RWTH AACHEN UNIVERSITY

MPI – Part I
Basics

General Information, Communication via Message Passing

MPI – Part I - Contents

What is MPI? (1 / 3)

- **MPI means *Message Passing Interface***
  - Messages are data packets exchanged between processes
  - Interface definition only
- **MPI is a de-facto industry standard API for message passing**
- **Latest Version 3.1, June 2015**
- **Different implementations exist:**
  - typically provided as libraries with C, C++, Fortran bindings
  - free MPI implementations:
    - MPICH http://www.mpich.org
    - OpenMPI (we use this) http://www.open-mpi.org

What is MPI? (2 / 3)

- Since the finalization of the first standard, MPI has replaced many previous message passing approaches
- The MPI API is very large (>300 subroutines), but with only 6 – 10 different calls serious MPI applications can be programmed
- Tools for debugging and runtime analysis are widely available

What is MPI? (3 / 3)

- MPI is easy to learn, but may be hard to apply to real world applications (practice!)
- OpenMP is an interesting alternative for shared memory architectures like today's multicore processors

Other Computer Languages

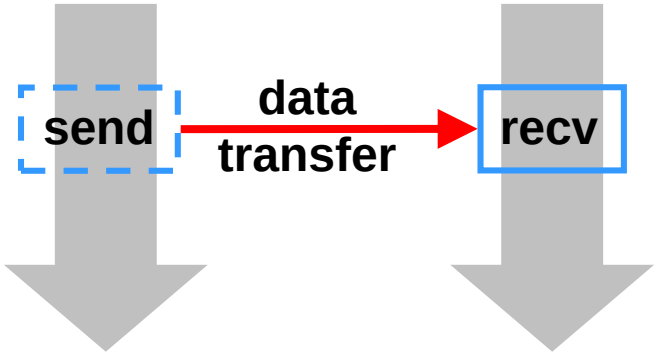
- **Official standard defines language bindings for Fortran, C, and C++; however, in 3.0 C++ binding is deprecated …**

- **Several unofficial bindings for other languages exist:**
  - Java: mpiJava, MPJ (Express)
  - .NET: MPI.NET
  - Python: ScientificPython, MYMPI, pyMPI
  - Ruby: ruby-mpi
  - Matlab: CMTM
  - R: Rmpi
  - Haskell hMPI, Haskell-MPI

- **Parallel processes with local address spaces**
- **Processes communicate (typical pattern for parallel programming)**
  - two-sided operation
  - send & receive
  - receiver has to wait until

    the data has been sent

    → synchronization



Message Passing Overhead

- Performance of communication primitives of parallel computers is critical for the overall system performance

- Characterization of communication overhead is very important to estimate the global performance of parallel applications and to detect possible bottlenecks

→ Goal: *optimize the trade off between computations and communications in parallel applications*

Parallelization Issues

- **Process handling**
  - Process pool and process identification
  - Synchronization mechanisms
- **Message handling**
  - Send and receive messages (with data)
  - One-to-One, One-To-All, All-to-All, …
  - Wait for data or not
- **Data handling**
  - different memory layouts and structures
- **IO handling**
  - Work with large data

Parallelization Issues - MPI Answers

- **Process handling**
  - Communicators, Groups, Ranks; Collective Commands
- **Message Handling**
  - Blocking, non-blocking, synchronous, asynchronous; send and receive;
- **Data handling**
  - Various datatypes and memory handling
- **IO Handling**
  - Large file and memory handling

Literature

- **Parallel Programming With MPI**
  Peter Pacheco
  Morgan Kaufmann (1996)

- **Using MPI 2: Advanced Features of the**
  Message-Passing Interface
  Gropp, Thakur, Lusk
  The MIT Press (1999)

- **Patterns for Parallel Programming**
  Mattson, Sanders, Massingill
  Addison Wesley (2004)

- **MPI: A Message-passing Interface Standard**
  Version 3.0

Further Information

- **Message Passing Interface Forum**
  http://www.mpi-forum.org

- **MPI: The Complete Reference**
  Volume 1 - The MPI Core
  Snir, Otto, Huss-Lederman, Walker, Dongarra
  The MIT Press; 2 Edition (1998)
  http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

MPI – Part I - Contents

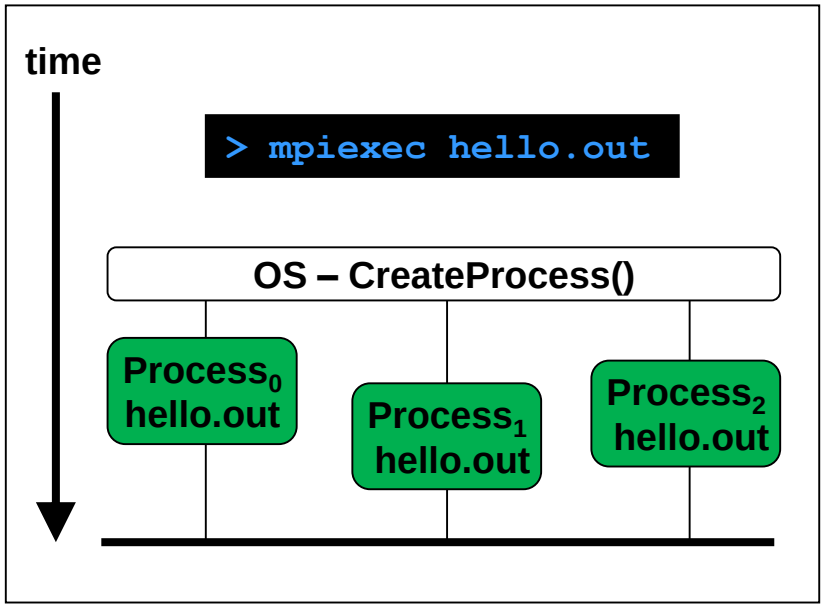
Example: Hello World

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

/*
 * gcc -o hello hello.c
 * ./hello
 * mpiexec -np 3 ./hello
 */

int main(int argc, char* argv[])
{
        printf("Hello World\n");
        return EXIT_SUCCESS;
}
```

Output of Hello World

- **Program produces the same result several times**
- **Program is spawned by OS and runs simultaneously on multiple processors / cores**
- **Program execution progress depends on system load**
  - Exec. and finish time unknown
- **What is missing:**
  - processes communication,

    identification and synchronization



⇨ Purpose of MPI commands

Initialization and Finalization

- **Each processor must initialize and finalize an MPI process**

- **Initialization of the MPI environment**
  - Must be done at the very beginning of the program
  - `int MPI_Init(int* argc, char ***argv)`
  - no MPI function calls before

- **Finalization of the MPI environment**
  - Typically done at the end of the program
  - `int MPI_Finalize(void)`
  - no MPI function calls after

MPI Error Handling

- All MPI routines return an error value or, in C++, throws an exception
- The function format in C is
  - `int error = MPI_Xxxx(parameter-list)`
- Before returning an error, the MPI error handler is called
- By default the handler aborts the MPI job!!!
- Without changing the error handler, you will
  - get a 0 as return value in case of success
  - or the MPI job will abort in case of an error

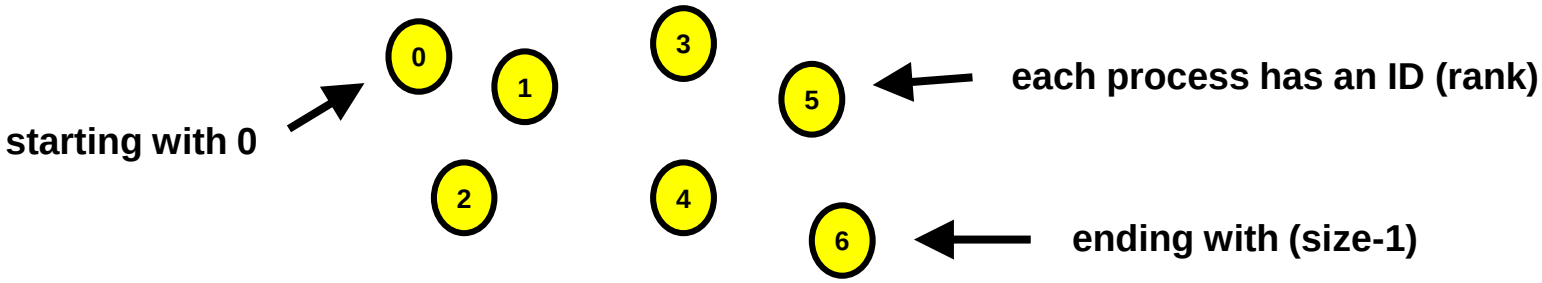
Example: Error

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

/*
 * mpicc -std=c99 error.c -o error
 * mpiexec -np 3 error
 */

int main(int argc, char* argv[])
{
        int procRank,procCount, error;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&procCount);

        error = MPI_Comm_rank(MPI_COMM_WORLD,&procRank);
        printf("Start[%d]/[%d], error %d \n",procRank,procCount,error);

        MPI_Finalize();

        return EXIT_SUCCESS;
}
```

MPI – Part I - Contents

Group

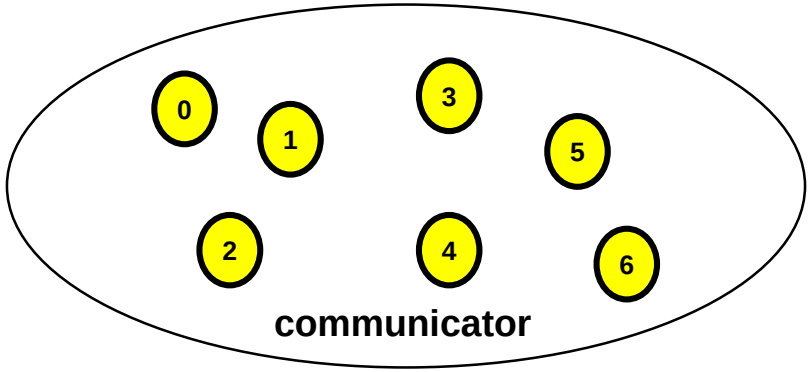
- A group is an ordered set of processes
- Each process in a group is associated with a unique integer rank



- One process can belong to two or more groups
- Groups are dynamic objects in MPI and can be created and destroyed during program execution (Part II)

Communicator

- MPI communication always takes place within a communicator
- Within a communicator a group is used to describe the participants in a communication "universe"



- User can associate an error handler with a communicator
- Communicators are dynamic, i.e., they can be created and destroyed during program execution (Part II)

Predefined Communicators

- `MPI_COMM_WORLD`
  - includes all of the started processes
- `MPI_COMM_SELF`
  - includes only the process itself

- **They are properly defined after `MPI_Init(…)` has been called**

Size and Rank

RWTH AACHEN UNIVERSITY

- **Size**
  - Number of processes within a group
  - Can be determined using
  - `int MPI_Comm_size(MPI_Comm comm, int *nprocs)`
- **Rank**
  - Identifies processes within a group
  - Can be determined using
  - `int MPI_Comm_rank(MPI_Comm comm, int *myrank)`

Example: Size and Rank

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

/*
 * mpicc ranks.c -o ranks
 * ./ranks
 * mpiexec -np 3 ranks
 */

int main(int argc, char* argv[])
{
        int procRank,procCount;

        MPI_Init(&argc,&argv);

        MPI_Comm_size(MPI_COMM_WORLD,&procCount);
        MPI_Comm_rank(MPI_COMM_WORLD,&procRank);

        printf("Start[%d]/[%d] \n",procRank,procCount);

        MPI_Finalize();

        return EXIT_SUCCESS;
}
```

Single Program Multiple Data (SPMD)

- **If necessary, specific behavior is done programmatically, e.g. by evaluation of some special process identifier**

```
if (processID == specialID) {
  /* specific behavior here */
} else {
  /* default behavior here */
}
```

- **Usually one process acts as a so-called `master´**
  - does initial stuff , distributes tasks, …
  - usually this is process with rank zero

Summary

- Basic concept of message passing
- Initialization and finalization
- Communicators `MPI_COMM_WORLD`, `MPI_COMM_SELF`
- Size & rank

RWTH AACHEN UNIVERSITY

3. The first MPI Program

MPI – Part I - Contents

Communication Issues

- **How does sender and receiver know, that a message is sent, and from whom?**

- **What kind of data is sent?**
  - How large is the message?
  - What datatype is used?
  - What about little and big endian?

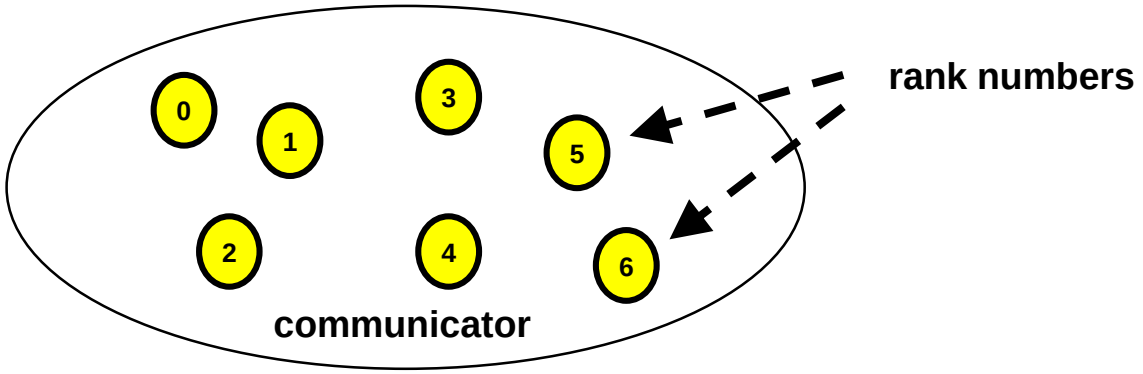
- **Waiting**
  - Does a sending operation wait until message is delivered?
  - Does a receiving operation wait until a message comes?

Communication Issues - MPI Answers

- **How does sender and receiver know, that a message is sent, and from whom?**
  - Program logic, i.e. the programmer is responsible (usually)
- **What kind of data is sent?**
  - Message infos facts
  - MPI datatypes
  - MPI takes care
- **Waiting**
  - Different send and receive functions available
  - Blocking, non-blocking, synchronous, asynchronous

Point to Point Communication (1 / 2)

- Communication between exactly two processes
- Communication takes place within a communicator
- Identification of processes by their rank numbers in the communicator



Example: Simple Send

```
[…]
        int message = -1;
        enum { tagSend = 1 };

        printf("Message undef.: %i\n",message);

        // root defines a message and sends it to the worker
        if (0 == procRank)
        {
                message = 42;

                // remember &message returns address of message
                MPI_Send(&message, 1, MPI_INT, 1, tagSend, MPI_COMM_WORLD);
        }
        // if (1==procRank); only one worker and worker receives the message
        else
        {
                // parameters must match!!!
                MPI_Recv(&message, 1, MPI_INT, 0, tagSend, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);

                printf("Recv. Message: %i\n",message);
        }
[…]
```

Example: More Send

```
[…]
        const int length = 5; // change output if length!=5
        int message[length];
        enum { tagSend = 1 };

        if (0 == procRank) // root recv.
        {
                for (int i=1; i<procCount; ++i)
                {
                        // receive from process i
                        MPI_Recv(message, length, MPI_INT, i, tagSend,
                                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                }
        }
        else // workers send
        {
                for (int i=0; i<length; ++i)
                {
                        message[i]=procRank+i;
                }

                MPI_Send(message, length, MPI_INT, 0, tagSend, MPI_COMM_WORLD);
        }
[…]
```

Send

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

- buf: starting address of the message
- count: number of elements
- datatype: type of each element
- dest: rank of destination in communicator *comm*
- tag: message identification
- comm: communicator

Receive

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

- buf: starting address of the message
- count: number of elements
- datatype: type of each element
- src: rank of source in communicator *comm*
- tag: message identification
- comm: communicator
- status: envelope information (message information)

MPI Data Types

- **MPI uses pre-defined data types which correspond to the data types of the used language**

MPI Type	C Type
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double

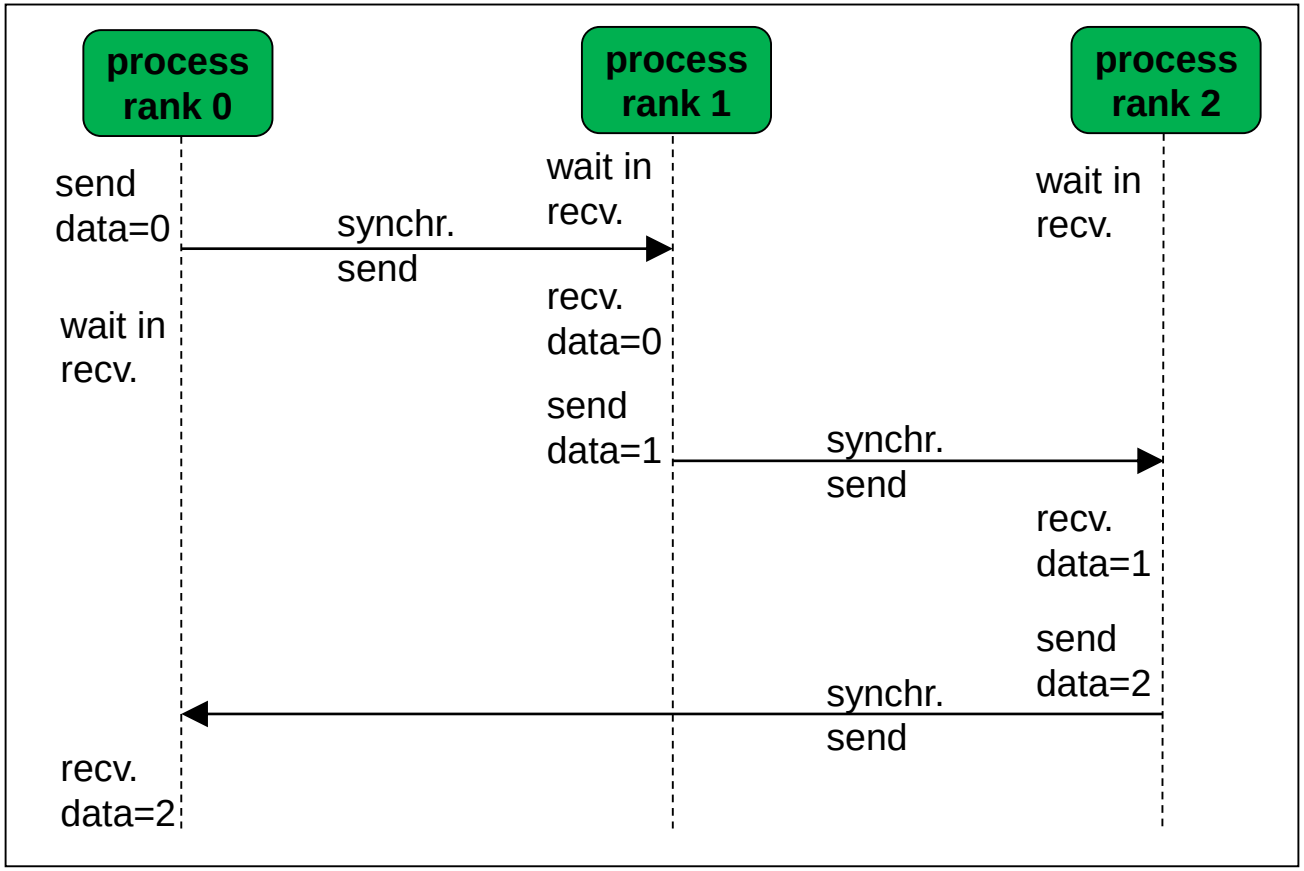
- **Different MPI types for Fortran**
- **Complete list in MPI reference**

Requirements

- Sender specifies a valid destination
- Receiver specifies a valid source
- Communicator is the same
- Matching tags
- Matching data types
- Receiver’s buffer large enough!!!

Data Chain – Timing Example

- 3 processes



MPI_ANY_SOURCE and MPI_ANY_TAG

- It is possible to receive messages from any other process and with any message tag by using `MPI_ANY_SOURCE` and `MPI_ANY_TAG`
- Actual source and tag are included in the status variable

Status Variable contains Envelope information

- **Source:**
  - `int src = status.MPI_SOURCE;`
- **Tag:**
  - `int tag = status.MPI_TAG;`
- **Message size (element count of data):**
  - `MPI_Get_count(*status, mpi_datatype, *count)`

Example: Anytag

```
[…]
        if (0 == procRank) // root recv.
        {
                 // receive messages from the other processes in arbitrary order
                for (int i=1; i<procCount; ++i)
                {
                        MPI_Status status;

                        // receive from process i
                        MPI_Recv(message, length, MPI_INT, MPI_ANY_SOURCE,
                                MPI_ANY_TAG, MPI_COMM_WORLD, &status);

                        printf("Message: %i %i %i %i %i\n",message[0],
                                message[1],message[2],message[3],message[4]);

                        printf("Source:   %d\n",status.MPI_SOURCE);
                        printf("Tag:      %d\n",status.MPI_TAG);
                }
        }
[…]
```

Receiving Messages of unknown Length

- `MPI_Probe` is a call that returns only after a matching message has been found (test for incoming messages)
  - `int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
- Processes can find out the message length using the status variable
  - `int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`
- Later they can receive the message (of length count)

Example: Message of unknown Length

```
[…]
        MPI_Status status;
        MPI_Probe(0, tagSend, MPI_COMM_WORLD, &status);
        int k;

        MPI_Get_count(&status,MPI_INT, &k);
        int* message = (int*)malloc(k*sizeof(int));

        // !!! message is still a pointer to the buffer !!!
        MPI_Recv(message, k, MPI_INT, 0, tagSend, MPI_COMM_WORLD, &status);
        printf("Recv. Message, proc %d, length %d: ... %d
                ...\n",procRank,k,message[2]);

        free(message);
[…]
```

Summary

- Send and receive operations
- Data types in MPI
- Receive messages of unknown length from unknown source

RWTH AACHEN UNIVERSITY

4. Ping Pong

5. Send Data to all Processes, Part I

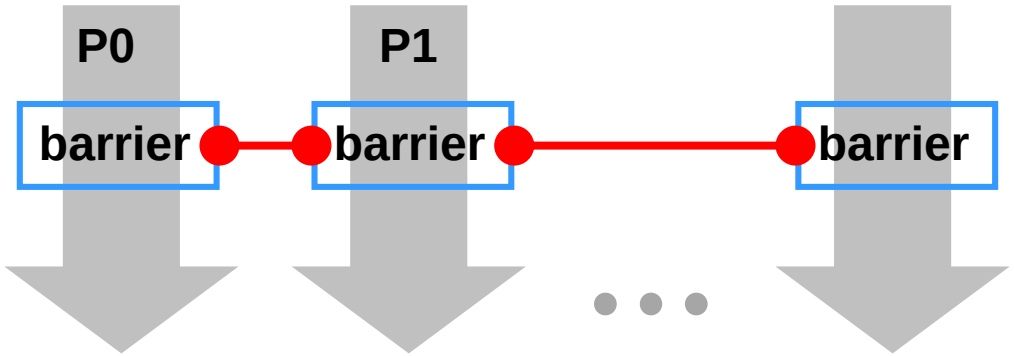
MPI – Part I - Contents

Collective Communication

- **Collective operation over a communicator**
- **All processes in the communicator must call the same routine**
- **Collective operations include synchronization between processes**
- **What we learn:**
  - barriers
  - time
  - broadcast
  - scatter, gather
  - global reduction operations

Barriers

- Synchronizes all processes in a communicator
  - `int MPI_Barrier(MPI_Comm comm)`
- Each process must wait until all have reached the barrier



- Usually never needed as all synchronization is done by data communication
- Used for debugging, profiling or time measurement

Time Measurement

- **To measure the runtime one can use**
  - `double MPI_Wtime(void)`
    - Returns a floating-point number of seconds representing elapsed wall-clock time since some time in the past

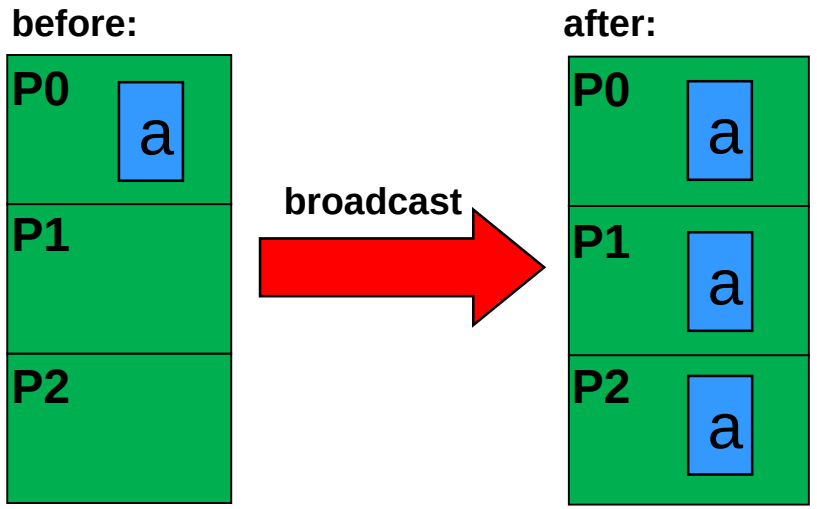
- **The “time in the past" is guaranteed not to change during the life of the process**

Example: Time and Barrier

```
[…]
        double tm0, tm1;

        tm0 = MPI_Wtime();

        sleep(procRank);

        // comment out
        MPI_Barrier(MPI_COMM_WORLD); // all wait

        tm1 = MPI_Wtime();

        printf("proc %d, Wtime %lf \n",procRank,(tm1-tm0));
[…]
```

Broadcast (One-To-All)

- A one-to-many operation
- Again, called by all processes
- Root distributes a message among all processes
  - `int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
    - root: rank of broadcast process in *comm*

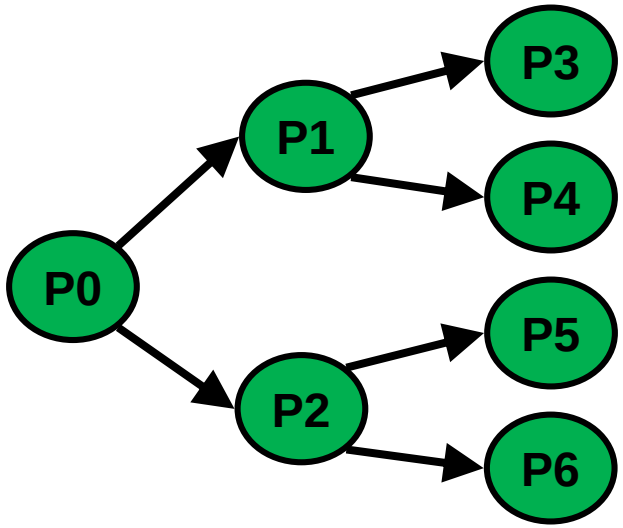


Example: Broadcast

```
[…]
        int message = -1;
        enum { tagSend = 1 };

        // init for master
        if (0 == procRank)
        {
                message = 42;
        }

        MPI_Bcast(&message,1, MPI_INT, 0, MPI_COMM_WORLD);

        printf("proc %d recv. bcast %d \n",procRank,message);
[…]
```

Broadcast Strategy

- All processes, i.e. sender and receiver must execute `MPI_Bcast`
- Don´t try to receive with MPI_Recv !
- Reason: all processes have to work together to make the broadcast most efficient



Broadcast strategy depends on the MPI implementation, but is probably like this

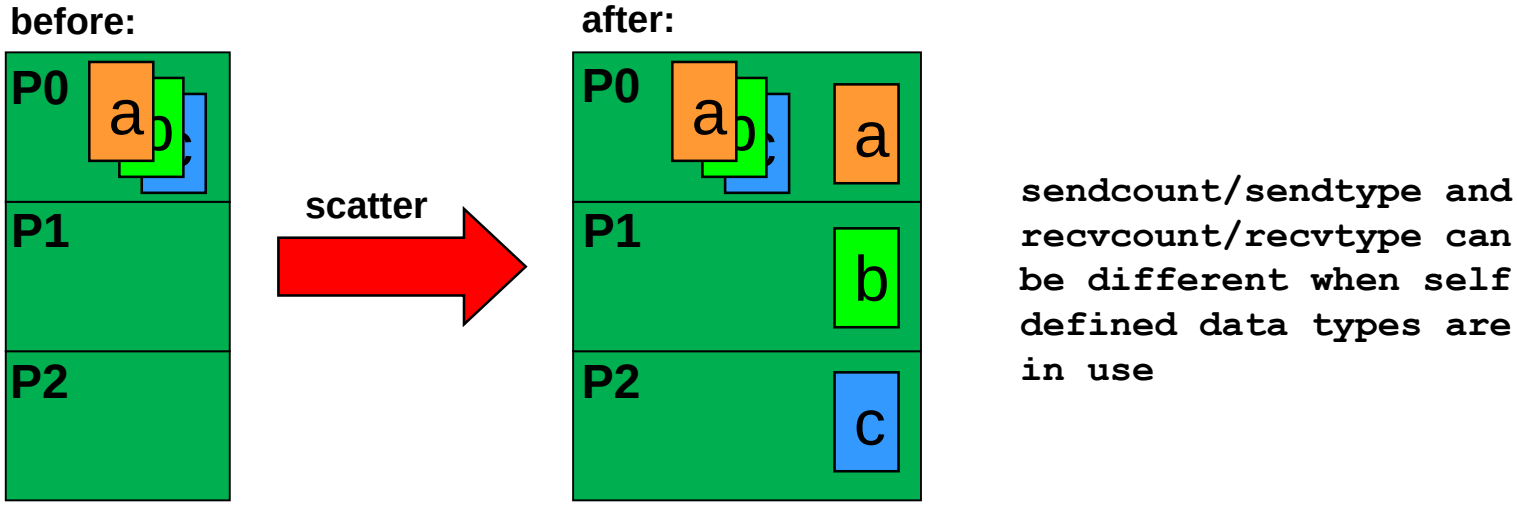
RWTH AACHEN UNIVERSITY

6. Send Data to all Processes, Part II

7. Send Data to all Processes, Part III

MPI_Scatter (One-To-All)

- Scatters an array of data to many processes;
- array index is rank number
  - `int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`



sendcount/sendtype and recvcount/recvtype can be different when self defined data types are in use

Example: Scatter an Array (1 / 2)

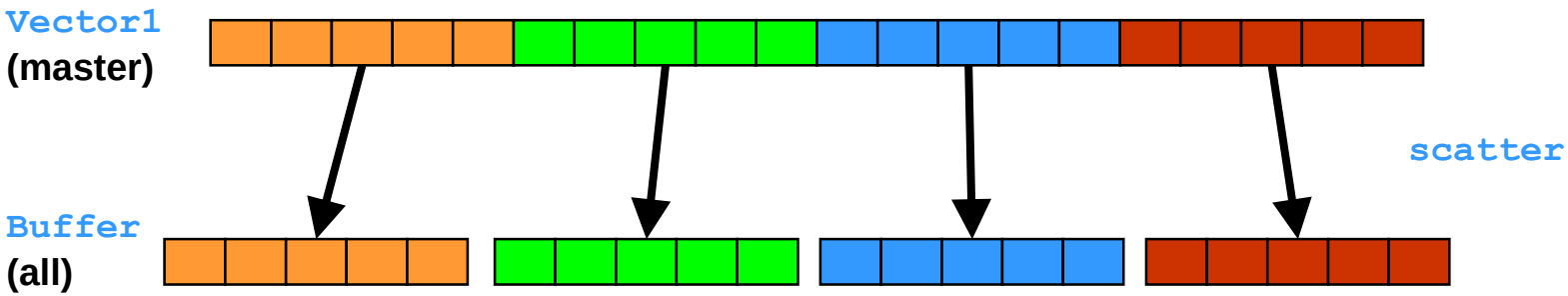
```
[…]
        const int k = 20;
        const int l = k/4;
        int vector1[k], buffer[l];

        // init for master
        if (0 == procRank)
        {
                for (int i=0; i<k; ++i)
                {
                        vector1[i]=i*35;
                }
        }

        MPI_Scatter(vector1, 5, MPI_INT, buffer, 5, MPI_INT, 0, MPI_COMM_WORLD);

        for (int i=0; i<l; ++i)
                printf("Process %i has %i -th element
                        %i\n",procRank,i, buffer[i]);
[…]
```

Example: Scatter an Array (1 / 2)

```
[…]
        const int k = 20;
        const int l = /4;
        int buffer[l];

        // init for master
        if (0 == procRank)
        {
                int vector1[k];
                for (int i=0; i<k; ++i)
                {
                        vector1[i]=i*35;
                }
                MPI_Scatter(vector, 5, MPI_INT, buffer, 5, MPI_INT, 0,
                MPI_COMM_WORLD);
        } else {
                MPI_Scatter(0, 5, MPI_INT, buffer, 5, MPI_INT, 0,
                        MPI_COMM_WORLD);
        }

                for (int i=0; i<l; ++i)
                printf("Process %i has %i -th element
                %i\n",procRank,i, buffer[i]);
[…]
```

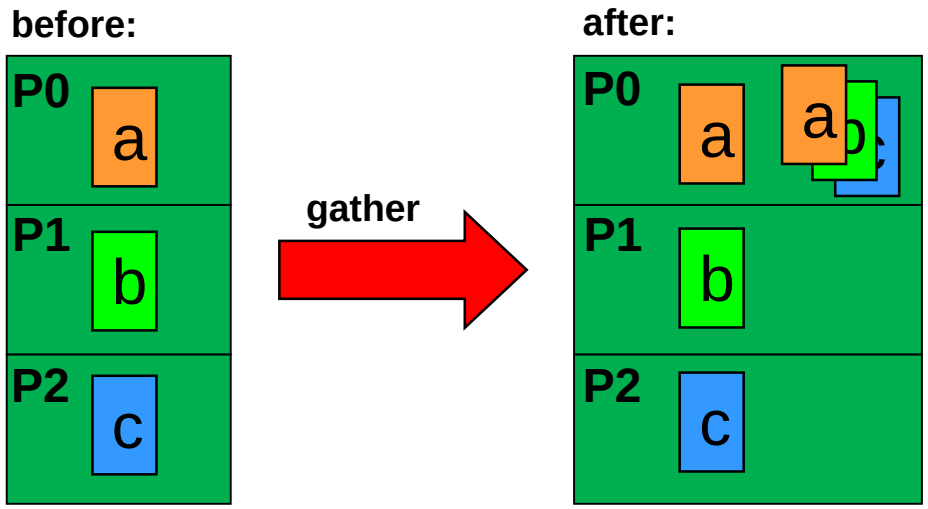
Example: Scatter an Array (2 / 2)

- 4 processes, vector length 20, buffer length $\frac{20}{4} = 5$



MPI_Gather (All-To-One)

- Inverse operation of `MPI_Scatter`
  - `int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount,MPI_Datatype recvtype, int root, MPI_Comm comm)`



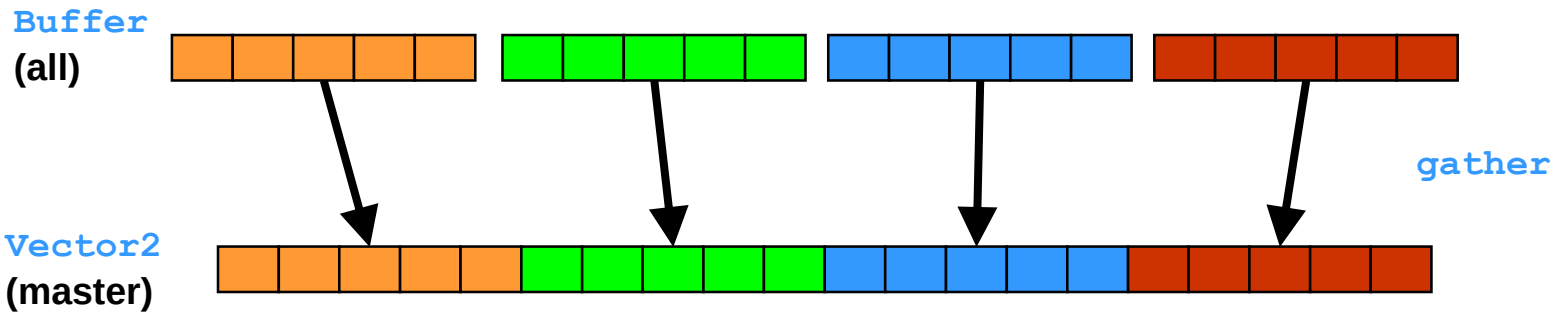
`sendcount/sendtype and recvcount/recvtype can be different when self defined data types are in use`

Example: Gather an Array (1 / 2)

```
[…]
        const int k = 20;
        const int l = k/4;
        int vector2[l], buffer[k];

        // compute!
        for (int i=0; i<l; ++i)
                vector2[i] = i - 10;

        MPI_Gather(vector2,5,MPI_INT,buffer,5,MPI_INT,0,MPI_COMM_WORLD);

        if (0 == procRank)
        {
                for (int i=0; i<k; ++i)
                        printf("vectors %i-th element: %i\n", i, buffer[i);
        }
[…]
```

Example: Gather an Array (2 / 2)

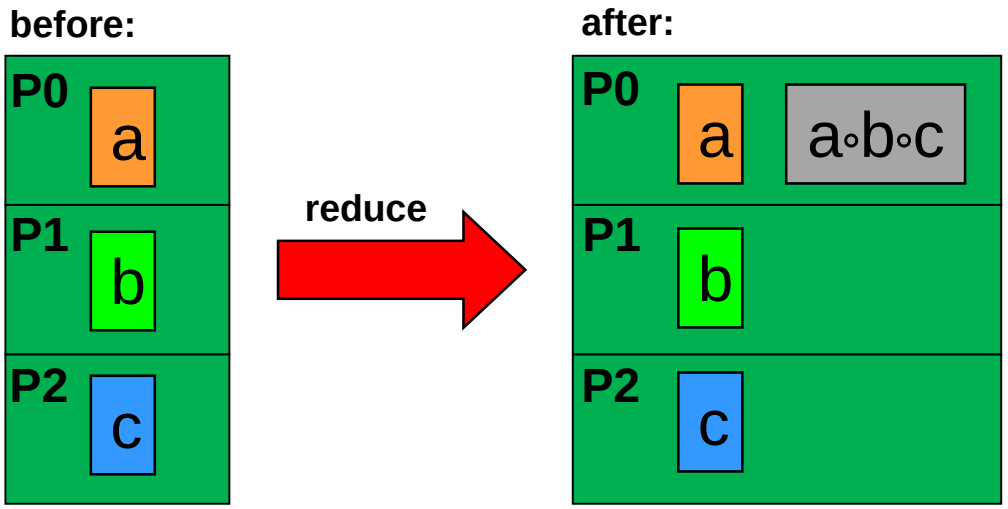
- 4 processes, vector length 20, buffer length $\frac{20}{4} = 5$



Reduce (All-To-One) (1 / 2)

- Collects data and combines it with a reduction operation
- Places the result in one process
  - `int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`

- op: reduce operation



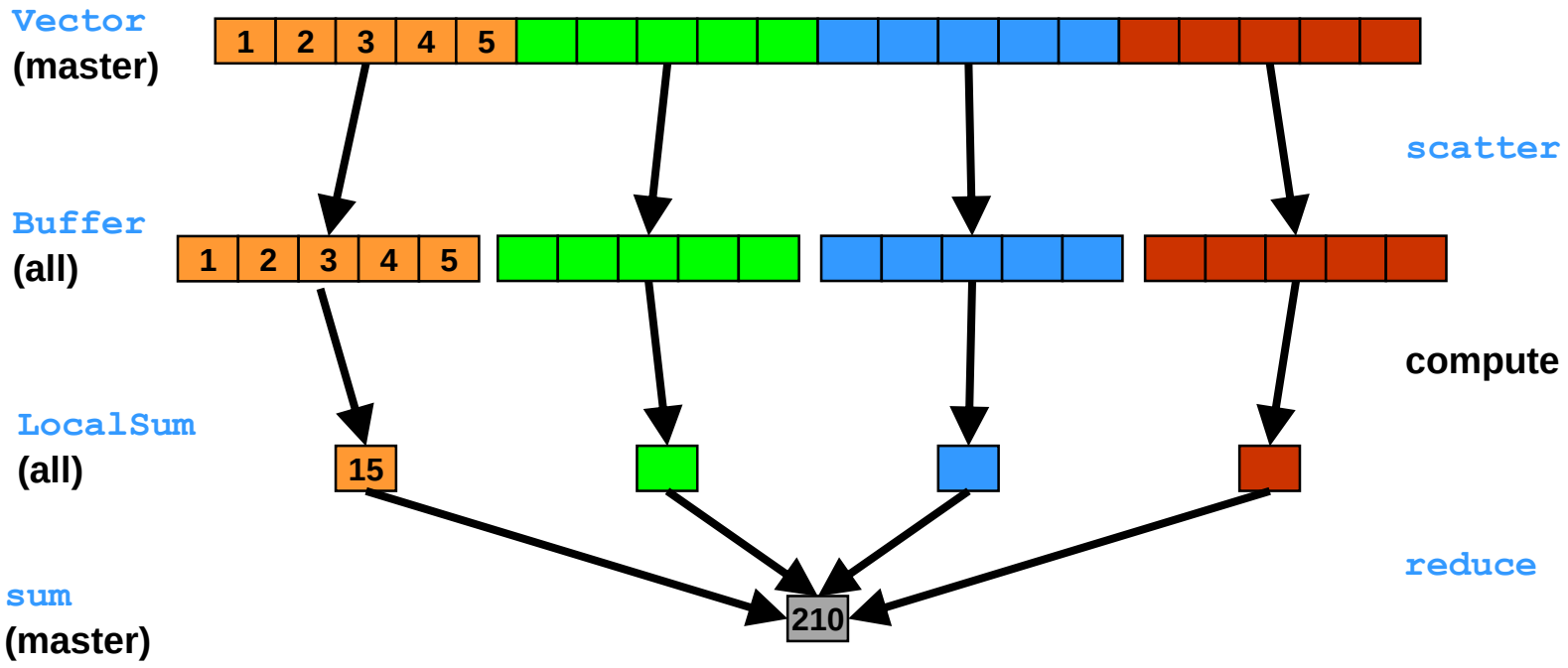
Reduce (All-To-One) (2 / 2)

- **Predefined MPI reduction operations available, such as**
  - `MPI_MAX` maximum over all data
  - `MPI_MIN` minimum over all data
  - `MPI_SUM` sum …
  - `MPI_PROD` product …
  - `MPI_LAND` log. AND …
  - `MPI_LOR` log. OR …
  - …

Example: Reduce (1 / 2)

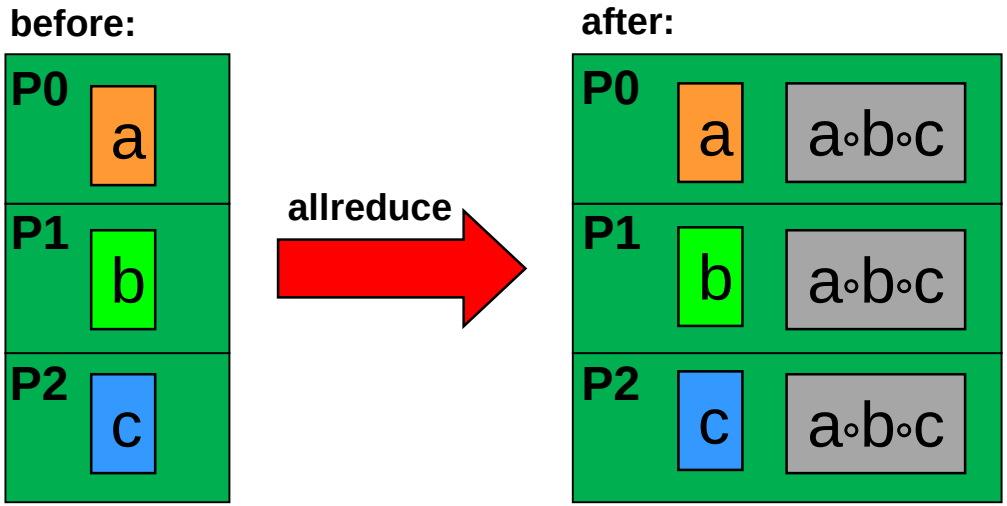
```
[…]
        // compute local results
        int result = procRank * procCount;

        // sum up
        int sum = 0;
        MPI_Reduce(&result, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

        if (0 == procRank)
        {
                printf("sum %d \n",sum);
        }
[…]
```

Example: Reduce (2 / 2)



Allreduce (All-To-All)

- Computes the same result as reduce
- Returns the result in all processes
  - `int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`



Example: Allreduce

```
[…]
        // compute local results
        int result = procRank * procCount;

        // sum up
        int sum = 0;
        MPI_Allreduce(&result, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

        printf("proc %d: sum %d \n",procRank,sum);
[…]
```

Summary

- Barriers
- Time
- Broadcast
- Scatter and Gather
- Reduce and Allreduce
- Some routines not mentioned: Allgather, …

Exercises

8. Integer Array, Part I
9. Integer Array, Part II
10. Integer Array, Part III
11. Integer Array, Part IV
12. Numerical Integration
13. Matrix Vector Multiplication
14. Clean Buggy Code

Summary of Part I

- MPI History and Basics
- Structure of MPI Programs
- Point to Point Communication
- Collective Communication

Exercises

MPI

Exercise 3 – The first MPI-Program

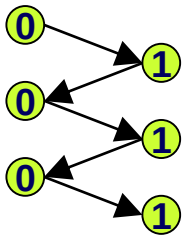
- **Enhance your “Hello World” program and use the MPI lib:**
  - Start with “Hello World” program
  - Initialize and finalize the MPI library.
    - Try what happens if you use a MPI command before initialization or after finalization.
  - Compile and link your program.
    - Use *mpicc your_source –o your_executable*
  - Run your program on three cores simultaneously.
    - Use *mpiexec –n 3 your_executable*
  - Query the rank of the process, determine the number of processes available and print this information.
    - Use more processes than three.

Exercise 4 – Ping Pong

Play Ping Pong:

- Write a program designed for exactly two processes generating

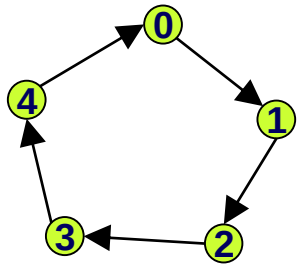
```
hpc@linux-sec2:~/exercise/04-PingPong/solution>
hpc@linux-sec2:~/exercise/04-PingPong/solution> mpiexec -np 2 pingpong |sort
1 sent: Ping from proc 0
2 sent: Pong from proc 1
3 sent: Ping from proc 0
4 sent: Pong from proc 1
5 sent: Ping from proc 0
6 sent: Pong from proc 1
Start[0]/[2]
Start[1]/[2]
hpc@linux-sec2:~/exercise/04-PingPong/solution>
```



- Send and receive alternately, i.e. the first process sends a message and writes “Ping”, the second process receives it and sends it back – “Pong”, etc. Repeat this three times (in a loop!).
- Enhance your program from Exercise 03.
- Send a counter as message data and use it in order to sort output; cf. output above.

Exercise 5 – Send Data to all Processes I

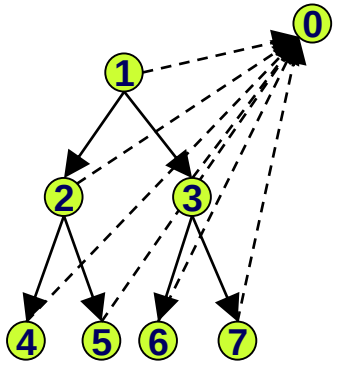
- **Forward a message from one process to the next:**
  - Start with the master or root process (process zero) and send data to the next process. The last process sends its data back to the master.
  - Use an integer as message data.
  - Modify your program from Exercise 04.



- **Measure the Round-Trip time:**
  - Repeat the loop above 100 times and calculate the average time for one loop with 16 processes.

Exercise 6 – Send Data to all Processes II

- **Simulate a Broadcast call:**
  - Strategy: Start with one process (rank 1) and send data to the following two processes (2,3). These processes receive this data and forward it to their successors (4,5 and 6,7). Repeat this for all “intermediate” processes. All worker processes (rank>0) send their data back to the master process.
  - Process zero accepts all messages until all worker processes have sent data.
  - Use an integer as message data.
  - Modify your program from Exercise 05.



- **Measure the Round-Trip time similar to Exercise 05.**

Exercise 7 – Send Data to all Processes III

- **Make a Broadcast:**
  - Send data via Broadcast from master process to all other processes. All processes (rank>0) send its data back to the master as in Exercise 06.
  - The master should accept all incoming messages until all processes have sent their data.
  - Use an integer as message data.
  - Modify your program from Exercise 06.

Measure the Round-Trip time similar to Exercise 05 and 06.

Scan array for numbers – Preparation:

- Serial solution: the program `arraydistI-serial.c´ creates an array of k (here k=50) integer numbers between 0 an 4 and then counts zeros. The output looks as follows:

```
hpc@linux-sec2:~/exercise/09-IntArray-PartI> ./arraydistI-serial
Array: 3 2 1 3 1 3 2 0 1 1 2 3 2 3 3 2 0 2 0 0 3 0 3 1 2 2 2 3 3 3 1 2 2 2 1 3 1 0 3 2 1 1 1 3 0 1 2 0 3 2
Number 0 was found 8 times in the array
hpc@linux-sec2:~/exercise/09-IntArray-PartI>
```

- Extend `arraydistI-serial.c´ such that the master sends the array, i.e. the data vector, to a second process.
- This second process has to determine the size of the incoming message, i.e. the data vector, and has to allocate the memory accordingly.
- Instead of the master the second process prints the array, counts all zeros and prints the result (verification step).

Exercise 9 – Integer Array II

- **Scan array for different numbers simultaneously:**
  - The master process creates an array as in Exercise 08 and sends this vector to all other processes. Choose the random integer number between 0 and #processes-1.
  - All worker processes have to determine the size of the incoming message and allocate the memory accordingly.
  - Additionally, the master sends a number to search for to each worker. For simplicity, use process' rank as search number.
  - Each worker and the master (!) count how many times its data vector contains the respective search number.
  - Each worker sends its result back to the master. Accept all incoming messages as in Exercise 06.
  - Modify your Exercise 08 solution.

Exercise 10 – Integer Array III

- **Scatter array and scan each part for one single number:**
  - The master process creates an array as before but scatters the array to all processes such that the master and each worker only work on a partition of the array. Assume, that the number of array elements is divisible by the number of processes.
  - The master sends the partial array length to all workers before.
  - Each worker searches the same number. Again this search number has to be send to the workers by the master process.
  - Each process counts the occurrence of the number and sends it back to the master as before.
  - Modify your program from Exercise 09 and choose an appropriate send method for each information.

Exercise 11 – Integer Array IV

- **Scatter array, scan each part and reduce results:**
  - Modify your program from Exercise 10 and use the reduce mechanism to sum up all results.

Calculation of Pi:

- According to $\frac{\pi}{4} = \int_0^1 \frac{1}{1+x^2} dx$

  π can be calculated numerically by using quadrature, e.g. with trapezium or midpoint rule (rectangle method); the latter reads:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} f(a + (i - 0.5)h)h \qquad h = \frac{(b-a)}{n}$$

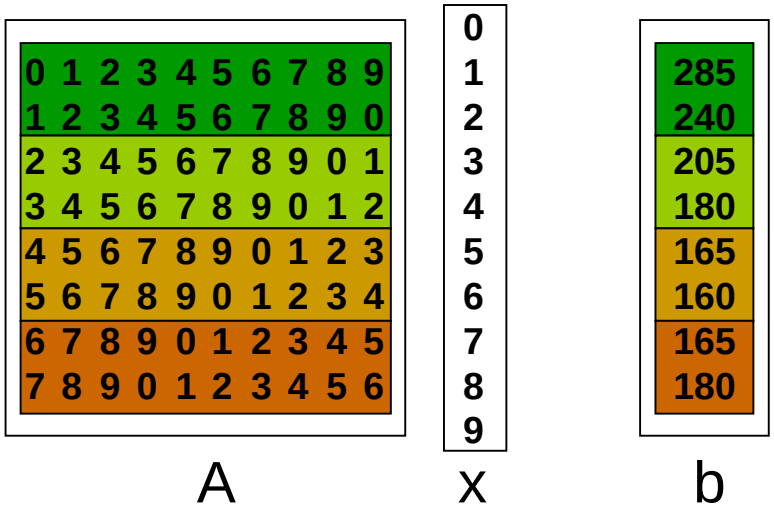
- Divide the integration domain into intervals and distribute the work such that all processes get the same amount of work.
- Use the reduce mechanism to obtain the final sum.
- Start with `calcPI-serial.c´ and compare results.

Exercise 13 – Matrix Vector Multiplication

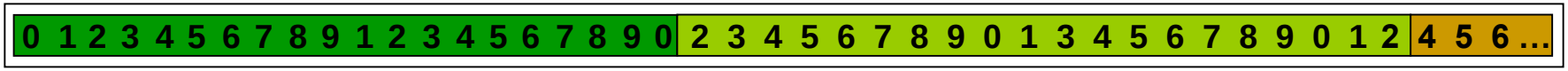
- **Parallelize the standard matrix vector multiplication $Ax = b$:**
  - The master distributes the complete vector x and different parts (rows) of matrix A to workers. Assume that the number of rows is divisible by the process number.
  - Each process computes one part of the result vector b for a sub matrix of A; cf. first figure next slide.
  - The master collects the calculated elements in b and prints the result.
  - Use the specific memory layout of C matrices; cf. figure.
  - Send height and width of matrix and vectors to workers, allocate memory dynamically.
  - Parallelize example `matrixvectorMult-serial.c´.

Parallelize standard matrix vector multiplication – continue:

- Parallelization strategy with four processes:



- Memory layout of matrix A (C language):



Exercise 14 – Clean Buggy Code

- **Find exactly 5 errors violating the syntax and/or MPI standard.**

```
/* This program does the same operation as an MPI_Bcast() but does it using MPI_Send() and MPI_Recv(). Find exactly
5 errors! */

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

int main(int argc, char **argv) {
    int nprocs;          /* the number of processes in the task */
    int myrank;          /* my rank */
    int i, int l = 0;
    int tag    = 42;     /* tag used for all communication */
    int tag2   = 99;     /* extra tag used for what ever you want */
    int data   = 0;      /* initialize all the data buffers to 0 */
    MPI_Status status;   /* status of MPI_Recv() operation */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    if (myrank == 0) {  /* Initialize the data for rank 0 process only.  */
        data = 399;
        for (i = 1; i < nprocs; i++)
            MPI_Send(&data, 1, MPI_BYTE, i, tag, MPI_COMM_WORLD);
    } else {
        MPI_Recv(data, l, MPI_INT, 0, tag2, MPI_COMM_WORLD, &status);
    }
    MPI_Barrier(MPI_COMM_WORLD);

    if (data != 399) fprintf(stdout, "Whoa! The data is incorrect\n");
    else             fprintf(stdout, "Whoa! Got the message ... \n");

    return 0;
}
```