Karlsruhe Institute of Technology

Programmierparadigmen

Prof. Dr. Ralf Reussner

Topic 5.3

Message Passing Interface (MPI)

DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY, FACULTY OF INFORMATICS

dsis.kastel.kit.edu


SDQ
Software Design and Quality
Winter Day 2025
Glühwein &
alkoholfreier
Punsch
Mon 15.12. 15:45-18:30
Infobau vor Raum 237
Kekse &
Waffeln
Firmen- &
Forschungskontakte
Jobs und Abschlussarbeiten
(in Deutsch & Englisch)
KIT
Karlsruher Institut für Technologie
FZI
andrena
OBJECTS

Overview on Today's Lecture

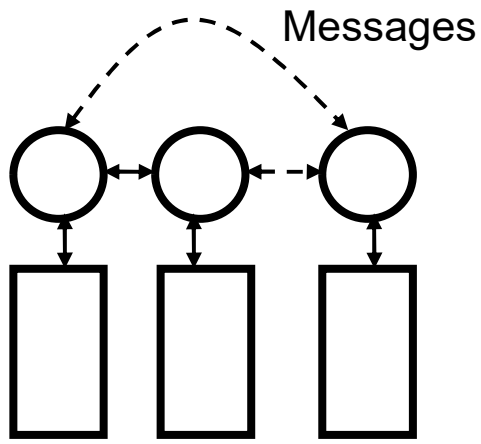
- Content
  - MPI introduction and basics
  - Message exchange with MPI
  - Collective operations in MPI
  - MapReduce

MPI Exercise Sheet now online (Deadline 19.12.)

- Learning goals: participants –
  - acquire an overview of the quasi standard in message-based parallel computing
  - understand the principle of message passing
  - are aware of the effects of collective operations
  - are able to apply MPI in simple settings
- Style
  - MPI functions, types, constants, etc. are colored blue, e.g. `MPI_Send`

Why Message Passing Interface?

- Processes have separate memory
  - They may only communicate via messages
- Proprietary message passing libraries existed long before MPI
  - They suffered from various issues
    - Code portability; vendor lock-in; flexibility…
  - MPI aimed at collecting "best practices“
    - Avoid shortcomings
    - Support for threads not required but not prohibited
- MPI offers point-to-point and broadcast communication
  - Synchronous and asynchronous messaging
  - Various, partially unusual communication primitives
- It only specifies "what", not "how"
  - i.e. it provides a “reliable middleware“, so that developer can focus on an application's functionality



Message passing

Message Passing Interface (MPI)

- MPI allows communication between processes via messages (also across computers)
- MPI has been under development since 1992
  - Has become a mature standard
  - Implementations are available for numerous platforms
  - Portable and widely used
  - APIs for C/C++/Fortran90
- History
  - MPI 1.0 released 1994
  - MPI 2.0 released 1997: Adds parallel I/O, access to memory of other processes, …
  - MPI 3.0 released 2012: Adds non-blocking collective operations, improved communication modes, …
  - MPI 4.0 released 2021: Adds large-count functions, persistent collectives, …
- We will focus on features that are available since MPI 1.0 and rely on the OpenMPI implementation for C

MPI Programming Model

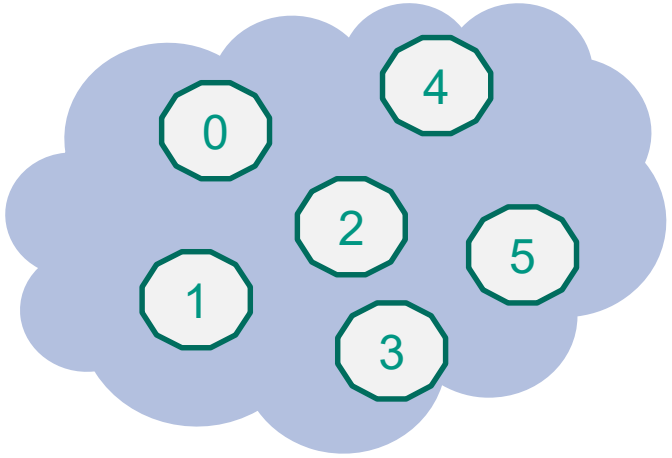
- C programs using MPI can be complied with the `mpicc` wrapper compiler: `mpicc INPUTFILES`
  - Internally uses the C compiler selected during configuration (e.g. `gcc`)
- Basically MPI follows the SIMD (single instruction, multiple data) model
  - The same program is started on *N* nodes:

    `mpirun -np N PROGRAM ARGUMENTS`
  - The program is loaded to each node and started there

➔ But you rather might want to say “single program, multiple data”

MPI Communicators

- Processes communicate via so-called communicators
  - A group of processes that can communicate with each other
  - `MPI_COMM_WORLD` is the default communicator, i.e. the collection of all processes
  - Needs to be provided to almost every MPI command as a parameter
- Within a communicator with *N* processes, each process *j* is identified by its individual rank R_j ($0 \leq R_j < N$)



MPI_COMM_WORLD

[http://www.uio.no/studier/emner/matnat/ifi/INF3380/v10/undervisningsmateriale/inf3380-week06.pdf]

MPI Process Ranks

- With `MPI_Comm_rank` a program can retrieve the number of the processing node (rank) it is running on within a communicator
- Depending on the rank the control-flow can branch
  - ➔ MIMD: Multiple instruction, multiple data becomes also possible
- With `MPI_Comm_size` the total number of processes in a communicator can be determined

```
int size, my_rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank == 0) {
    …
}
```

- Both methods return an integer, indicating the success of the operation
  - For that reason, size and rank are returned using the passed pointer
  - Here, we assume that it was successful and do not check the return value

MPI Communication

- MPI communication is built upon send and receive operations
  - A process may send a message to another providing its rank and a tag
  - The receiver can post a receive for a message with the tag
  - Called point-to-point communication
- In many cases, processes need to communicate with all other processes rather than a single one
  - Writing send and receive operations would be cumbersome and not use the network in an optimal manner
  - MPI provides collective operations for that purpose, e.g. for …
    - Broadcasting data to all processes
    - Scattering and gathering data

Hello World

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

int main(int argc, char** args) {
   int size;
   int myrank;

   MPI_Init(&argc, &args);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

   printf("Hello world, I have rank %d out of %d.\n", myrank, size);

   MPI_Finalize();

   return 0;
}
```

Import MPI commands

Initialize MPI

Clean up afterwards

Hello World

- Putting the program in a `hello.c` file, we can …
  - Compile it: `mpicc hello.c`
  - Execute it, e.g. in 3 processes: `mpirun -np 3 a.out`
  - Yielding the output (only one of 6 possibilities):

```
Hello World, I have rank 0 out of 3.
Hello World, I have rank 2 out of 3.
Hello World, I have rank 1 out of 3.
```

  ➔ **To stress it again, the same program is automatically executed in all processes**

➔ There is no implicit “master activity”

  - However, as a convention process 0 is typically used as “master control program”, known as root in MPI, so that MIMD becomes also possible

Synchronization

- `MPI_Barrier` blocks until all processes have called it

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

  - It is similar to Thread.join() in Java
  - It makes sure that all processes have reached a certain point

- Replace the simple `printf` in the Hello World program with the following code:

```
int i;
for (i = 0; i < size; i++) {
  MPI_Barrier(MPI_COMM_WORLD);
  if (i == myrank) {
    printf("Hello World, I have rank %d out of %d.\n", myrank, size);
  }
}
```

- What does the code do?

Message Exchange (1)

- Two important MPI communication primitives are …
  - `MPI_Send` and `MPI_Recv`
    - both are *blocking* and *asynchronous*
      - i.e. no synchronous sending/receiving necessary
    - `MPI_Send` blocks until the message buffer can be reused
    - `MPI_Recv` blocks until the message is received in the buffer *completely*

```
int MPI_Send( void* buffer, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
```

  - `buffer`: the initial memory address of the sender's buffer
    - C/C++ uses **`void`*** for arguments with a “free choice” type
  - `count`: number of elements that will be send
  - `datatype`: type of buffer's elements
  - `dest`: rank of the destination process
  - `tag`: "context" of the message (e.g. a conversion ID)
  - `comm`: communicator of the process group

Type of data needs to be specified explicitly

Message Exchange (2)

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

Wildcard possible: `MPI_ANY_SOURCE`

Wildcard possible: `MPI_ANY_TAG`

- Except for `source` and `status`, parameters are identical with the send counterpart
  - `source` and `tag` need to match a send operation
  - Fewer datatype elements than `count` can be received
    - More would be an error
    - `MPI_PROBE` can be used to receive messages of unknown length
  - `MPI_Status` is required because tag and source can be unknown due to wildcards
    - `status.MPI_SOURCE` and `status.MPI_TAG` contain the required information

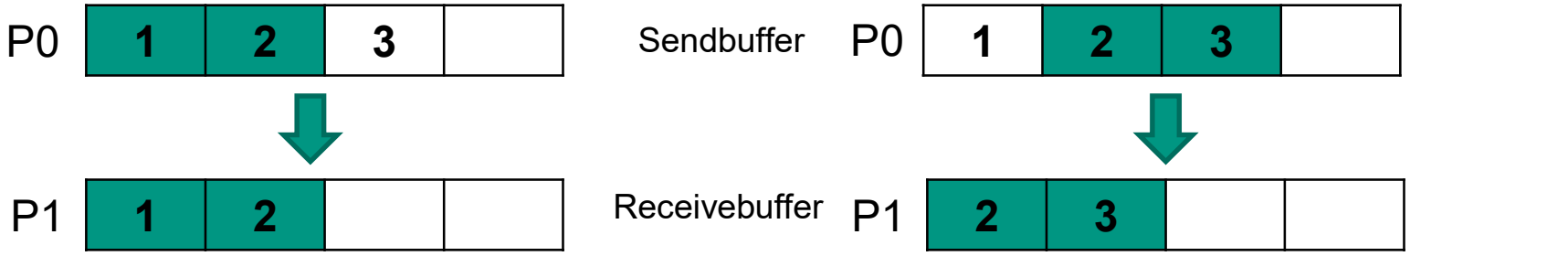
Message Exchange (3)

```
int MPI_Send( void* buffer, int count, MPI_Datatype datatype, …)
int MPI_Recv( void* buffer, int count, MPI_Datatype datatype, …)
```

- `buffer` provides the (initial) memory address of the send or receive buffer (see C/C++ Chapter)
- Elements are sent from or stored to the `count` consecutive memory addresses starting from `buffer`
- ➔ Possible to send from or store elements in an array not starting from its first memory address by provision of other address in the array

```
MPI_Send(sendbuffer, 2, MPI_INT,…)
```

```
int* sendbuffer;
MPI_Send(sendbuffer+1, 2, MPI_INT,…)
```



MPI Send/Receive Example

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

int main(int argc, char** argv) {
    char msg[20];
    int myrank, tag=42;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0) {
        strcpy(msg, "Hello student!");
        MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
    } else {
        MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
        printf("received \"%s\"\n", msg);
    }

    MPI_Finalize();
    return 0;
}
```

What is the potential problem with the destination rank "1" in this example?

Why can't you say `strlen(msg)+1` here (like in the send operation)?

What happens here?

```
#include ...

int main(int argc, char** argv) {
  char msg[20];
  int myRank, tag=42;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  strcpy(msg, "Hello student!");
  if (myRank == 0) {
    MPI_Recv(msg, 20, MPI_CHAR, 1, tag, MPI_COMM_WORLD, &status);
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
    printf("received \"%s\"\n", msg);
  } else {
    MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
    printf("received \"%s\"\n", msg);
  }

  MPI_Finalize(); return 0;
}
```

Temporal Sorting of Messages

- There is no global order on communication events
- But: messages are "non-overtaking", i.e. sorted in time between one sender-receiver pair
  - If a sender sends messages A and B that match two receives, message A will be received before B
  - This is not true if no matching “receive” is available, for example:
    - P0 sends message *A* with no matching "receive"@P1 available
    - ➔ *A* is buffered and sending@P0 terminates
    - P0 sends message *B* (blocking send) with "receive"@P1 available that matches B
    - ➔ sending@P0 terminates and P1 receives message B
    - ➔ A is still in buffer, but receive@P1 finished

➔ As a bottom line, "first matching receive is served"

- Fairness in receiving (from different senders) is not guaranteed
  - i.e. buffering policy/details left to MPI implementation

Communication Modes

- There are four communication modes for send operations:
  1. Synchronous
     - No buffer, synchronization (both sides wait for each other)
  2. Buffered
     - Explicit buffering, no synchronization (no wait for each other)
  3. Ready
     - No buffer, no synchronization, matching receive must already be initiated
  4. Standard
     - May buffer or not, can be synchronous (implementation dependent)
- There is only one receive mode – it matches all four sending modes
- Orthogonal to the modes, communication can be (non-)blocking
  - Non-blocking operations return immediately, independent of the send state
  - The send buffer cannot be safely reused after non-blocking operations
  - The send buffer can be safely reused after blocking operations
  - Per default, MPI operations are blocking

Synchronous and Buffered Send

1. Synchronous send
   - Send can be initiated before receive was started
   - Matching receive must be posted, before operation sends anything ("rendezvous semantics", non-local operation) – no buffer used
   - Can be used for synchronization instead of a barrier
   - Safest and most portable option
     - Order of send and receive operation is irrelevant
     - No hidden internal buffers are used
   - Potentially high synchronization overhead
   - Operation is non-local: matching receive is required
2. Buffered send
   - Can complete before receive even starts (local operation)
   - Explicit user-defined buffer is used, allocation with `MPI_Buffer_attach`
   - Error, if buffer overflow (especially if no receive follows)
   - No synchronization overhead, but extra copy

Ready and Standard Send

3. Ready send
   - Receive **must** initiate before send starts (user is responsible for writing a correct program) – if not, behavior is undefined
   - No buffer used, data is transferred directly to the receiver
   - Lowest sender overhead
     - No synchronization, no waiting for receiver
     - No extra copy to a buffer
4. Standard send
   - Sends message with unknown receive state
   - May buffer: buffer may be on sender, receiver or both sides
   - May be synchronous: gets synchronous if hidden buffer size exceeded
   - Can lead to unexpected timing behavior

MPI Send Operation Modes

- `MPI_Send`: standard-mode blocking send
- `MPI_Bsend`: buffered-mode blocking send
- `MPI_Ssend`: synchronous-mode blocking send
- `MPI_Rsend`: ready-mode blocking send
- All send operations have the same parameter list

- `MPI_Sendrecv` is blocking
  - But internally parallel in MPI (like threads with **`join`**)
  - Send buffers and receive buffers must be disjoint
- `MPI_Sendrecv_replace` available with **one** buffer for send and receive
  - Can be seen as "OUTIN" parameter semantics

(Non-)Blocking Communication

- Orthogonal to the communication modes, communication can be blocking or non-blocking
- Blocking communication
  - Call does not return until operation has been completed
  - It is clear when it is safe to use received or reuse sent data: Send buffer can be safely be reused when the call returns
- Non-blocking communication
  - Call returns immediately
  - Can perform other operations while waiting for communication completion
  - Less deadlock possibility
  - More error-prone: Manually check when the operation is finished and when the send buffer can be safely reused (or reuse an additional buffer)

Non-blocking Operations

- Non-blocking send and receive operations:

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

  - `I` stands for immediate
  - `request` is a pointer to status information about the operation
- Send and receive operations can be checked for completion

```
int MPI_Test(MPI_Request* r, int* flag, MPI_Status* s)
```

  - Non-blocking check
  - `flag` set to 1 if operation completed (0 if not yet)

```
int MPI_Wait(MPI_Request* r, MPI_Status* s)
```

  - Blocking check

Non-blocking Receive Example

- Waiting for a message, the receiver can perform other operations:

```
int *msg, msglen, flag=0;
MPI_Status status;
MPI_Request request;

…
MPI_Irecv(msg, msglen, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
    MPI_COMM_WORLD, &request);
…

while (flag==0) {
    MPI_Test(&request, &flag, &status);
    // other calculations
}
…
```

Adapted from: https://www.tu-chemnitz.de/informatik/RA/projects/mpihelp/test.html

Data Distribution

- A typical usage pattern in parallel programming is
  1. Distribute (breakup) input data
  2. Do calculations
  3. Collect results

➔ This can be achieved with collective operations in MPI

- Such as for broadcasting and collecting data
- We have seen `MPI_Barrier` already

- A procedure is collective if all processes in a process group (ultimately) need to invoke it
  - Thus, a collective procedure is *synchronizing* if all processes in a group must have started this procedure before *any* process can complete its invocation of this procedure
  - Collective calls that use the same communicator must be executed in the same order by all members of a process group

Broadcasting

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

- `root` is the rank of the message sender
- `root` uses `buffer` to *provide* data
- All others (i.e. receivers) use `buffer` for receiving data
  - Other parameters (`count, type, comm`) must be identical
- `root` sends the data to itself, too
  - into its part of *receive* buffer

- And how is a broadcast received?
  - Remember that everybody has to participate in a collective operation

see: http://stackoverflow.com/questions/7864075/using-mpi-bcast-for-mpi-communication
more examples: http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node68.html#Node68

Broadcast Example

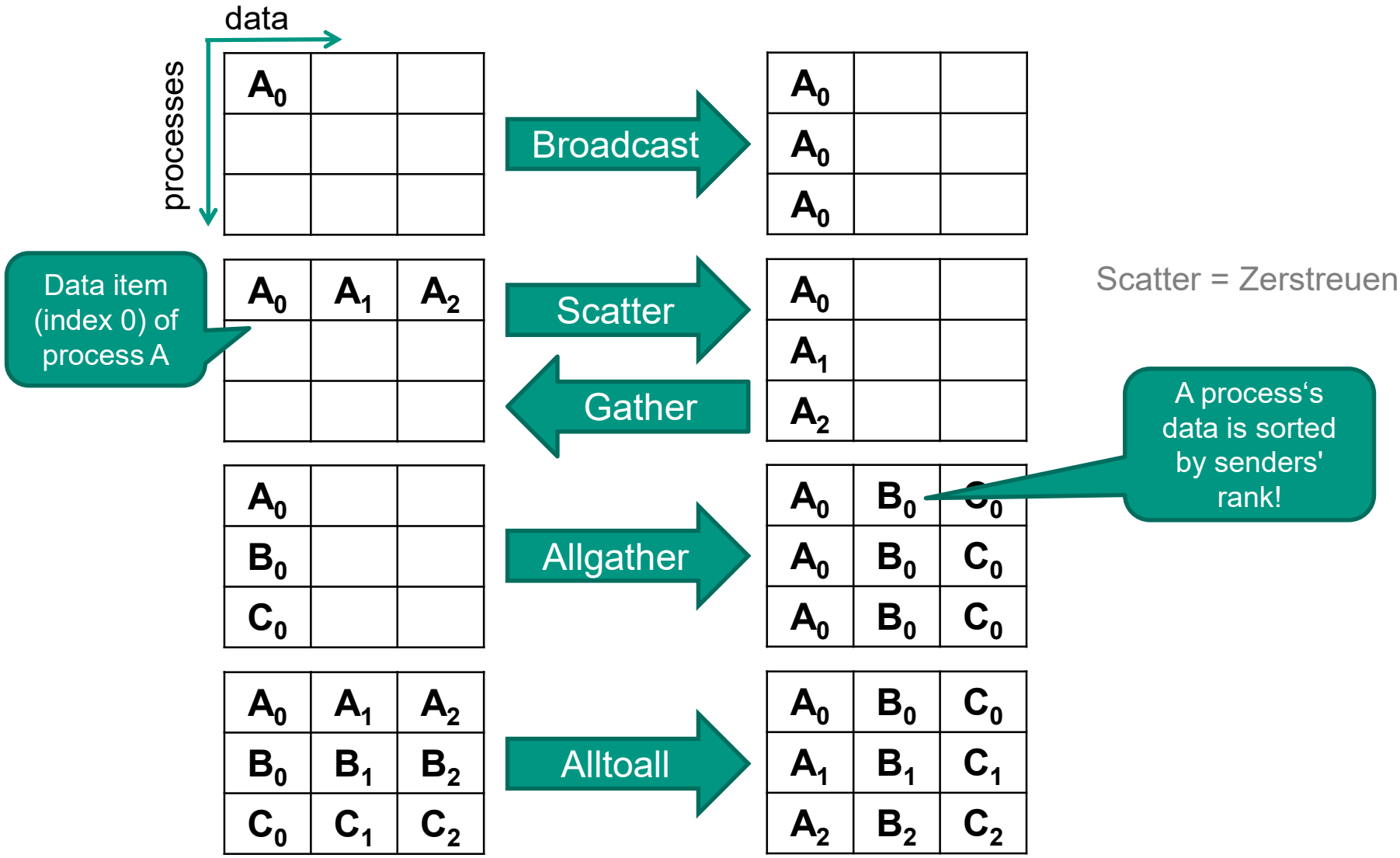
```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int rank, buf;

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

    if(rank == 0) {
       buf = 777;
    }

    printf("[%d]: Before Bcast, buf is %d\n", rank, buf);
    MPI_Bcast(&buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("[%d]: After Bcast, buf is %d\n", rank, buf);

    MPI_Finalize();
    return 0;
}
```

Same program for everyone!

Master process is responsible for distributing data

Everyone has to call the collective operation!

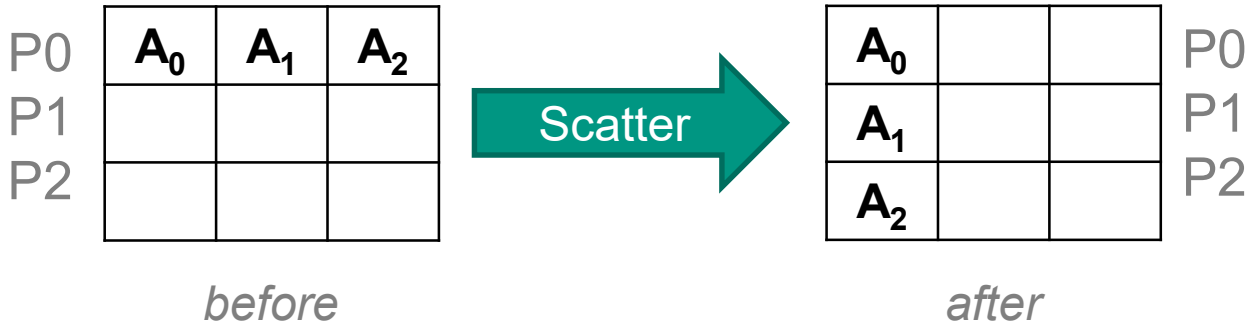
Global Collective Operations



MPI_Scatter

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

- All receivers get equal-sized but *content-different* data
  - ➔ scatter and broadcast are different
- `sendcount` and `recvcount` are the numbers of elements sent to and received by one process and are usually equal



more examples: http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node72.html#Node72

Vector variant of `MPI_Scatter`

```
int MPI_Scatterv( void* sendbuf, int* sendcounts, int* displacements,
                  MPI_Datatype sendtype, void* recvbuf, int recvcount,
                  MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- Allows varying counts for data sent to each process
- `sendcounts`: integer array with the number of elements to send to each process
- `displacements`: integer array, entry *i* specifies the displacement relative to `sendbuf` from which to take the outgoing data to process *i* (gaps allowed but no overlaps)
- `sendtype`: data type of send buffer elements (handle)
- `recvcount`: number of elements in receive buffer (integer)
- `recvtype`: data type of receive buffer elements (handle)

MPI_Gather

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

- root's buffer contains collected data *sorted by rank*, including root's own buffer contents
- Receive buffer is ignored by all non-root processes
- recvcount: number of items received from ***each*** process
- Recall that MPI_Scatter is the inverse of MPI_Gather

- MPI_Gatherv is the vector variant again

more examples: http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node70.html#Node70

Scatter/Gather Example

```
int main(int argc, char** argv){
   // …
   int total = 0;
   local_array = (int*) malloc(count * sizeof(int));
   if (rank == 0) {
      size = count * numnodes;
      send_array =(int*) malloc(size * sizeof(int));
      back_array =(int*) malloc(numnodes * sizeof(int));
      for (i = 0; i < size; i++) send_array[i]=i;
   }

   MPI_Scatter( send_array, count, MPI_INT, local_array, count,
                MPI_INT, 0, MPI_COMM_WORLD);
   // … (each processor sums up his local_array into back_array)
   MPI_Gather( &total, 1, MPI_INT, back_array, 1, MPI_INT, 0,
               MPI_COMM_WORLD);

   // …
}
```

e.g. count = 4

Reserve memory for two arrays

Fill send_array with integer numbers

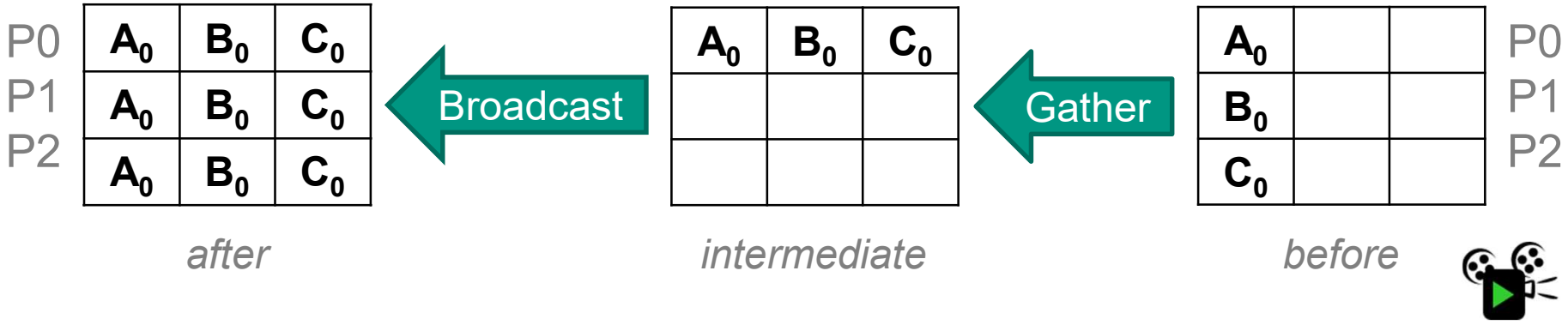
"Multi-Broadcast" with `MPI_Allgather`

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

- Basically this is a gather + broadcast
- Multi-Broadcast: for each process p_j in `comm`: p_j collects and sends the same data to all other processes in `comm`

➔ At the end, the buffer of each process in `comm` has the *same* data (incl. its own data) in the *same* order

- Again, the vector variant is `MPI_Allgatherv`



more examples: http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node74.html#Node74

MPI_Alltoall

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

- A sender p_s sends to receiver p_r only its r-th element
- A receiver p_r stores information from sender p_s at the position s in its buffer

- `MPI_Alltoallw` allows separate specification of count, displacement and datatype for each block
- Again, the vector variant is `MPI_Alltoallv`

Collective Operations with Send/Receive

- Global collective operations can be seen as a combination of send/receive operations
  - Many implementations make use of further MPI features to increase performance
- Parameters of collective operations can be mapped to parameters of send/receive operations

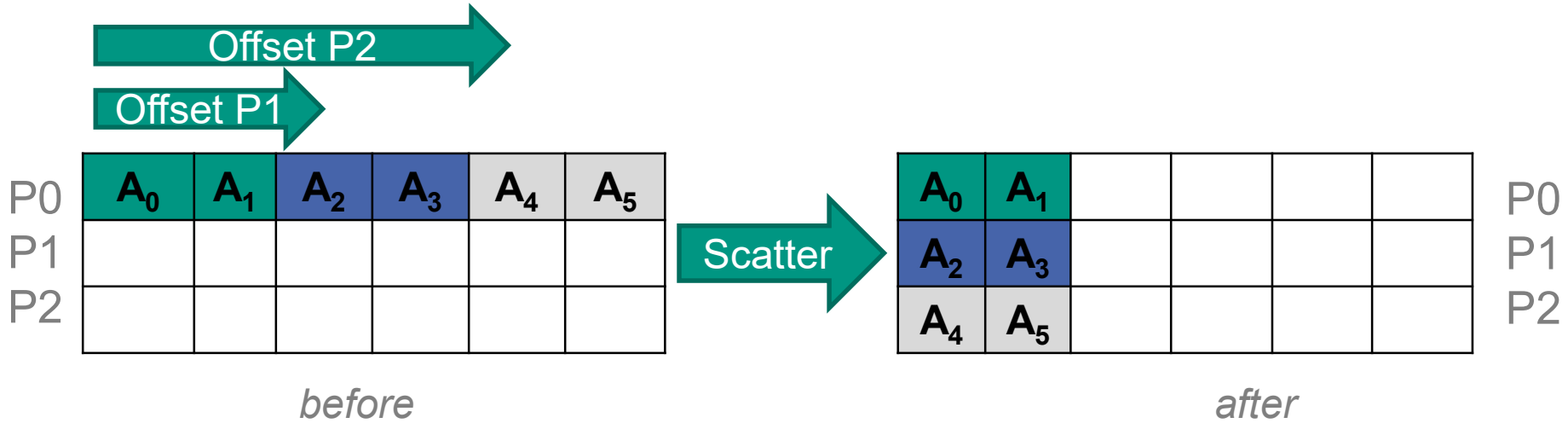
```
int MPI_Scatter( void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

```
int MPI_Send( void* sendbuf, int sendcount, MPI_Datatype sendtype,
              int dest, int tag, MPI_Comm comm)
```

```
int MPI_Recv( void* recvbuf, int recvcount, MPI_Datatype recvtype,
              int source, int tag, MPI_Comm comm, MPI_Status* status)
```

Collective Operations with Send/Receive (2)

- Collective operations partition the data for each process according to the use of send/receive
- The memory address of the buffer to send elements from or store them into is calculated for each proccess by
  - The buffers initial address
  - For each process: an „offset“ depending on the number of send/received elements and the rank of the process

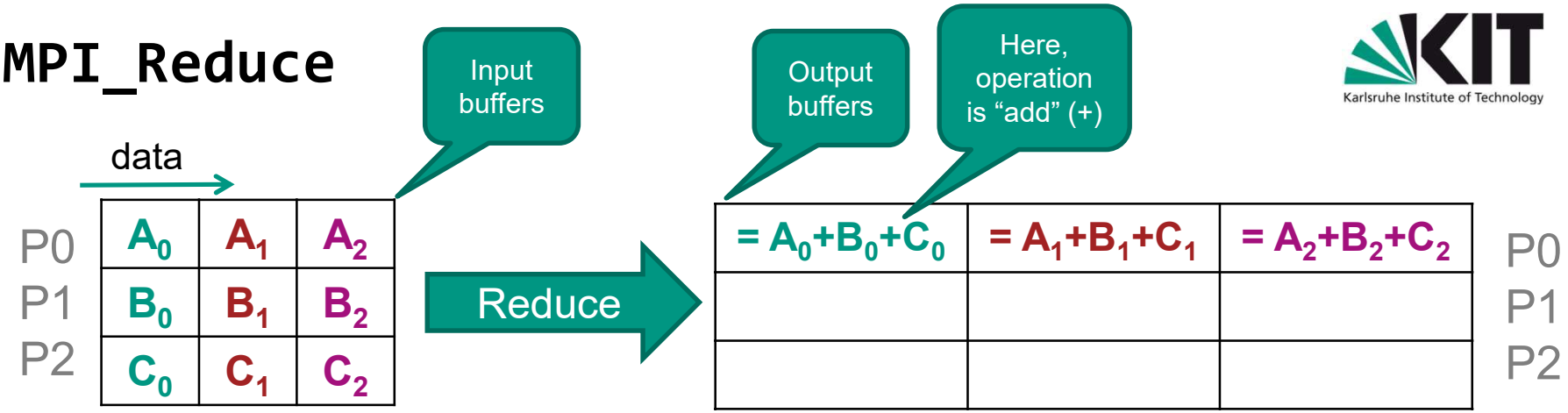


Example: Scatter with send-/recvCount = 2

Collective Operations with Send/Receive (3)

```
#include ...

int main(int argc, char** argv) {
  int sendBuffer[] = {1,2,3,4,5,6};
  int myRank, tag=42, commSize /* commSize = 3 */;
  //… get myRank and commSize
  int count = 6 / commSize; /*partition sendBuffer (6 Elements) evenly*/
  int recvBuffer[count];

  if (myRank == 0) {
    for(int proc = 0; process < size; process++){
        int offset = process * count;
        MPI_Isend(sendBuffer + offset, count, MPI_INT, process, tag,
                  MPI_COMM_WORLD);
    }
  }
  MPI_Recv(recvBuffer, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
 }

  MPI_Finalize(); return 0;
}
```

MPI_Reduce




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

- Applies an operation to the data in `sendbuf` and stores the result in `recvbuf` of the root process
- `count`: number of columns in the output buffer
- `op`: can be …
  - logical / bitwise “and” / “or”: `MPI_LAND` / `MPI_BAND` / `MPI_LOR` / `MPI_BOR` / ...
  - `MPI_MAX` / `MPI_MIN` / `MPI_SUM` / `MPI_PROD` / ...
  - `MPI_MINLOC` / `MPI_MAXLOC` find local minimum / maximum and return the value of the "causing" rank
  - own operations can also be defined

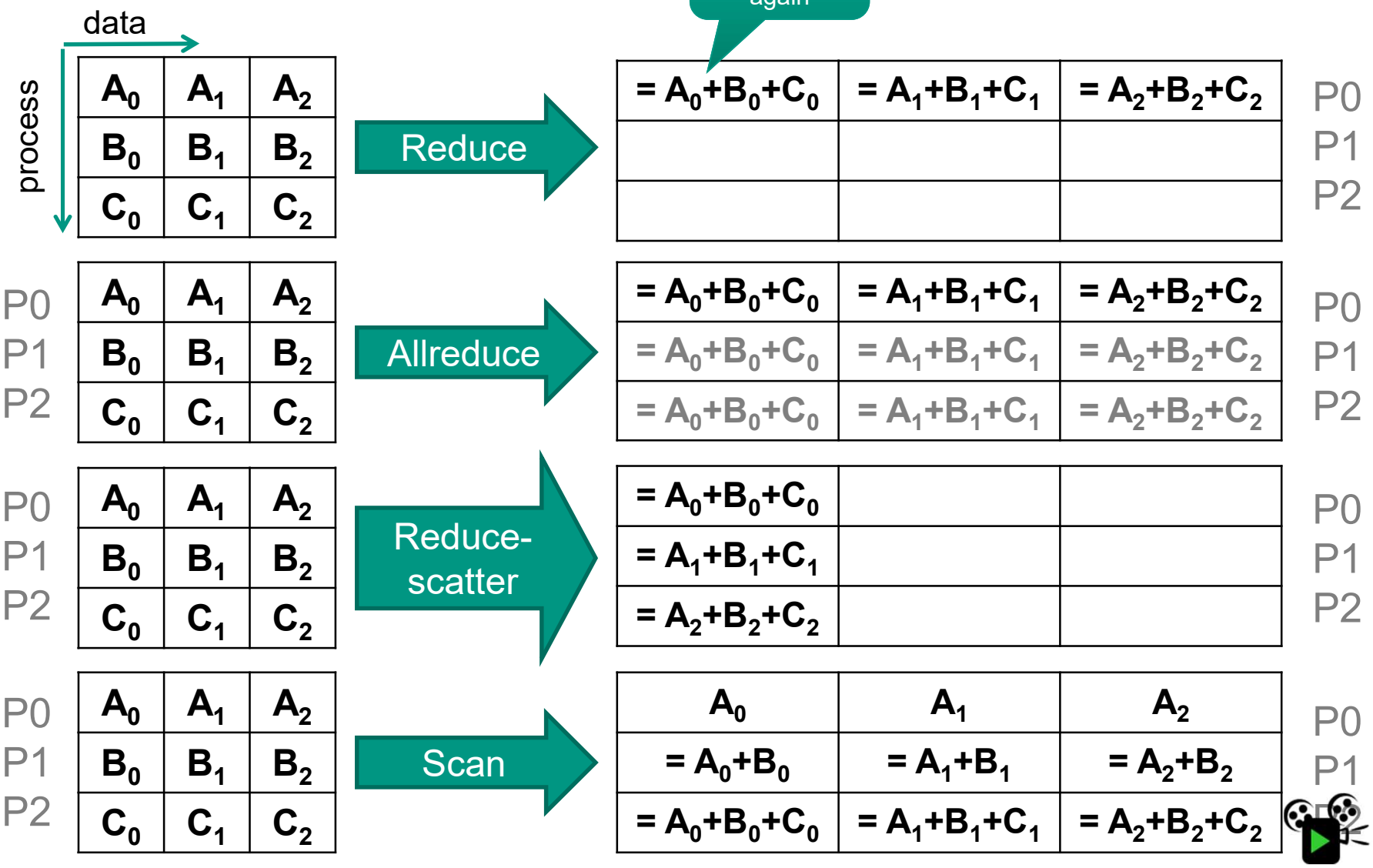
MPI_Reduce: Example

```
int     myrank, numprocs;
double  mytime, /* variables used for gathering timing statistics */
        maxtime, mintime, avgtime;

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Barrier(MPI_COMM_WORLD);  /* synchronize all processes */

mytime = MPI_Wtime();  /* get time just before work section */
/* Do some work */
mytime = MPI_Wtime() - mytime;  /* get time just after work section */

/* compute max, min, and average timing statistics */
MPI_Reduce(&mytime, &maxtime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Reduce(&mytime, &mintime, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
MPI_Reduce(&mytime, &avgtime, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (myrank == 0) {
    avgtime /= numprocs;
    printf("Min: %lf  Max: %lf  Avg:  %lf\n", mintime, maxtime, avgtime);
}
```

Adapted from: http://www.cs.utexas.edu/~pingali/CS378/2011sp/lectures/mpiExamples.txt

Further Reduce Functions

Operation is "add" ("+") again



MapReduce

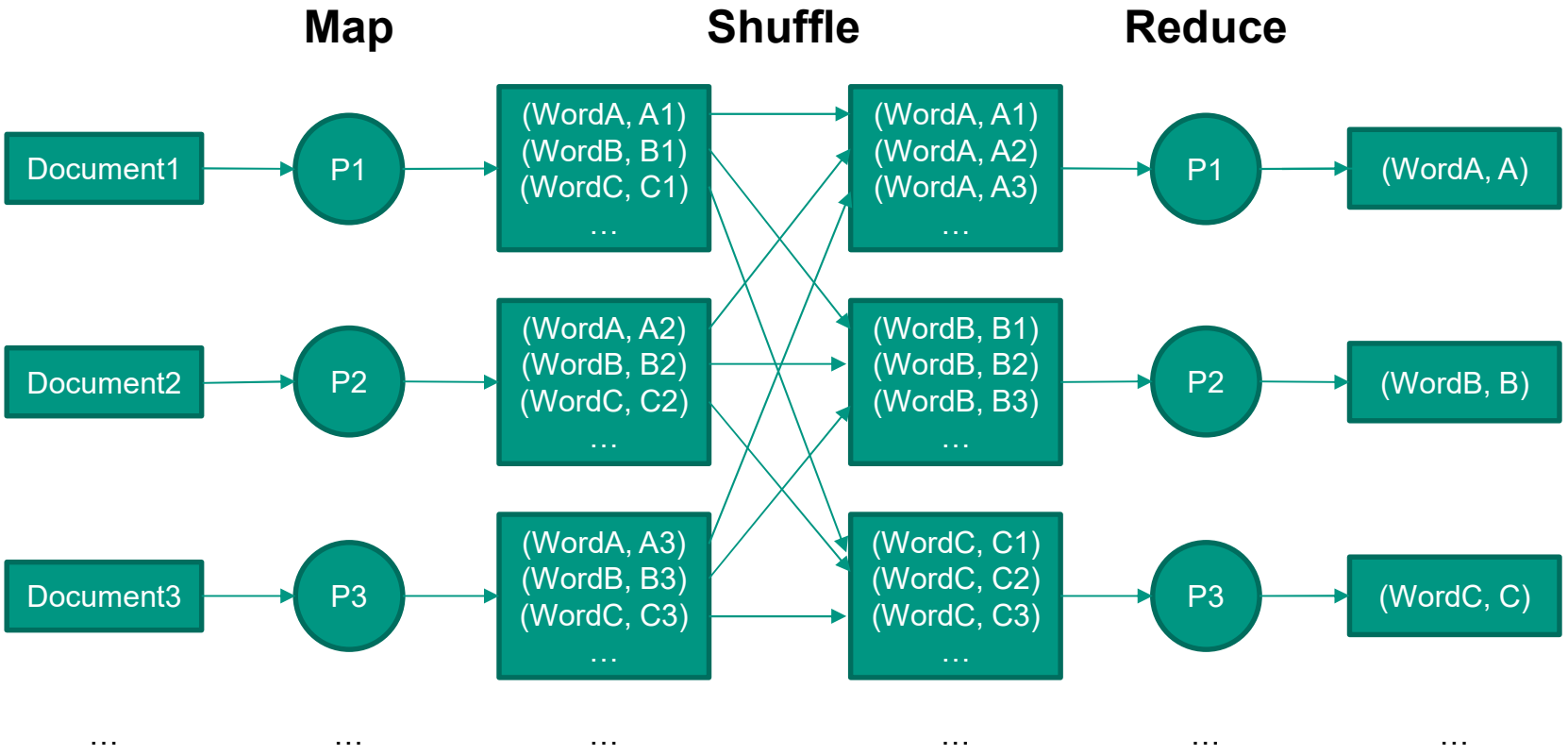
- Programming model for processing data sets with parallel algorithms
- Originally developed by Google
- Assumes data distributed on different computing nodes
- Consists of three steps
  1. Map: apply a map function to the data on each computing node once, returning key-value pairs as a result
  2. Shuffle: redistribute data by output keys of the map function, so that one computing node contains all data for one key
  3. Reduce: apply reduce function on each key once
- The map and reduce steps are user defined, whereas the shuffle step is generic

Example: Word Counts in Document Set

Map: Calculate count of each word for one document on each node

Shuffle: Move all counts for one word to the same computing node

Reduce: Sum up the partial counts of each word to get the overall sum



MPI and MapReduce

- MapReduce is a programming model for specific problems (non-iteratively solvable with one map and reduce function), whereas MPI is a generic library for inter-process communication
- MapReduce can be realized using MPI
  - Map can be realized using MPI scatter operations
  - Reduce can be realized with MPI reduce operations
- Apache Hadoop is a popular library implementing the MapReduce principle for Java
- There is also a library for MPI: http://mapreduce.sandia.gov

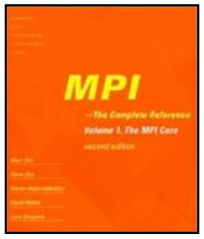
[http://www-01.ibm.com/software/data/infosphere/hadoop/mapreduce]

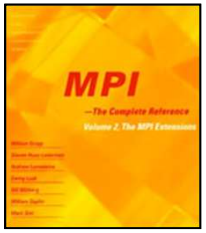
Conclusion

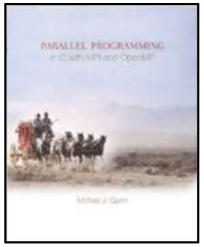
- MPI is a parallelization solution based on messages
  - Targets distributed systems
  - No shared memory is available
- In MPI, all data needs to be distributed appropriately
  - Primitive send and receive operations
    - Blocking / non-blocking variants
    - Synchronous / asynchronous variants (communication modes)
  - Collective operations can ease data distribution
    - Broadcast
    - Scatter, Gather
    - Allgather, Alltoall
    - Reduce
- MapReduce is a programming model for processing large data sets with specific parallel algorithms

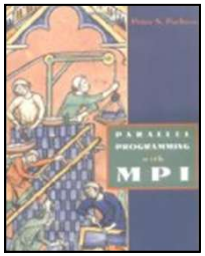
Literature and References

- **http://www.mpi-forum.org**: official information, highly recommended
  - Free download of MPI-2.2 as PDF (600+ pages); has examples, explanations
- **MPI: The Complete Reference**
  Vol. 1, The MPI Core, 2nd Edition
  M. Snir et al, 1998, MIT Press
- **MPI: The Complete Reference**
  Vol. 2, The MPI-2 Extentions
  W. Gropp et al., 1998, MIT Press
- **MPI - eine Einführung (in deutsch)**
  W. Gropp, 2007, Oldenbourg-Verlag
- **Using MPI-2: Advanced Features**
  W. Gropp et al., 1998, MIT Press
- **Parallel programming in C with MPI and OpenMP**
  M. Quinn, 2003, Mc-Graw Hill
- **Parallel programming with MPI**
  P. Pacheco, 1994 (reprint 2009)
  Morgan-Kaufmann Publishers
  outdated, does not cover MPI-2
- Some slides were inspired by
  http://www.uio.no/studier/emner/matnat/ifi/INF3380/v10/undervisningsmateriale/inf3380-week06.pdf









APPENDIX

Appendix: Terms and Notation

- Distinguish: procedure call ≠ procedure execution!
  - MPI context: buffers for loosely coupled send and receive
- **Non-blocking procedure**: call (invocation) can return
  - ... before the started operation (execution) completes
  - ... while the involved resources are still being used
  - Request objects to "track" progress of non-blocking calls
- **Blocking procedure**:
  - Return from procedure call means that reuse of resources (not only those specified in the call) allowed
- In C/C++: **`void`*** for arguments with a "choice type"

Appendix: Further Terminology

- **Local** procedure: its completion does not depend on the execution of other processes' MPI calls
- **Non-local** procedure: may (but does not have to!) depend on the execution of other processes' calls
- **Opaque objects:** size/structure not visible to the user; accessed and (de)allocated via *handles*
  - In C++, handles are distinct objects
  - Use: managing system memory (messages, buffers, ...)

Appendix: Combining Send and Receive

- Frequent sequential pattern: "send one message and receive a different one"
  - Destination and source can be different or equal
  - Useful for deadlock prevention in cyclic communication

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

  - `MPI_Sendrecv` is blocking

Same communicator, different tags!

Appendix: Messages with Structs

- MPI also allows to send around custom datatypes (structs)
  - Define the struct

```
struct {
   char text[50]; int counter; double x, y;
} myStruct;
```

  - Setup some MPI information

```
int blockcounts[3] = {50, 1, 2};
MPI_Datatype types[3];
MPI_Aint displacements[3];
MPI_Datatype type;
MPI_Address(&myStruct.text, &displacement[0]);
                           /* analogue for counter and x */
types[0] = MPI_CHAR;       /* analogue with int and double */
```

  - Adjust displacements in a for loop

```
displacements[i] -= (displacements[0]);
```

  - Create and commit the data type

```
MPI_Type_struct(3, blockcounts, displacements, types, &myStruct);
MPI_Type_commit(&myStruct);
```

  - And use it

```
MPI_Bcast(MPI_BOTTOM, 1, cmdtype, 0, MPI_COMM_WORLD);
```