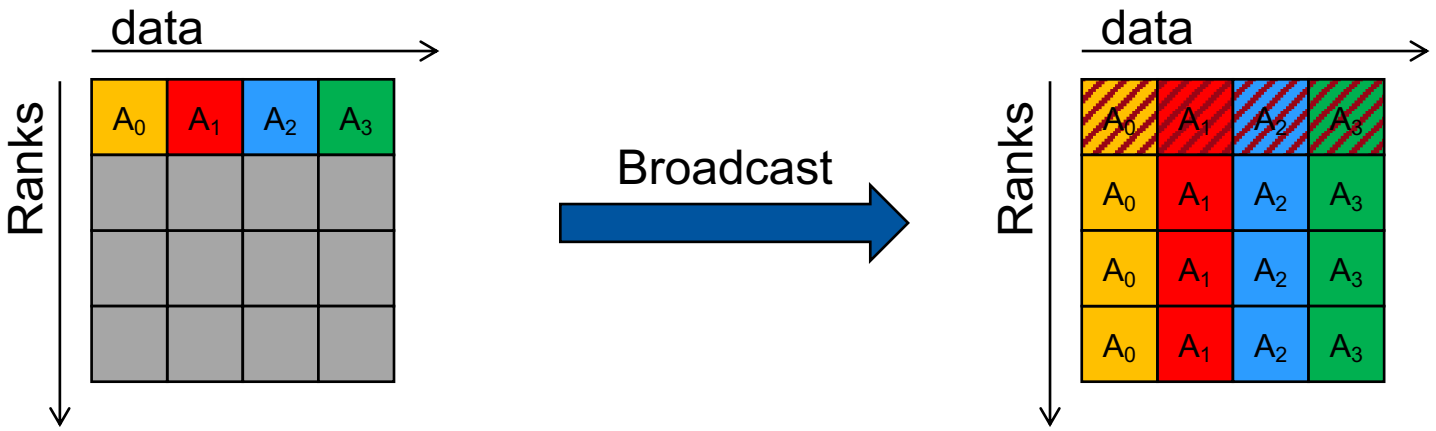
  - **`MPI_BARRIER`** guarantees that $\min_i t_i^e \geq \max_i t_i^s$
    - Processes are allowed to exit the barrier at different times, but not before all other processes have entered it
    - The only collective that is ***guaranteed synchronous***

Data replication – broadcast

- Replicate data from root process to all other processes in comm

```
MPI_Bcast (data, count, datatype, root, comm)
```

  - **data** points to the data source in process with rank **root**
  - **data** points to the destination buffer in all other processes
  - the amount of data sent must be equal to the size of the receive buffer



Data replication – broadcast

- Replicate data from root process to all other processes in comm

```
MPI_Bcast (data, count, datatype, root, comm)
```

  - **data** points to the data source in process with rank **root**
  - **data** points to the destination buffer in all other processes
  - example use:

```
int ival;

if (rank == 0)
   ival = read_from_somewhere();

MPI_Bcast(&ival, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Same value provided by all processes!

Data replication – broadcast

- Straightforward implementation of MPI_BCAST

```
void Bcast(void *data, int count, MPI_Type datatype,
           int root, MPI_Comm comm)
{
  int i, rank, nprocs;
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &nprocs);
  if (rank == root) {
    for (i = 0; i < nprocs; i++)
      if (i != root)
        MPI_Send(data, count, datatype, i, 999, comm);
  }
  else
    MPI_Recv(data, count, datatype, root, 999, comm,
             MPI_STATUS_IGNORE);
}
```

Global reduction

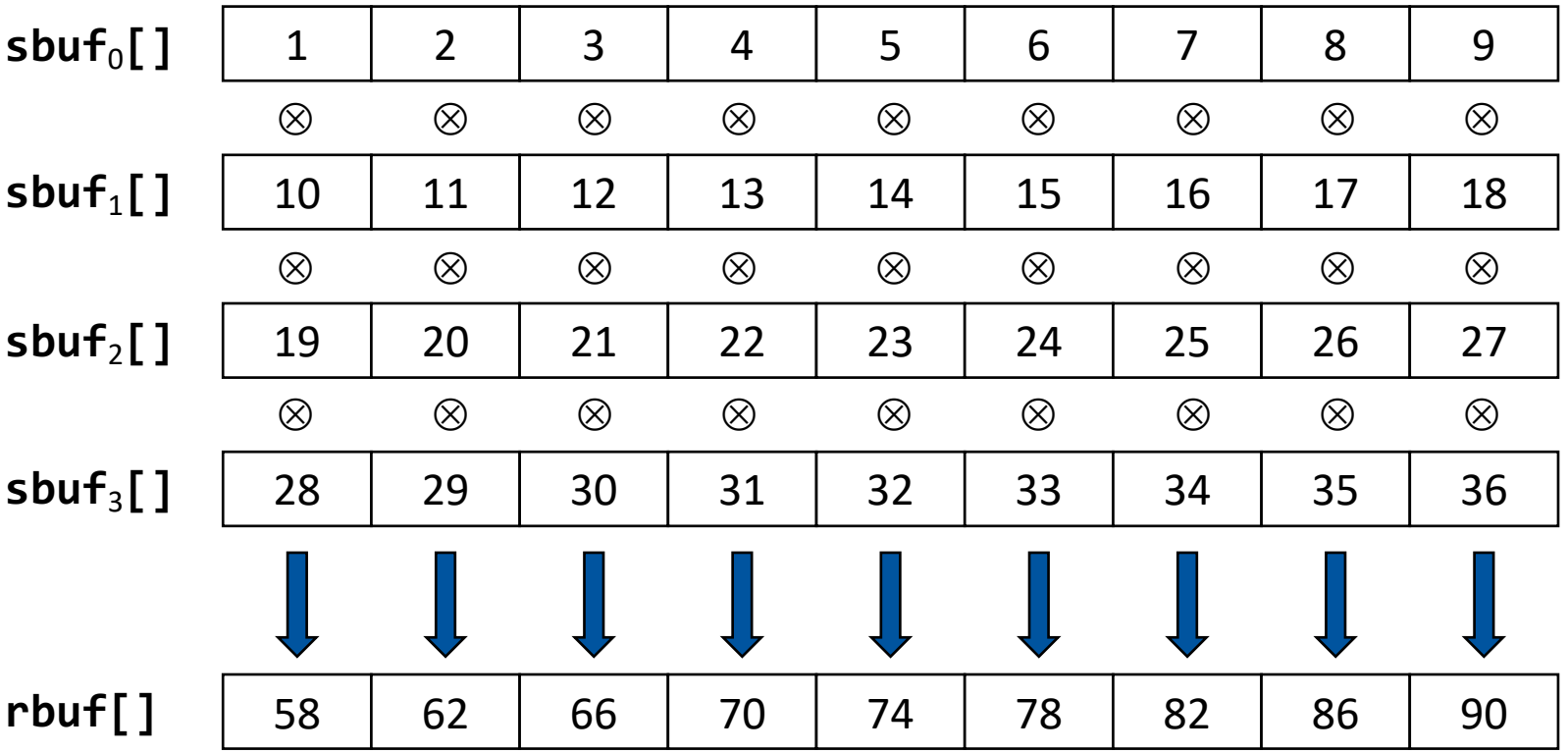
- Combine elements from all processes using a reduction operation

```
MPI_Reduce (sbuf, rbuf, count, dtype, op, root, comm)
```

Argument	Meaning
sbuf	Data source
rbuf	Receive buffer (significant only at **root**), different from **sbuf**
count	Number of elements in both buffers (same everywhere!)
dtype	Data type (same everywhere!)
op	Reduction operation
root	Rank of the data receiver
comm	Communicator handle

Global reduction

- Global element-wise operation
  - **rbuf[i] = sbuf$_0$[i] op sbuf$_1$[i] op sbuf$_2$[i] op … sbuf$_{nranks-1}$[i]**



⊗ = MPI_SUM

Global reduction – predefined operations

- MPI provides the following predefined reduce operations

Name	Meaning
`MPI_MAX`	maximum
`MPI_MIN`	minimum
`MPI_SUM`	sum
`MPI_PROD`	product
`MPI_LAND / MPI_BAND`	logical / bit-wise AND
`MPI_LOR / MPI_BOR`	logical / bit-wise OR
`MPI_LXOR / MPI_BXOR`	logical / bit-wise XOR
`MPI_MAXLOC`	max value and location
`MPI_MINLOC`	min value and location

  - All predefined operations are **associative** and **commutative**
  - Watch out for non-associativity of floating-point arithmetic!