RWTH AACHEN UNIVERSITY

Introduction to Parallel Programming

MATSE-Training

Jens Hollmann
(hollmann@rz.rwth-aachen.de)

WS 2012/13
Version 1.13 (30.11.2012)

Organizational Matters

RWTH AACHEN UNIVERSITY

Times

- Start: 08:15
- End: 17:00
- Lunchbreak: 1 hour, starting between 11.30 am. and 12.30 pm., depends…

Authors

RWTH AACHEN UNIVERSITY

The following slides have been created and revised by the following people:

- Prof. Dr. Hans Joachim Pflug
- Dr. Tatjana Streit
- Dipl.-Inform. Christian Terboven
- Dr. Alexander Voß
- Bastian Küppers, BSc.

Contents

RWTH AACHEN UNIVERSITY

- **Lecture „Introduction to Parallel Programming“**
  - Introduction
  - Java-Threads
  - MPI - Part I – Basics
  - MPI - Part II – Advanced Topics
  - Introduction to OpenMP
- **Exercises**
  - Java-Threads
  - MPI
  - OpenMP
  - Hybrid

RWTH AACHEN UNIVERSITY

Introduction

Computer Architectures, Parallelization at a Glance

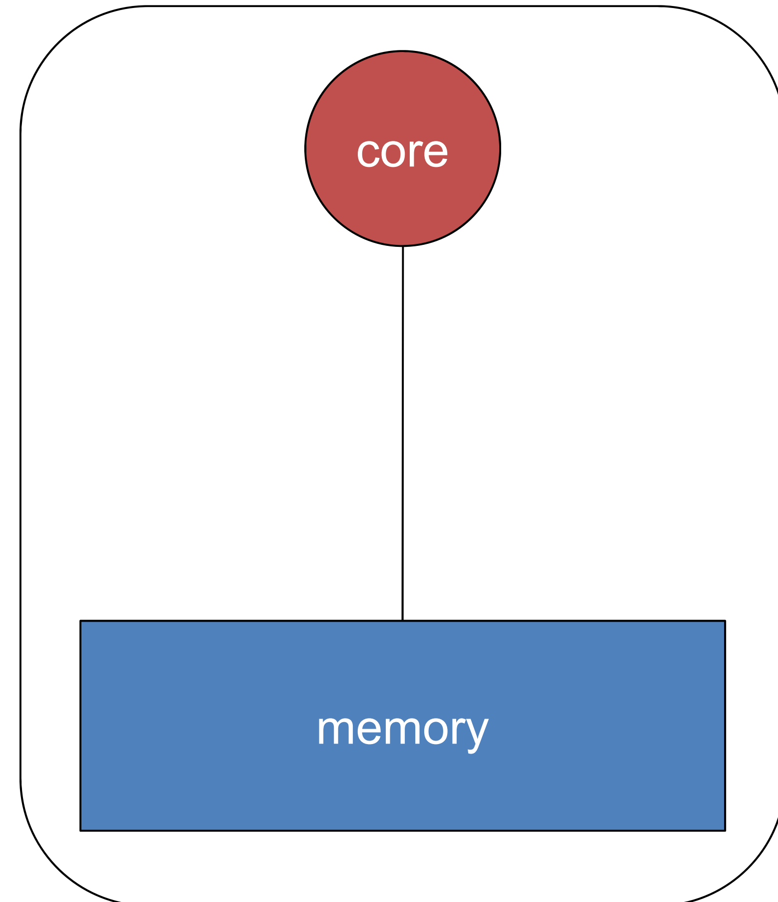
Introduction - Contents

RWTH AACHEN UNIVERSITY

- **Computer Architectures**
  - Basic Computer Architectures
  - Shared-Memory Parallel Systems
  - Distributed-Memory Parallel Systems
- **Parallelization at a Glance**
  - Basic Concepts
  - Parallelization Strategies
  - Prominent Issues

Single Processor Systems (1 / 2)

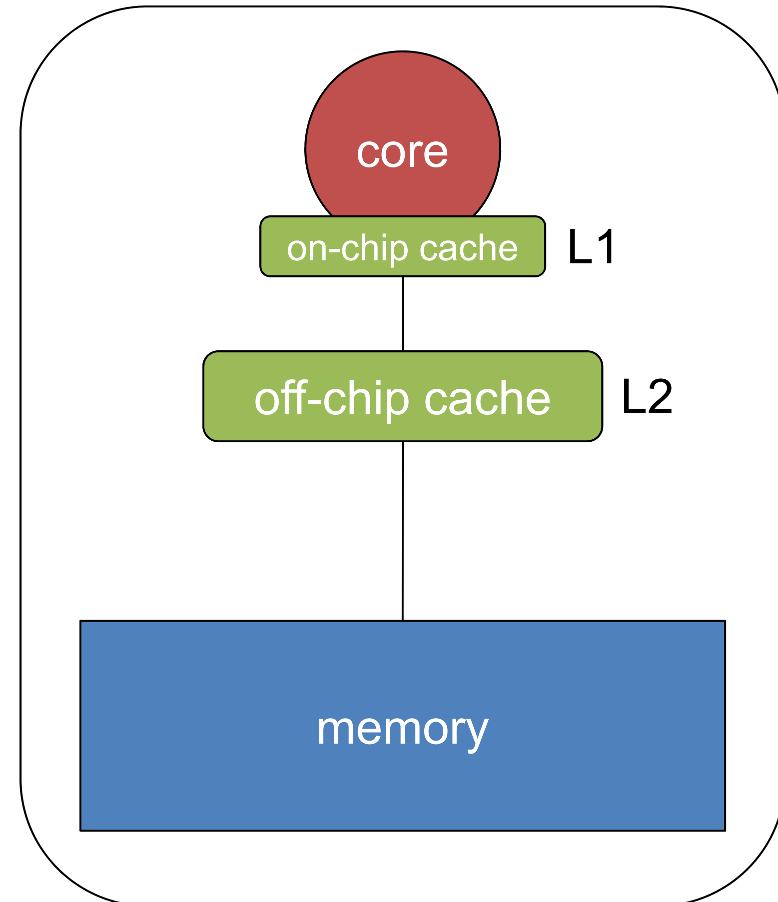
- **Processor**
  - Fetch program from memory
  - Execute program instructions
  - Load data from memory
  - Process data
  - Write results back to memory
- **Main Memory**
  - Store program
  - Store data
- **Input / Output is not covered here!**



Single Processor Systems (1 / 2)

RWTH AACHEN UNIVERSITY

- **CPU**
  - Fast (order of **3.0 GHz**)
- **Main Memory**
  - Slow (order of **0.3 GHz**)
  - Large (order of **GB**)
- **Caches**
  - Fast, but expensive
  - Small (order of **MB**)

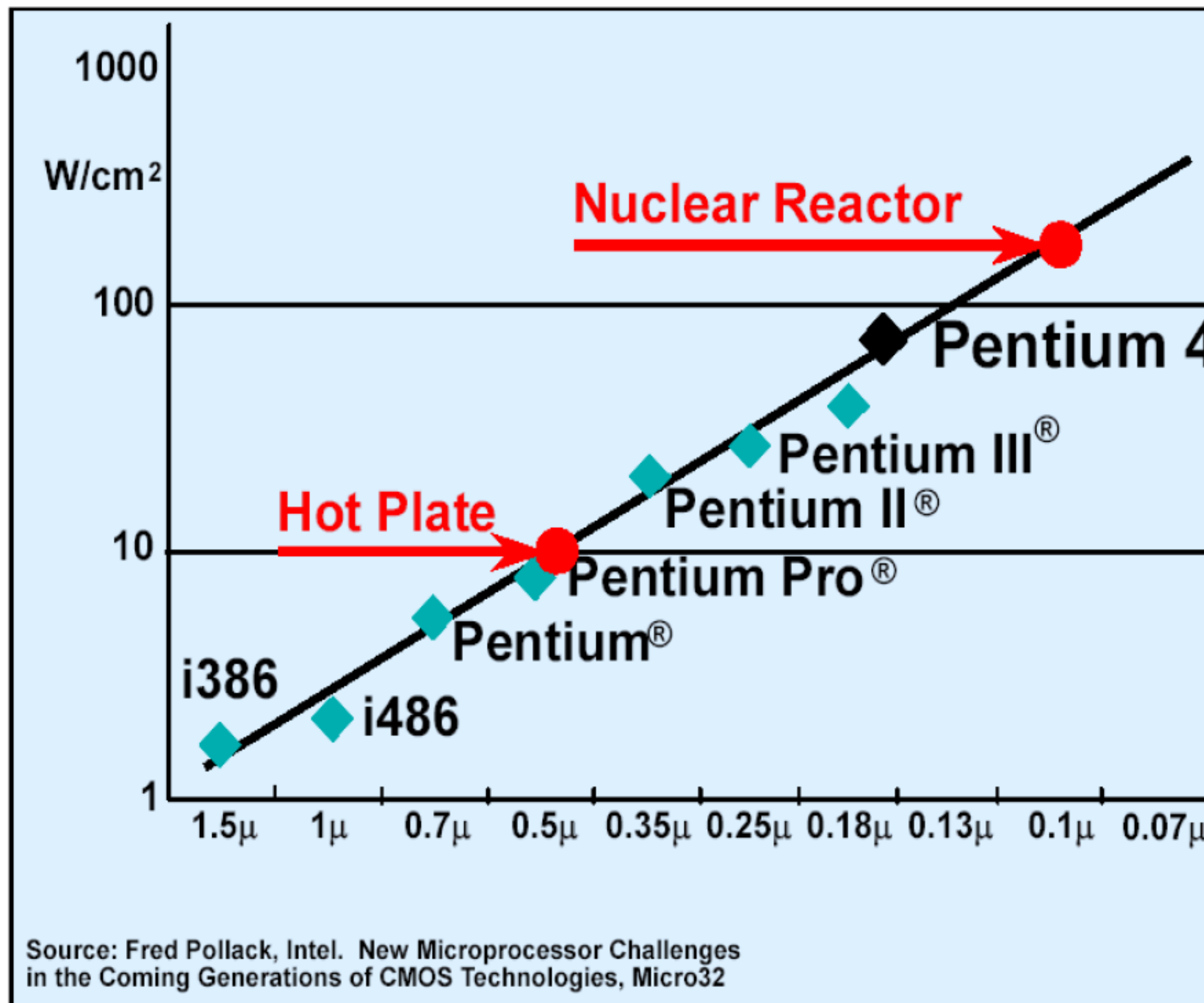


- **Usage of Cache is mandatory for good performance on parallel applications.**

Why aren’t CPUs getting faster anymore?

RWTH AACHEN UNIVERSITY

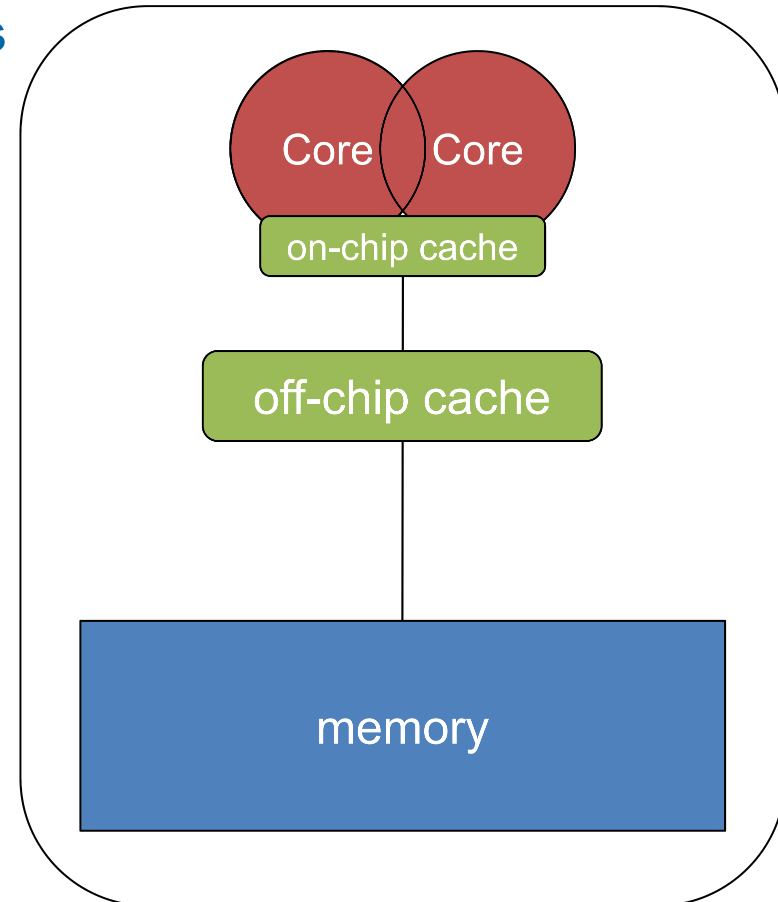
- The CPU would get too hot!



Fast clock cycles make processor chips more expensive, hotter and more power consuming.

Multi-Core Processor Systems

- **Since 2005/2006 dual-core processors are produced for the home user.**
- **Number of cores per chip increases since then**
  - Today: up to 8 cores per chip for a standard CPU
- **Any recently bought PC or Laptop is a multi-core system already.**



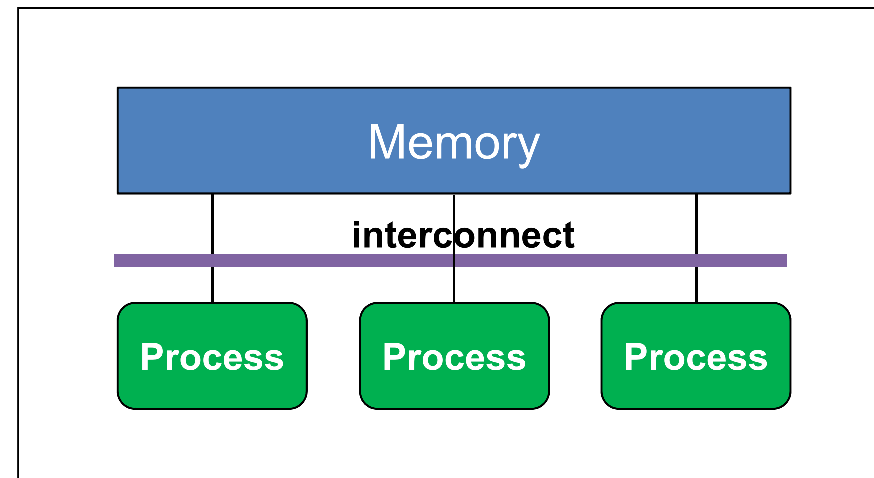
Introduction - Contents

RWTH AACHEN UNIVERSITY

- **Computer Architectures**
  - Basic Computer Architectures
  - Shared-Memory Parallel Systems
  - Distributed-Memory Parallel Systems
- **Parallelization at a Glance**
  - Basic Concepts
  - Parallelization Strategies
  - Prominent Issues

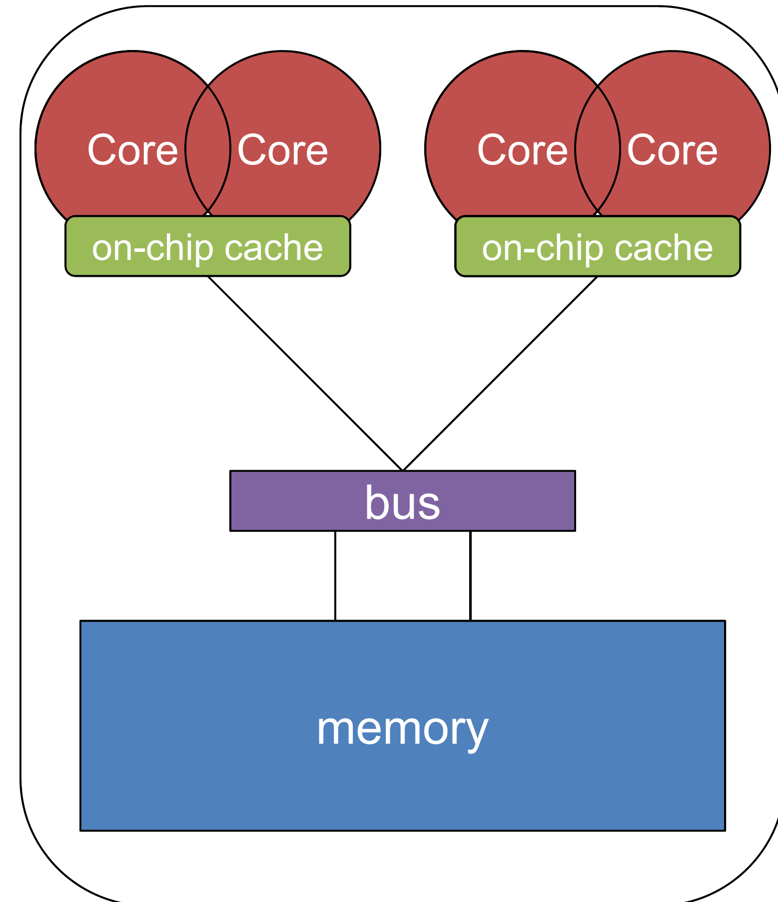
Shared-Memory Parallel Systems

- **Implicit data distribution**
- **Implicit communication**
- **Different types of shared-memory architectures**
- **Programming via …**
  - OpenMP
  - Java-Threads



SMP

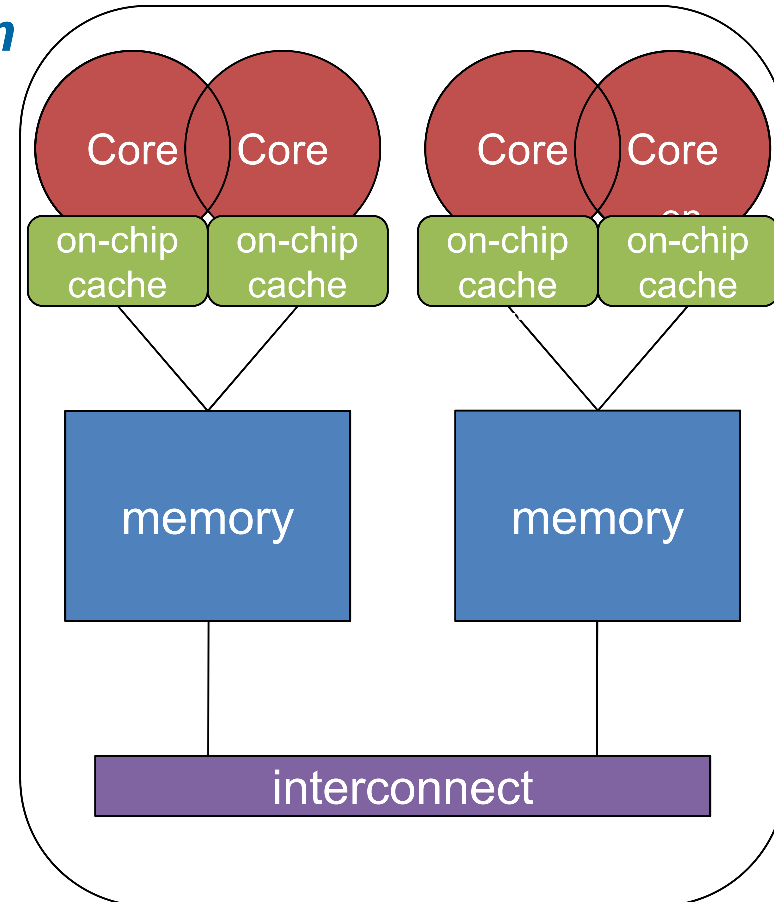
- **Abbr. for *Symmetric Multi Processing***
- **Memory access time is uniform on all cores**
- **Limited scalability**
- **Example: *Intel Woodcrest***
  - Two cores per chip, 3.0 GHz
  - Each chip has 4 MB of L2 cache on-chip, shared by both cores
  - No off-chip cache
  - Bus: Frontsidebus



ccNUMA

RWTH AACHEN UNIVERSITY

- **Abbr. for *cache-coherent Non-Uniform Memory Architecture***
- **Memory access time is non-uniform**
- **Scalable**
- **Example: *AMD Opteron***
  - Two cores per chip, 2.4 GHz
  - Each core has separate 1 MB of L2-cache on-chip
  - No off-chip cache
  - Interconnect: Hypertransport



Cache Coherence (cc)

RWTH AACHEN UNIVERSITY

- **If there are multiple caches not shared by all cores in the system, the system takes care of the cache coherence.**

- **Example:**

```
int a[some_number]; //shared by all threads
thread 1: a[0] = 23;      thread 2: a[1] = 42;
--- thread + memory synchronization (barrier) ---
thread 1: x = a[1];       thread 2: y = a[0];
```

  - Both `a[0]` and `a[1]` are stored in caches of thread 1 and 2
  - Changes to data in the cache is at first only visible for the CPU that modified its cache
  - After synchronization point all threads need to have the same view of (shared) main memory

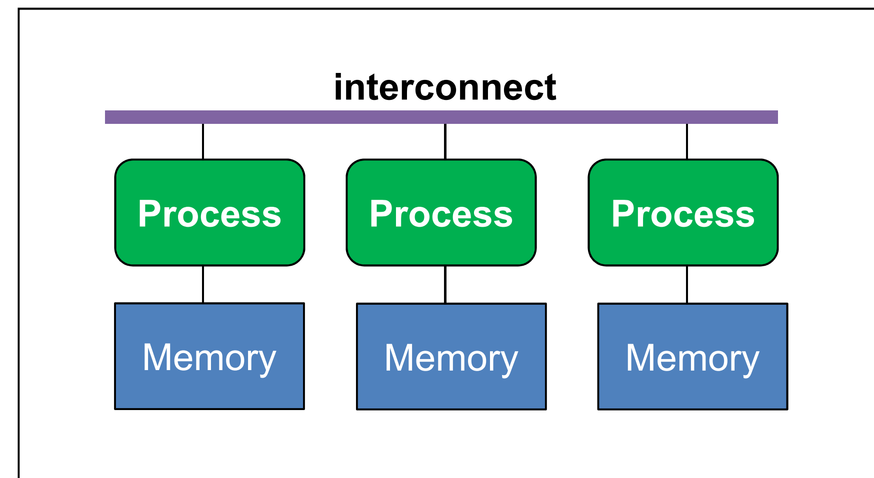
Introduction - Contents

RWTH AACHEN UNIVERSITY

- **Computer Architectures**
  - Basic Computer Architectures
  - Shared-Memory Parallel Systems
  - Distributed-Memory Parallel Systems
- **Parallelization at a Glance**
  - Basic Concepts
  - Parallelization Strategies
  - Prominent Issues

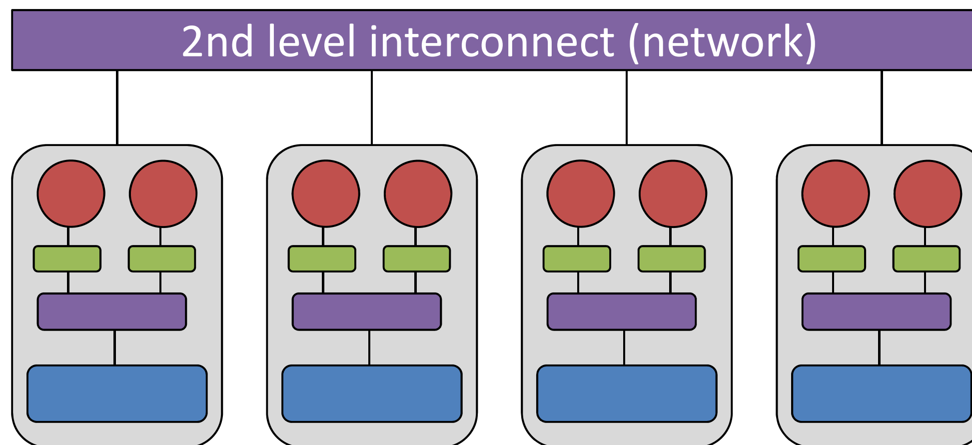
Distributed-memory Parallel Systems

- Explicit data distribution
- Explicit communication
- Scalable
- Programming via MPI



Clusters

- **Various independent computers are connected to each other over a non-cache-coherent *second level interconnect***
  - Infiniband
    - Latency: <= 5 µs
    - Bandwidth: >= 1200 MB/s
  - GigaBit Ethernet
    - Latency: <= 60 µs
    - Bandwidth: >= 100 MB/s



Latency:
Time required to send a message of size zero
(time to setup the communication)

Bandwidth:
Rate at which large messages (>= 2 MB) are transferred

Introduction - Contents

RWTH AACHEN UNIVERSITY

- **Computer Architectures**
  - Basic Computer Architectures
  - Shared-Memory Parallel Systems
  - Distributed-Memory Parallel Systems
- **Parallelization at a Glance**
  - Basic Concepts
  - Parallelization Strategies
  - Prominent Issues

Processes

RWTH AACHEN UNIVERSITY

- **A process is the abstraction of a program in execution**

- **It can be in different states**
  - Running
  - Waiting
  - Ready

- **Each process has its own address-space**
  - → No common variables between processes

Threads

RWTH AACHEN UNIVERSITY

- **A thread is a *lightweight* process**

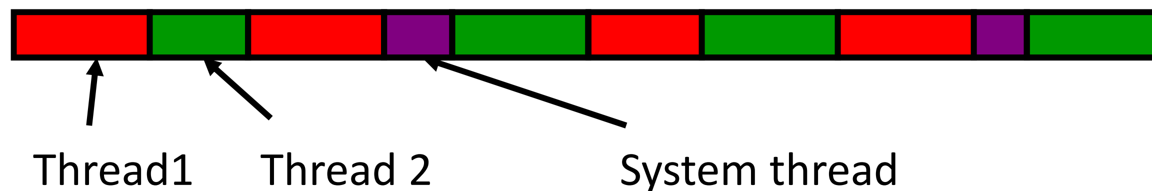
- **In difference to a process, a thread shares the address-space with all other threads of the process it belongs to, but has its own stack.**
  - → Common variables between threads

Process and Thread Scheduling by the OS

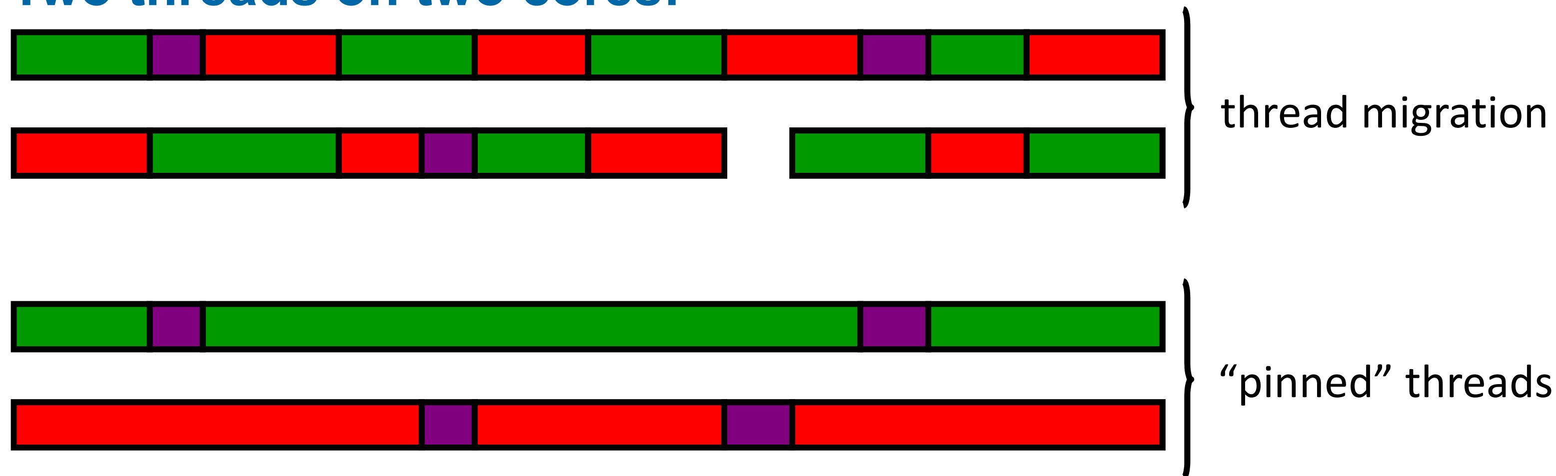
RWTH AACHEN UNIVERSITY

- **Even on a multi-socket / multi-core system you should not make any assumption which process / thread is executed when an where!**

- **Two threads on one core:**



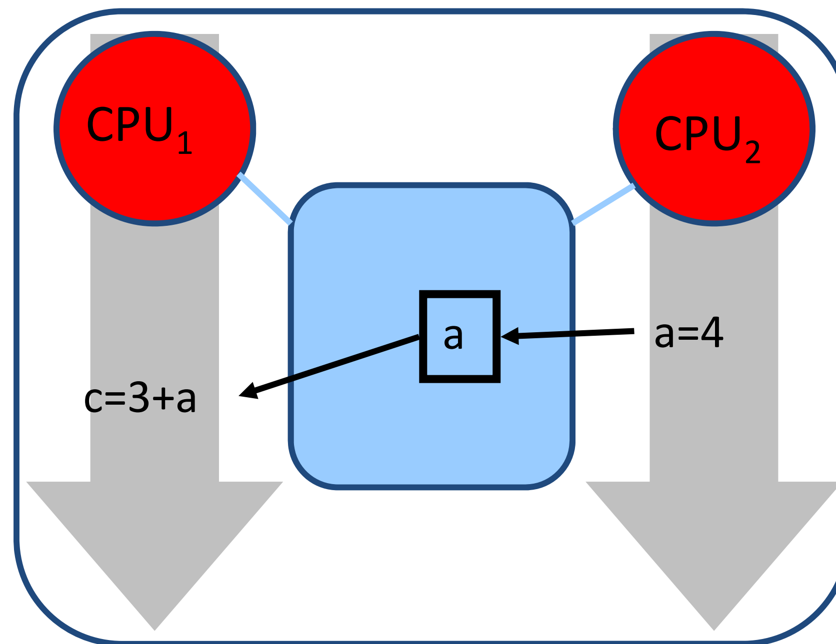
- **Two threads on two cores:**



Shared-memory Parallelization

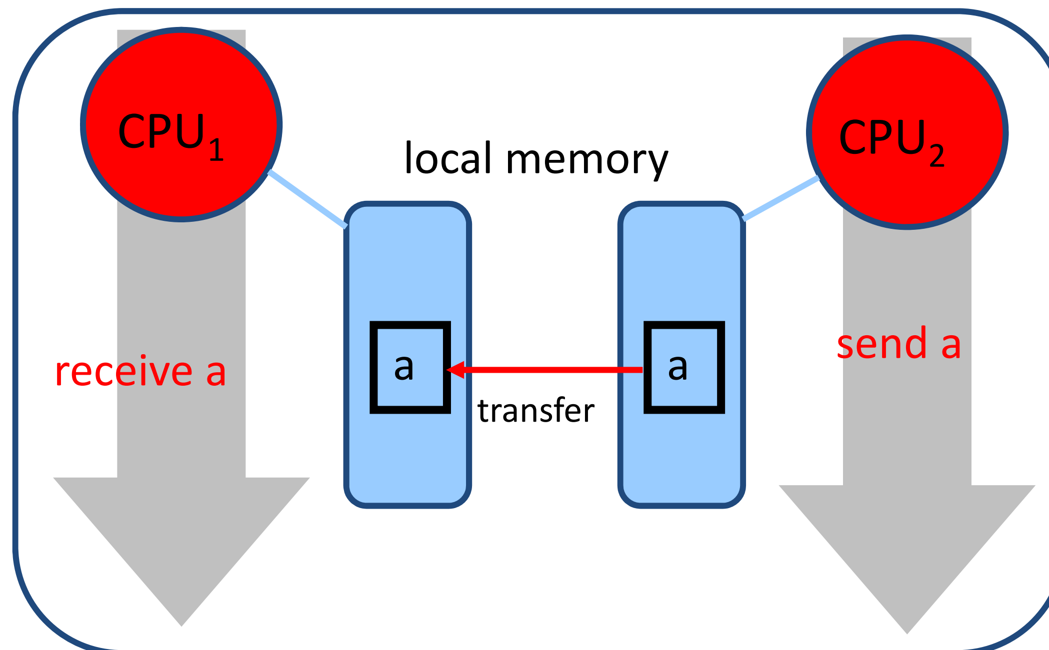
RWTH AACHEN UNIVERSITY

- **Memory can be accessed by several threads running on different cores in a multi-socket / multi-core system**



Distributed-memory Parallelization

- **Each process has its own distinct memory**
- **Communication via *Message Passing***



Introduction - Contents

- **Computer Architectures**
  - Basic Computer Architectures
  - Shared-Memory Parallel Systems
  - Distributed-Memory Parallel Systems

- **Parallelization at a Glance**
  - Basic Concepts
  - Parallelization Strategies
  - Prominent Issues

Speedup and Efficiency (1 / 2)

RWTH AACHEN UNIVERSITY

- **Time using 1 CPU:** $T(1)$
- **Time using p CPUs:** $T(p)$
- **Speedup S:** $S(p) = \frac{T(1)}{T(p)}$
  - Measures how much fast the parallel computation is
- **Efficiency E:** $E(p) = \frac{S(p)}{p}$

Speedup and Efficiency (2 / 2)

- **Example:**
  - $T(1) = 6s,\ T(2) = 4s$

    $\rightarrow S(2) = \frac{6}{4} = \frac{3}{2} = 1.5$

    $\rightarrow E(2) = \frac{1.5}{2} = \frac{3}{4} = 0.75$

- **Ideal case: $T(p) = T(1)/p$**
  - $S(p) = p$
  - $E(p) = 1.0$

Amdahl's Law

RWTH AACHEN UNIVERSITY

- **Describes the influence of the serial part onto scalability (without taking any overhead into account).**

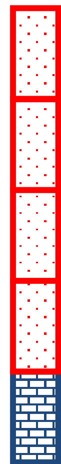
- $S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{f * T(1) + (1-f) * \frac{T(1)}{p}} = \frac{1}{f + \frac{1-f}{p}}$
  - f: serial part ($0 \leq f \leq 1$)
  - $T(1)$: time using 1 CPU
  - $T(p)$: time using p CPUs
  - $S(p)$: speedup; $S(p) = \frac{T(1)}{T(p)}$
  - $E(p)$: efficiency; $E(p) = \frac{S(p)}{p}$

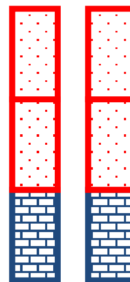
- **It is rather easy to scale to a small number of cores, but any parallelization is limited by the serial part of the program!**

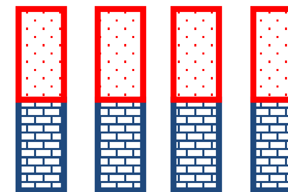
Amdahl's Law illustrated

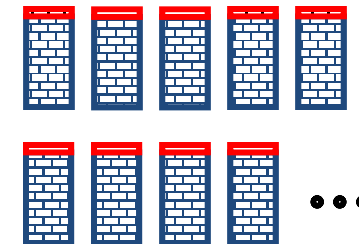
RWTH AACHEN UNIVERSITY

- **If 80% (measured in program runtime) of your work can be parallelized and "just" 20% are still running sequential, then your speedup will be:**







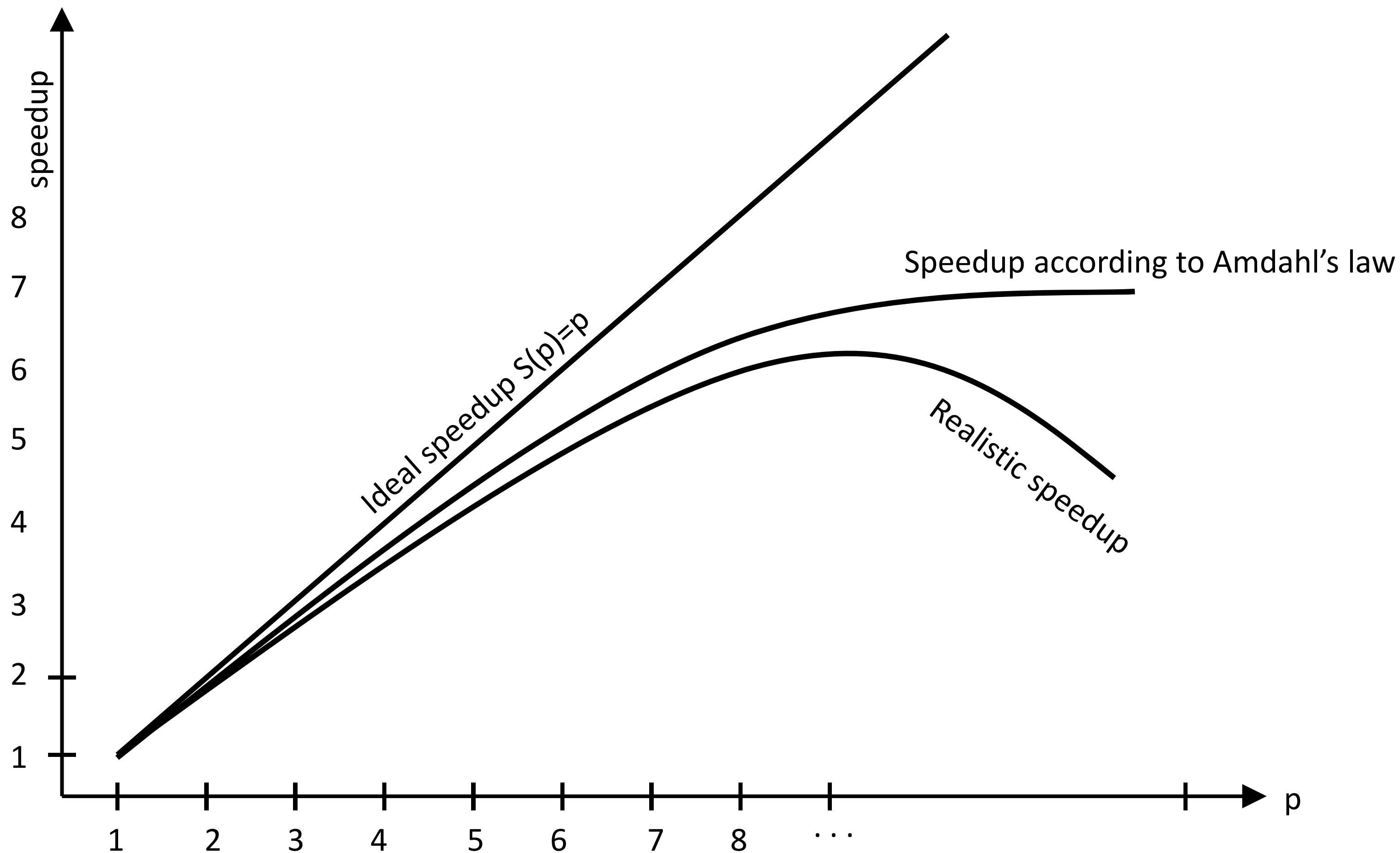


1 processor:	2 processors:	4 processors:	∞ processors:
time: 100%	time: 60%	time: 40%	time: 20%
speedup: 1	speedup: 1.7	speedup: 2.5	speedup: 5

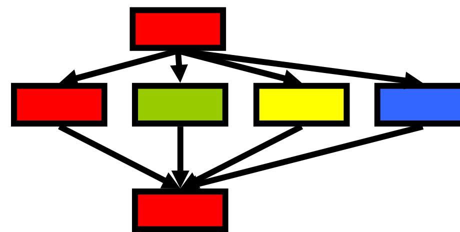
Speedup in Practice

RWTH AACHEN UNIVERSITY

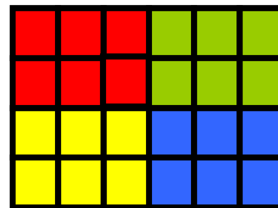
- **After the initial parallelization of a program, you will typically see speedup curves like this:**



- **Chances for concurrent execution:**
  - Look for tasks that can be executed simultaneously (task decomposition)



  - Decompose data into distinct chunks to be processed independently (data decomposition)

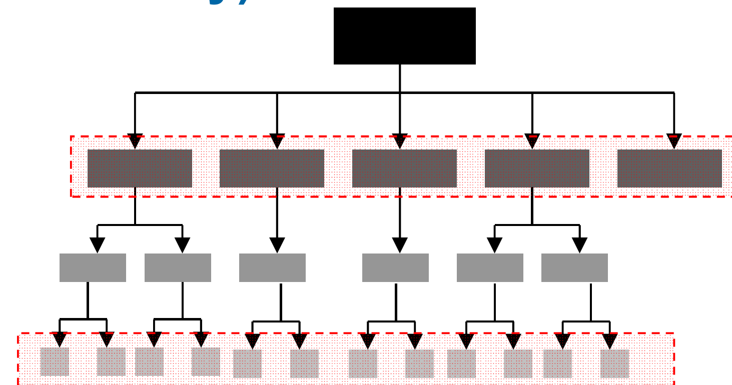


Granularity

RWTH AACHEN UNIVERSITY

- **Parallelization on a High Level (low granularity)**
  - Chances of low synchronization / communication cost
  - Danger of load balancing issues

- **Parallelization on a Low Level (high granularity)**
  - Danger of high synchro-nization / communication cost
  - Chances of avoiding load balancing issues



- **Compute intensive programs may employ multiple levels of parallelization, maybe even with multiple parallelization paradigms (hybrid parallelization).**

Introduction - Contents

RWTH AACHEN UNIVERSITY

- **Computer Architectures**
  - Basic Computer Architectures
  - Shared-Memory Parallel Systems
  - Distributed-Memory Parallel Systems
- **Parallelization at a Glance**
  - Basic Concepts
  - Parallelization Strategies
  - Prominent Issues

Issues in Parallel Programming: Overview

RWTH AACHEN UNIVERSITY

- **You can still run into all issues of Serial Programming ☹ !**
- **Additional issues:**
  - Is your parallelization correct?
  - It is harder to debug parallel code than serial code!
- **Specific issues of Parallel Programming:**
  - Introduction of overhead by parallelization
  - Data Races / Race Conditions
  - Deadlocks
  - Load Balancing
  - Serialization
  - Irreproducibility / Different numerical results

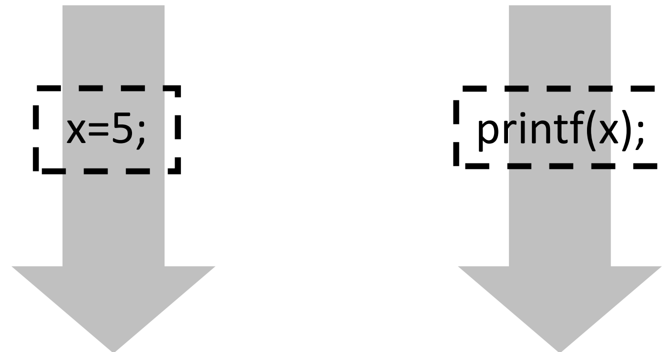
Parallelization Overhead

RWTH AACHEN UNIVERSITY

- **Overhead introduced by the parallelization:**
  - Time to start / end / manage threads
  - Time to send / exchange data
  - Time spent in synchronization of threads / processes
- **With parallelization:**
  - The total CPU time increases,
  - The Wall time decreases,
  - The System time stays the same.
- **Efficient parallelization is about minimizing the overhead introduced by the parallelization itself!**

Data Races / Race Conditions

- **Data Race: Concurrent access of the same memory location by multiple threads without proper synchronization**
  - Let x be initialized with 1



  - Depending on which thread is faster, you will see either 1 or 5
  - Result is nondeterministic (i.e. depends on OS scheduling)

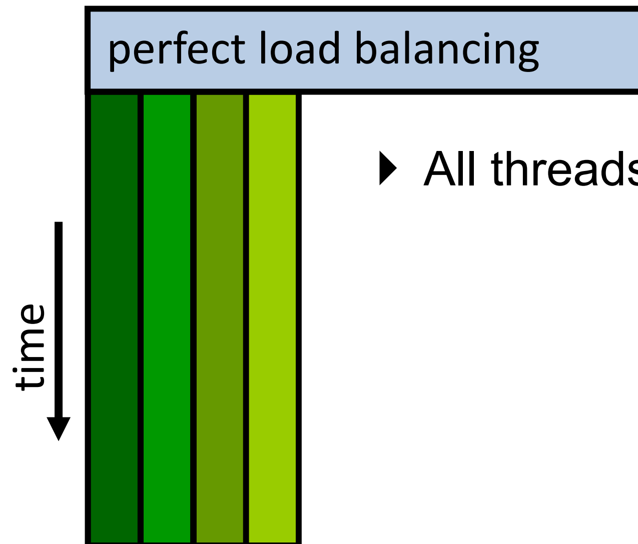
- **Data Races (and how to detect and avoid them) will be covered in more detail later!**

Deadlock

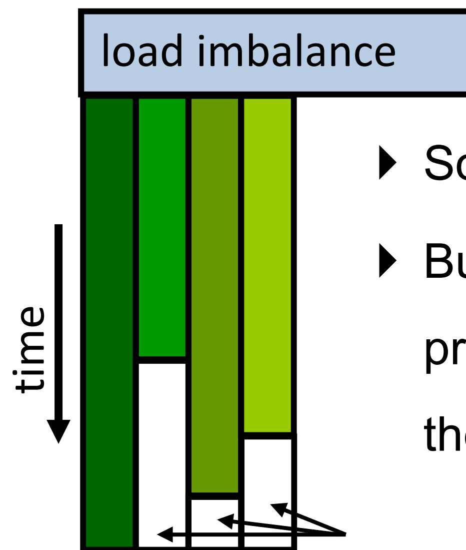
RWTH AACHEN UNIVERSITY

- **When two or more threads / processes are waiting for another to release a resource in a circular chain, the program appears to „hang“:**

Load Balancing



- All threads / processes finish at the same time

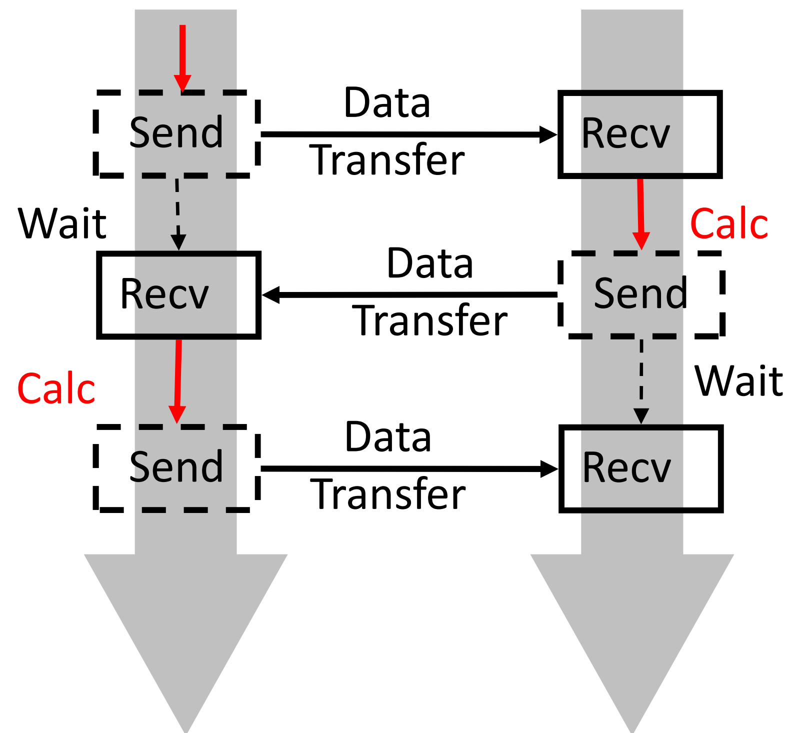


- Some threads / processes take longer than others
- But: All threads / processes have to wait for the slowest thread / process, which is thus limiting the scalability

Serialization

- **Serialization: When threads / processes wait „too much“**
  - Limited scalability, if at all

- **Simple (and stupid) example:**



Contents

RWTH AACHEN UNIVERSITY

- **Lecture „Introduction to Parallel Programming“**
  - Introduction
  - Java-Threads
  - MPI - Part I – Basics
  - MPI - Part II – Advanced Topics
  - Introduction to OpenMP
- **Exercises**
  - Java-Threads
  - MPI
  - OpenMP
  - Hybrid

Java-Threads

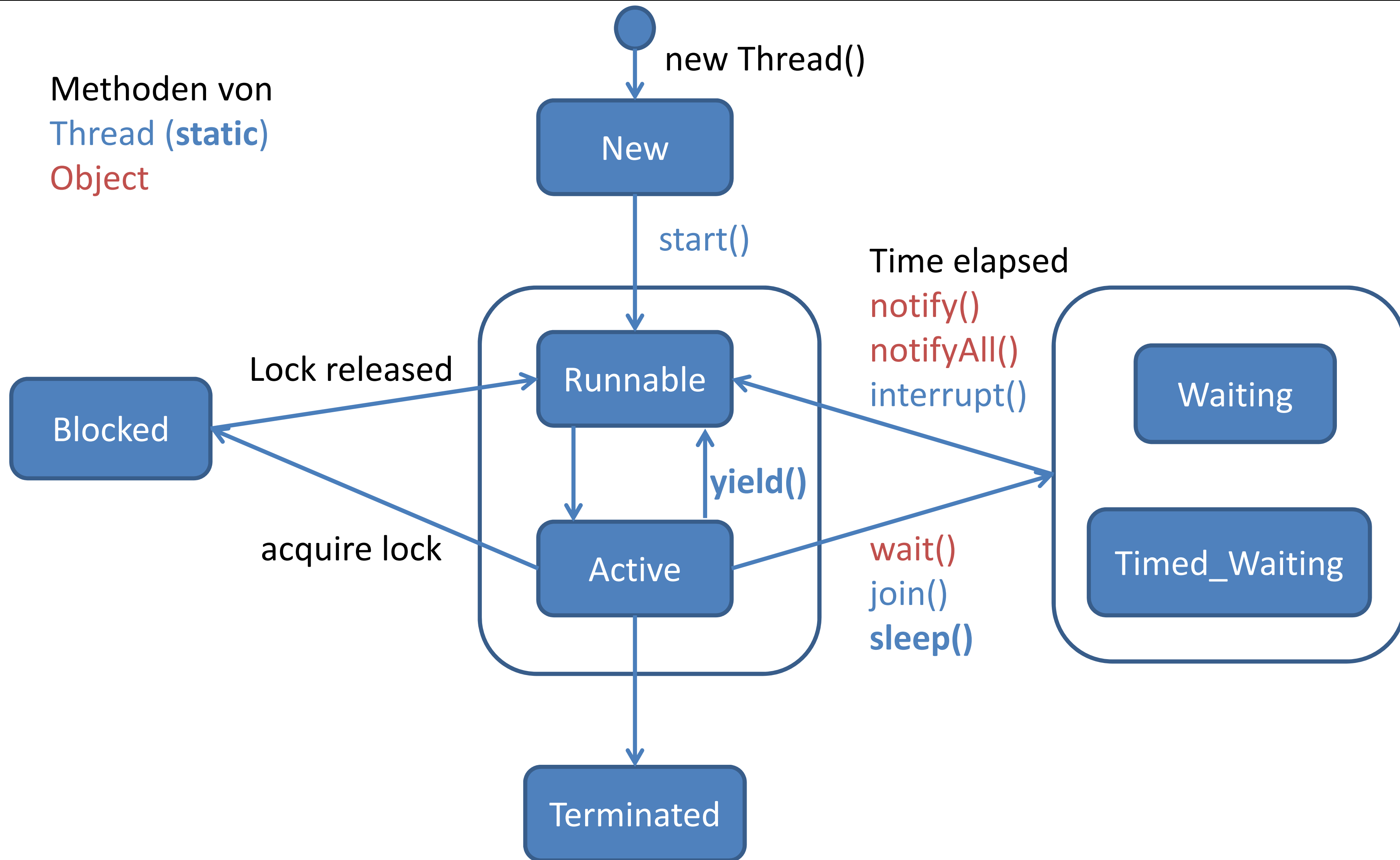
Thread-Parallelization in Java

State of Java Threads

RWTH AACHEN UNIVERSITY

- **A Thread is in either on of the following states**
  - New
  - Runnable
  - Active
  - Blocked
  - Waiting (Timed-Waiting)
  - Terminated

State of Java Threads



Starting a Thread in Java

RWTH AACHEN UNIVERSITY

- **A thread has to know in which line of code it starts**
- **Idea**
  - The new thread calls a method
  - The thread is destroyed after the method has ended
- **Problem**
  - A function pointer would be good, but since Java has no function pointers, there is another method:
  - Calling the native Thread-class with an own object, implementing the Runnable-Interface

Starting a Thread in Java - Runnables

RWTH AACHEN UNIVERSITY

```
public class MyRunnable implements Runnable
{
   public void run()
   {
     // do something useful
   }
}

[...]

Thread t = new Thread(new MyRunnable());
t.start();

[...]
```

Starting a Thread in Java – Live Demo

RWTH AACHEN UNIVERSITY

- **ThreadBasics - startingThreads**

Ending a Thread in Java

RWTH AACHEN UNIVERSITY

- **A thread will destroy itself when the method, that it was executing, is over**

- **Question**
  - Is there a way to wait unless a thread finishes?

- **Answer**
  - Yes!

```
Thread t = new Thread (new MyRunnable());
t.start();
// do something useful
[...]
t.join(); // wait for thread
```

Ending a Thread in Java – Live Demo

RWTH AACHEN UNIVERSITY

- **ThreadBasics - endingThreads**

Motivation for synchronizing Threads

RWTH AACHEN UNIVERSITY

- **To avoid race conditions, it's sometimes necessary to synchronize threads**
  - Synchronization means to actively affect the order of the threads execution

- **There are several methods to realize a synchronization**
  - Atomic operations / atomic data types
  - Mutex locks
  - Barriers
  - …

Race Conditions – Live Demo

RWTH AACHEN UNIVERSITY

- **ThreadSynchronisation1 - withoutSynchronisation**

Atomic operation / Atomic data type

RWTH AACHEN UNIVERSITY

- **An atomic operation is a non-interruptible operations**
  - No other thread or process can perform an operation, while the atomic operation is executed
- **An atomic data type is a data type which operations are atomic**
  - For example `AtomicInteger` in Java
- **Example**

```
AtomicInteger atomic = new AtomicInteger(5);
int nonAtomic = atomic.addAndGet(10);
// nonAtomic is now 15
```

Atomic data type – Live Demo

RWTH AACHEN UNIVERSITY

- **ThreadSynchronisation1 - atomicDatatypes**

Mutex Lock (1)

RWTH AACHEN UNIVERSITY

- A **mutex lock** (*abbr. for* mut*ual* ex*clusion*) takes care for only one thread entering a certain part of the code (***critical region***) at a time

- Example

```
ReentrantLock mutex = new ReentrantLock();
mutex.lock();
// do something useful }
mutex.unlock();
```

- The code between `lock()` and `unlock()` is executed by only one thread at a time

Mutex Lock – Live Demo

RWTH AACHEN UNIVERSITY

- **ThreadSynchronisation1 – mutexLock – reentrantLock**

Mutex Lock (2)

- **A mutex can also be used with a `synchronized`-block.**
  - A `synchronized`-Block needs an object as mutex
  - Also the `this`-object can server as mutex
  - All `synchronized`-Blocks, that share the same object, thus the object with the same memory address, belong together

- **Example**

```
SomeObject mutex = new SomeObject();
synchronized( mutex );
{
     // do something useful }
}
```

Mutex Lock (3)

RWTH AACHEN UNIVERSITY

```
public synchronized void func()
{
   // do something useful }
}
```

is the same as

```
public void func()
{
   synchronized(this)
   {
      // do something useful
   }
}
```

Mutex Lock – Live Demo

RWTH AACHEN UNIVERSITY

- **ThreadSynchronisation1 – mutexLock – synchronizedBlock**

Pipe

RWTH AACHEN UNIVERSITY

- **A pipe (also called queue) is an uni- or bidirectional datastream, that works with the FIFO (*first in, first out*) principle**

- **Example**

```
LinkedBlockingQueue < Integer > queue =
     new LinkedBlockingQueue < Integer >();

// Thread a
int t = queue.take (); // blocks if queue is empty

// Thread b
int p = 5;
queue.put(p)
```

Pipe – Live Demo

RWTH AACHEN UNIVERSITY

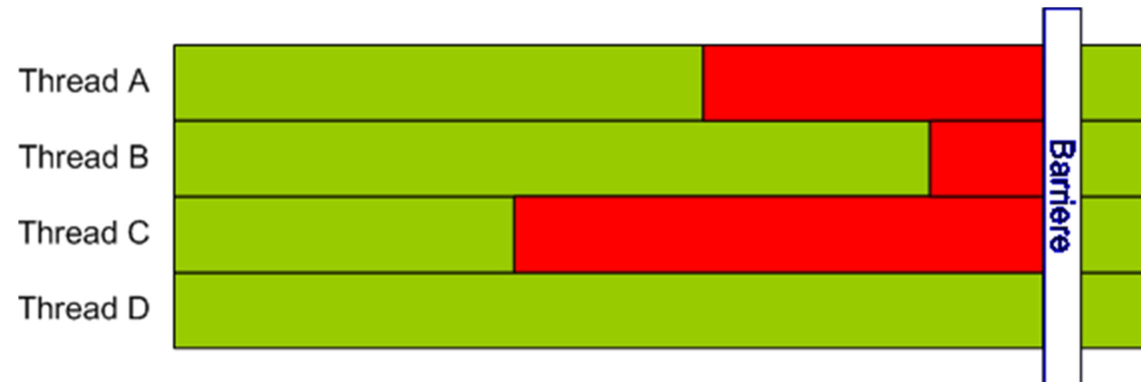
- **ThreadSynchronisation2 – pipe**

Barrier (1)

- **A barrier blocks all threads arriving at the barrier until a certain number of threads has reached the barrier**
  - The number of waiting threads is adjustable
  - When the last thread reaches the barrier, all threads are released
    - The barriers “breaks”.
- **Example**

```
int n = 4;
CyclicBarrier barrier = new CyclicBarrier(n);

try
{
    barrier.await();
}
catch( Exception e) { /* do something /* }
```

Barrier (2)



Barrier – Live Demo

- **ThreadSynchronisation2 – barrier**

Threadpool

RWTH AACHEN UNIVERSITY

- A **future** is an object which acts as placeholder for data, that will be available in the future

- Example

```
ExecutorService pool =
     Executors.newFixedThreadPool(5);

Callable <String > task = new TaskImplementation();
Future <String > f = pool.submit( task );
// Do something useful…
String result = f.get (); // blocks if necessary
```

Threadpool – Live Demo

RWTH AACHEN UNIVERSITY

- **ThreadSynchronisation2 – threadPool – runnables**

Future

RWTH AACHEN UNIVERSITY

- **A threadpool is a group of threads**
  - Each thread in the pool sleeps, until it gets a task
  - After finishing a task a thread returns to the pool
  - New tasks are queued if all threads are busy

- **Example**

```
ExecutorService pool =
     Executors.newFixedThreadPool (5);

Runnable task = new TaskImplementation();
pool.execute( task );
```

Future – Live Demo

RWTH AACHEN UNIVERSITY

- **ThreadSynchronisation2 – threadPool – futures**

Contents

RWTH AACHEN UNIVERSITY

- **Lecture „Introduction to Parallel Programming“**
  - Introduction
  - Java-Threads
  - MPI - Part I – Basics
  - MPI - Part II – Advanced Topics
  - Introduction to OpenMP
- **Exercises**
  - Java-Threads
  - MPI
  - OpenMP
  - Hybrid

MPI – Part I
Basics

General Information, Communication via Message Passing

MPI – Part I - Contents

RWTH AACHEN UNIVERSITY

- **Introduction**
- Structure of MPI Programs
- Groups and Communicators
- Point to Point Communication
- MPI Data Types
- Collective Communication
- Summary of Part I

What is MPI? (1 / 3)

RWTH AACHEN UNIVERSITY

- **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 2.2, September 2009**
- **Different implementations exist:**
  - typically provided as libraries with C, C++, Fortran bindings
  - free MPI implementations:
    - MPICH2 http://www.mcs.anl.gov/research/projects/mpich2
    - LAM/MPI (OpenMPI) http://www.lam-mpi.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)

RWTH AACHEN UNIVERSITY

- **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**

MPI History

RWTH AACHEN UNIVERSITY

- **Version 1.0 (1994) Fortran77 and C supported, 129 routines**
- **Version 1.1 - 1.3 corrections and clarifications**
- **Version 2.0 (1997) 193 routines, major enhancements**
  - one-sided communications
  - parallel I/O
  - dynamic process generation
  - Fortran90 and C++ support
  - thread safety
- **Version 2.1 - 2.2 (2009) corrections and clarifications**
- **Major work of current MPI Forum is preparation of MPI 3.0**

Other Computer Languages

RWTH AACHEN UNIVERSITY

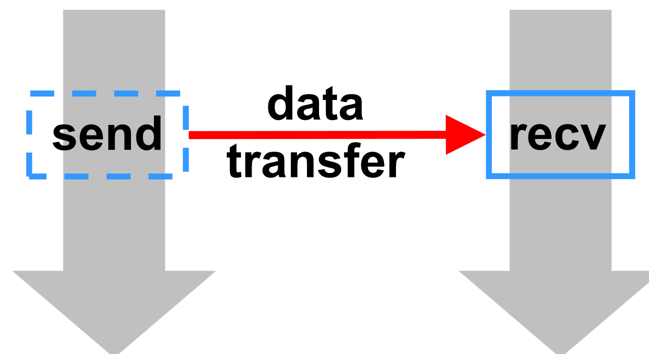
- **Official standard defines language bindings for Fortran, C, and C++; however, in 2.2 C++ binding is deprecated …**

- **Several unofficial bindings for other languages exist:**
  - Java: mpiJava, MPJ
  - .NET: PureMPI.NET
  - Python: ScientificPython, MYMPI, pyMPI
  - Ruby: ruby-mpi
  - Matlab: CMTM

Basic Concepts of 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

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

- **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 2.2

Further Information

RWTH AACHEN UNIVERSITY

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

- **mpiJava Interface**
  http://www.hpjava.org/mpiJava.html

- **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

RWTH AACHEN UNIVERSITY

- Introduction
- **Structure of MPI Programs**
- Groups and Communicators
- Point to Point Communication
- MPI Data Types
- Collective Communication
- Summary of Part I

Example: Hello World

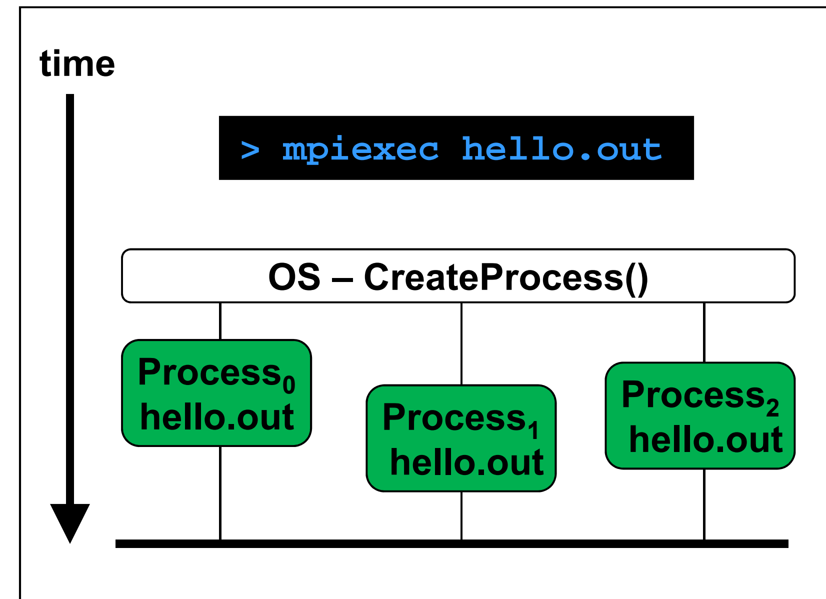
RWTH AACHEN UNIVERSITY

```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

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

```
#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;
}
```

Exercises

1. **Check Development Environment**

2. **Hello World**

MPI – Part I - Contents

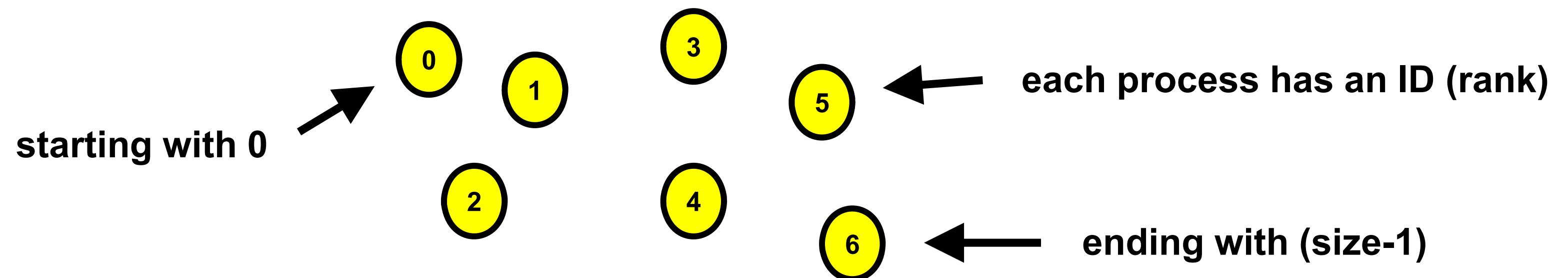
RWTH AACHEN UNIVERSITY

- Introduction
- Structure of MPI Programs
- **Groups and Communicators**
- Point to Point Communication
- MPI Data Types
- Collective Communication
- Summary of Part I

Group

RWTH AACHEN UNIVERSITY

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

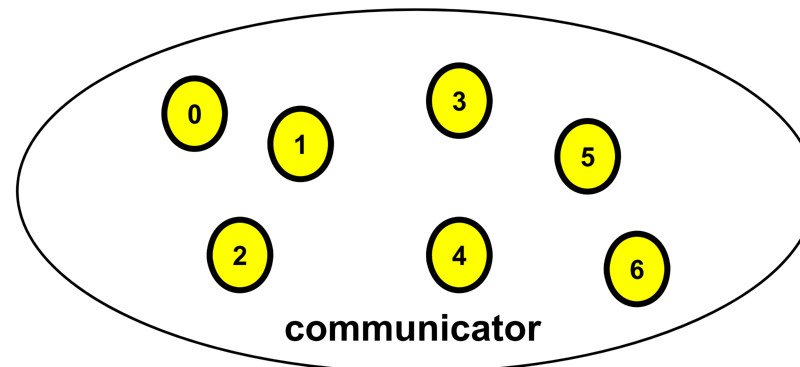


- One process can belong to two or more groups
- Groups are dynamic objects in MPI and can be created and destroyed during program execution (Part II)

Communicator

RWTH AACHEN UNIVERSITY

- 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

RWTH AACHEN UNIVERSITY

- `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

- **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

RWTH AACHEN UNIVERSITY

```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)

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

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

3. The first MPI Program

- Introduction
- Structure of MPI Programs
- Groups and Communicators
- **Point to Point Communication**
- **MPI Data Types**
- Collective Communication
- Summary of Part I

Communication Issues

RWTH AACHEN UNIVERSITY

- **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?

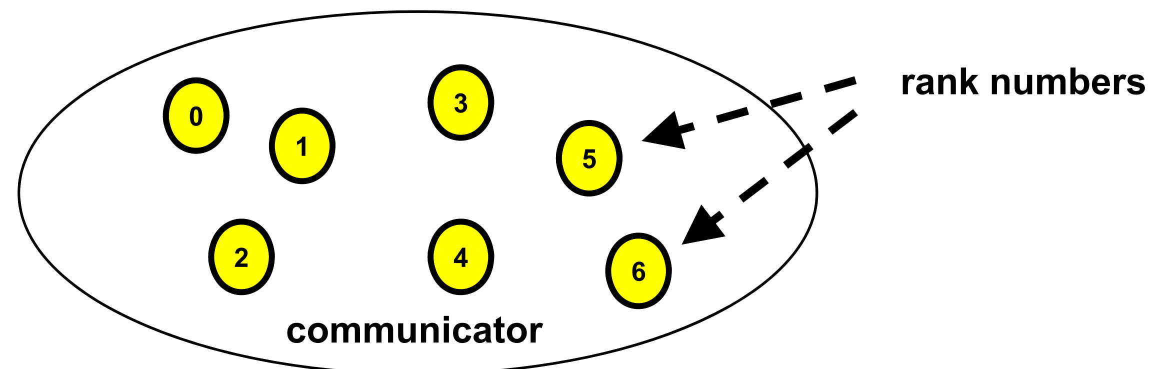
Communication Issues - MPI Answers

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

```
[…]
        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

RWTH AACHEN UNIVERSITY

```
[…]
        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

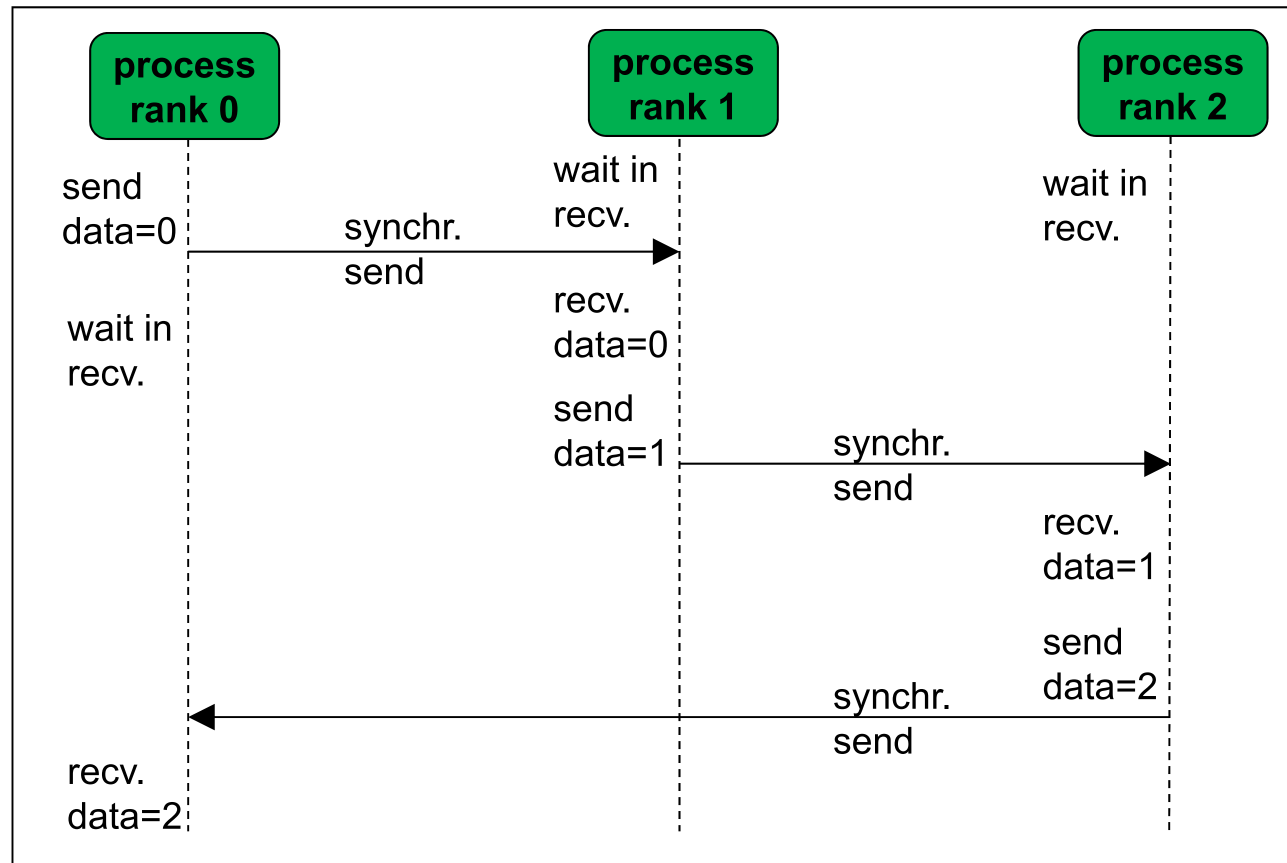

```
int MPI_Send(void *buf, int count,
    MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm)
```

- 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

RWTH AACHEN UNIVERSITY

- ```
  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

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

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

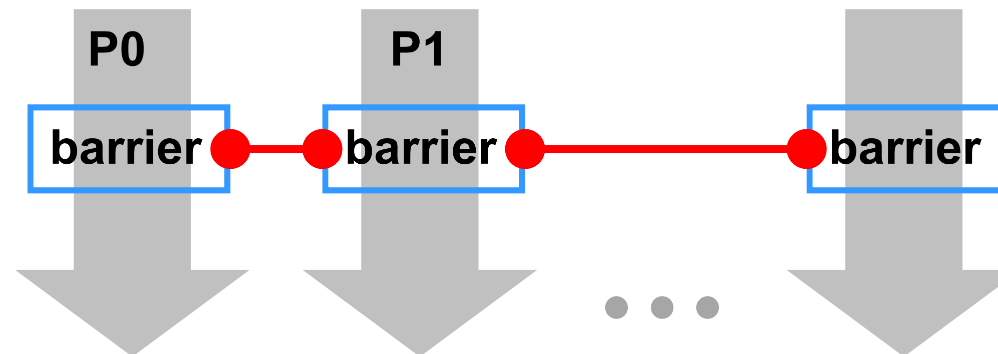
                        // receive from process i
                        MPI_Recv(message, length, MPI_INT, MPI_ANY_SOURCE,
                                MPI_ANY_TAG, MPI_COMM_WORLD, &status);

                        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

RWTH AACHEN UNIVERSITY

- `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

RWTH AACHEN UNIVERSITY

```
[…]
        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]);

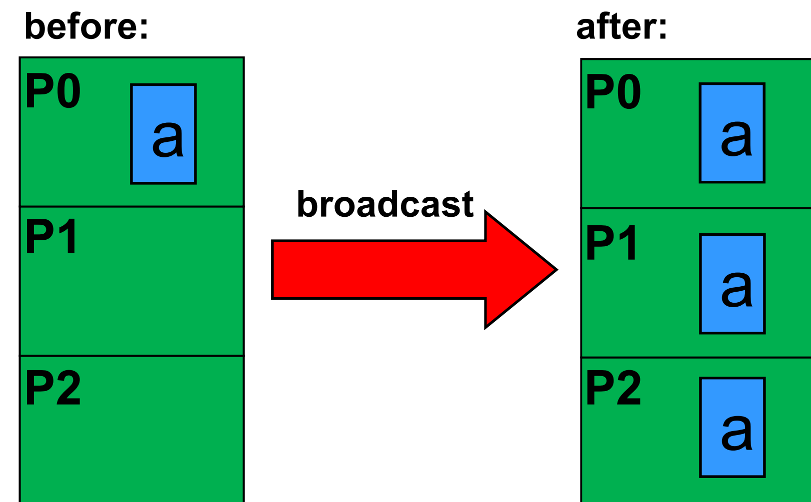
        free(message);
[…]
```

Summary

RWTH AACHEN UNIVERSITY

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

Exercises

4. **Ping Pong**

5. **Send Data to all Processes, Part I**

MPI – Part I - Contents

RWTH AACHEN UNIVERSITY

- Introduction
- Structure of MPI Programs
- Groups and Communicators
- Point to Point Communication
- MPI Data Types
- **Collective Communication**
- Summary of Part I

Collective Communication

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

```
[…]
         double tm0, tm1;

         tm0 = MPI_Wtime();

         sleep(procRank);

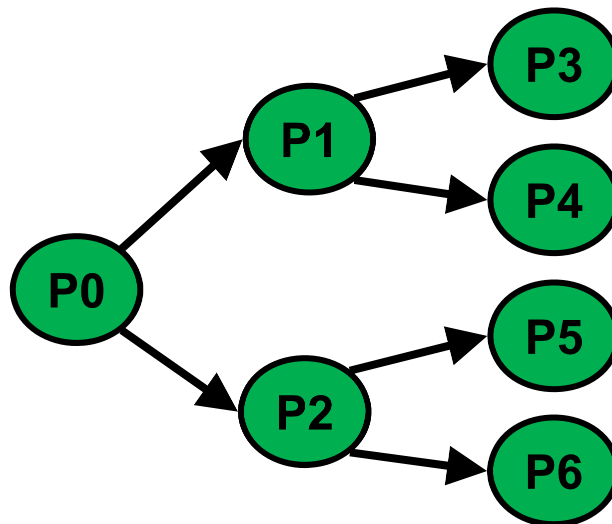
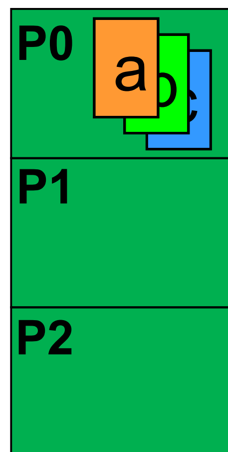
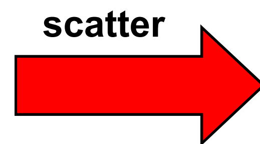
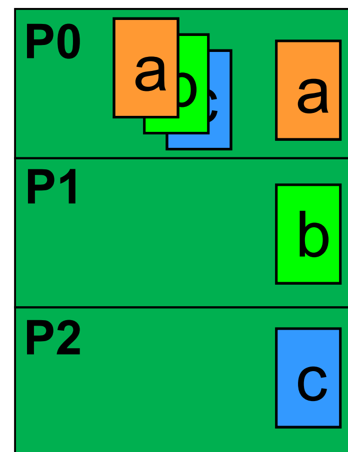
         // comment out
         MPI_Barrier(MPI_COMM_WORLD); // all wait

         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 };

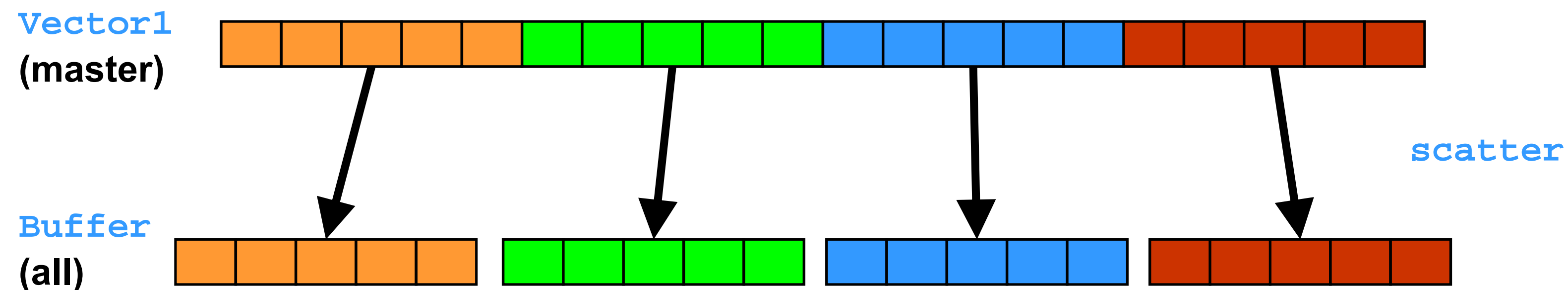
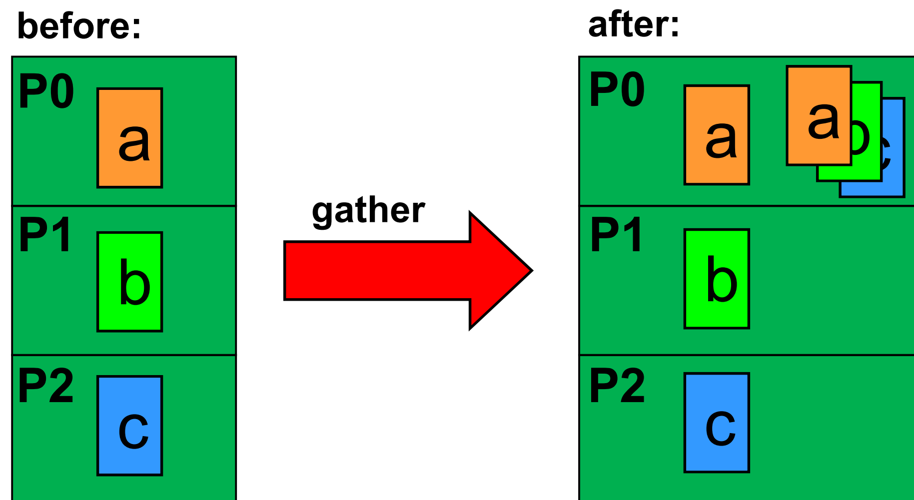
        // init for master
        if (0 == procRank)
        {
                message = 42;
        }

        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

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)

RWTH AACHEN UNIVERSITY

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

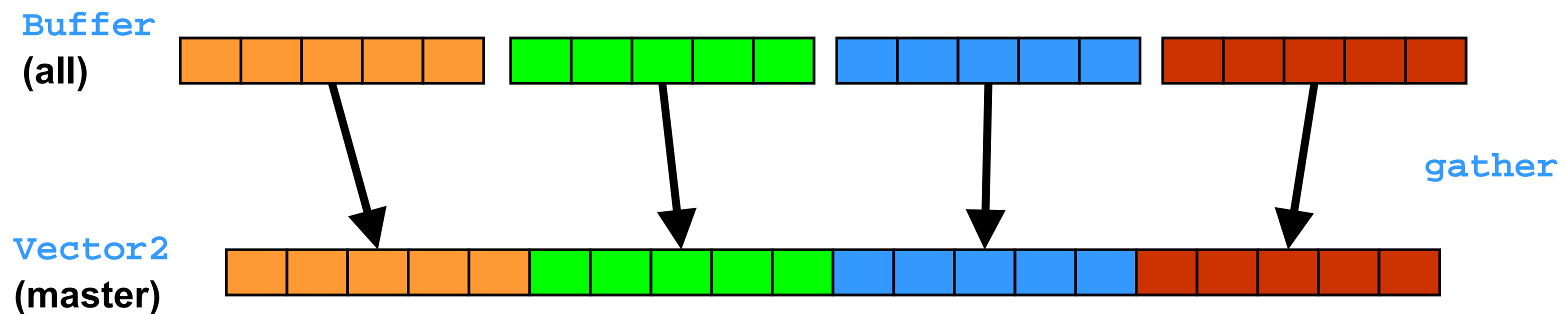
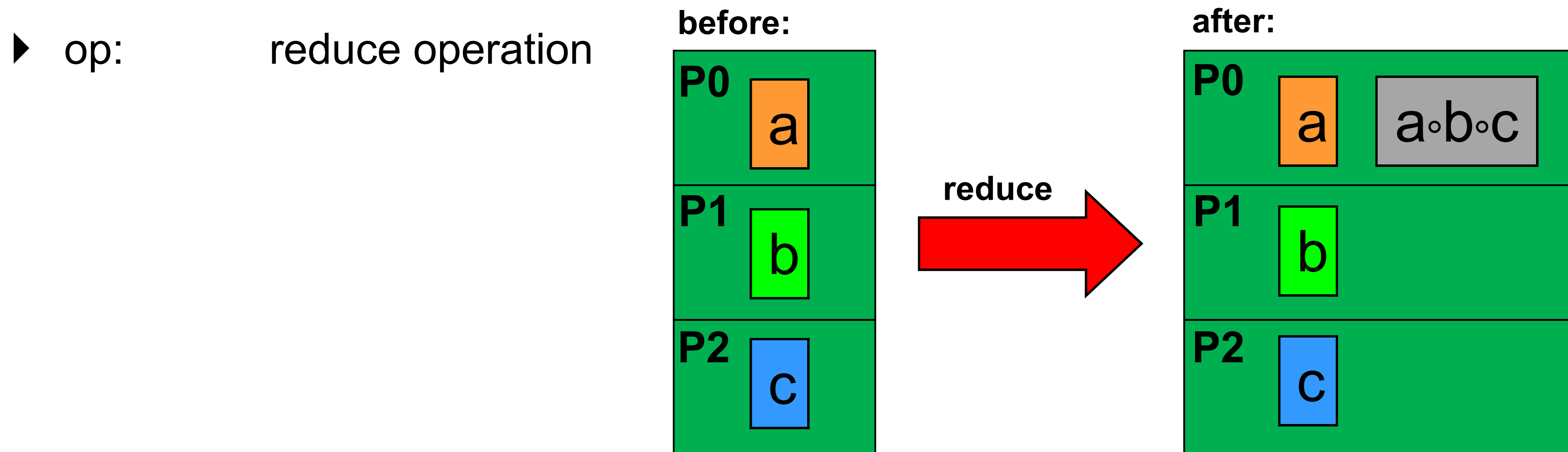
        // init for master
        if (0 == procRank)
        {
                for (int i=0; i<k; ++i)
                {
                        vector1[i]=i*35;
                }
        }

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

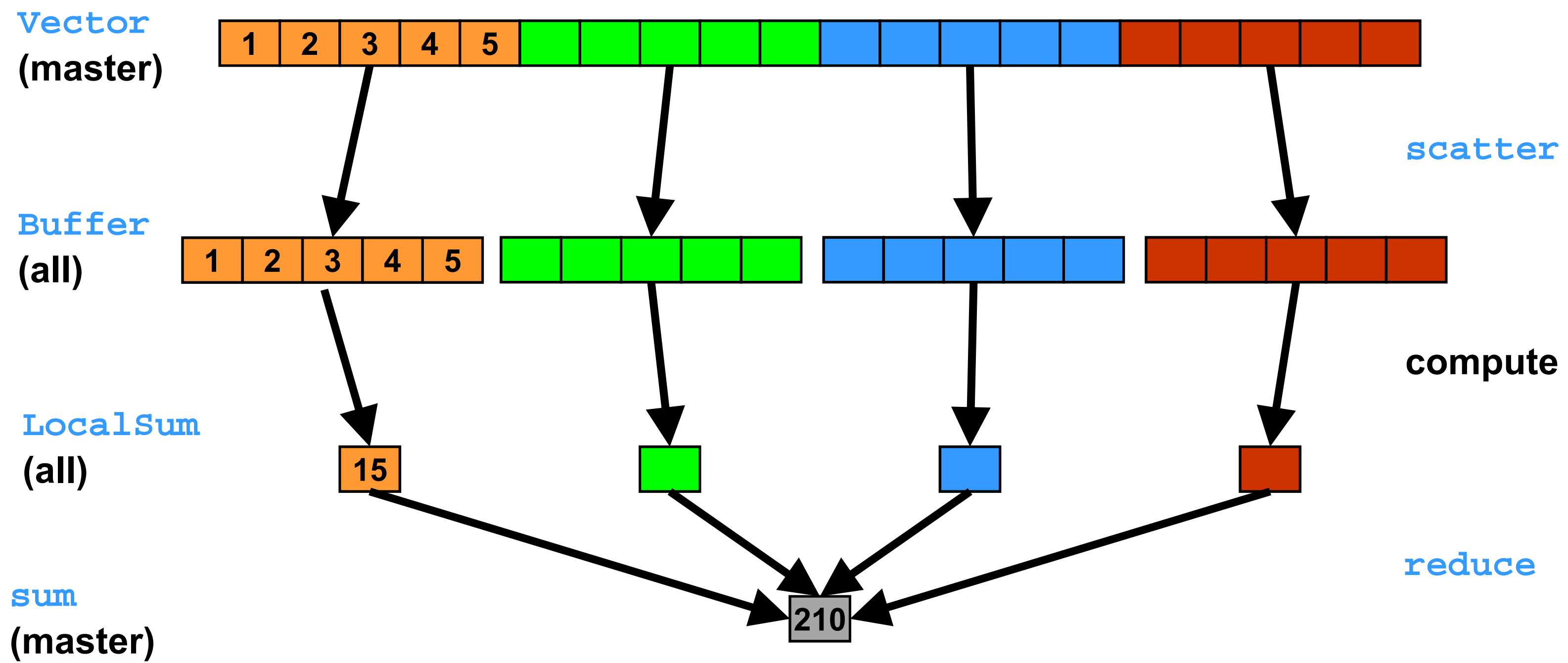
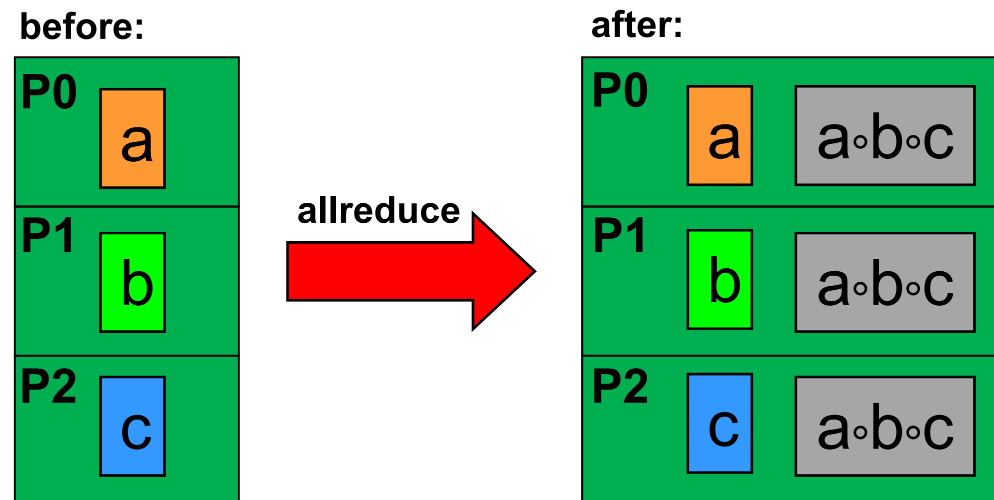
        for (int i=0; i<l; ++i)
                printf("Process %i has %i -th element
                        %i\n",procRank,i, buffer[i]);
[…]
```

Example: Scatter an Array (2 / 2)

RWTH AACHEN UNIVERSITY

- **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)

RWTH AACHEN UNIVERSITY

```
[…]
        // 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

RWTH AACHEN UNIVERSITY

```
[…]
        // 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

RWTH AACHEN UNIVERSITY

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

Exercises

RWTH AACHEN UNIVERSITY

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

MPI – Part I - Contents

RWTH AACHEN UNIVERSITY

- Introduction
- Structure of MPI Programs
- Groups and Communicators
- Point to Point Communication
- MPI Data Types
- Collective Communication
- **Summary of Part I**

Summary of Part I

RWTH AACHEN UNIVERSITY

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

Contents

- **Lecture „Introduction to Parallel Programming“**
  - Introduction
  - Java-Threads
  - MPI - Part I – Basics
  - MPI - Part II – Advanced Topics
  - Introduction to OpenMP
- **Exercises**
  - Java-Threads
  - MPI
  - OpenMP
  - Hybrid

RWTH AACHEN UNIVERSITY

MPI – Part II
Advanced Topics

More on Communication, Self-defined Datatypes, Self-defined Communicators, Load Balancing, Case Study: Game of Life

MPI – Part II - Contents

RWTH AACHEN UNIVERSITY

- Different Types of Communication
- Complex Data Structures
- Self-defined Communicators
- Load Balancing
- Case Study: Game of Life

Questions

RWTH AACHEN UNIVERSITY

- **Where is the data kept until it is received?**

- **When is a send complete?**

Blocking and Non-Blocking

RWTH AACHEN UNIVERSITY

- **Blocking:**
  - Relates to the completion of an operation in the sense, that used resources, i.e. buffers, are free to use again
- **Non-blocking:**
  - Functions return as soon as possible but provided buffers must not be touched until another appropriate call successfully indicates that they are not in use anymore
  - Even read-only access may be prohibited
  - Non-blocking communications are primarily used to overlap computation with communication to effect performance gains
- **Blocking sends can be combined with non-blocking receives and vice versa.**

Completion of Non-Blocking Operations

RWTH AACHEN UNIVERSITY

- **In non-blocking send-variants we need to check for the communication's completion**
- **There are two options in checking for a communication's completion:**
  - Wait until the communication is complete using `MPI_Wait`
  - Loop with test until communication is completed using `MPI_Test`
- **To track communication requests an integer request handle is provided by the MPI system, e.g.**
  - `int MPI_Isend(... like MPI_Send, MPI_Request *req)`
  - `int MPI_Wait(MPI_Request *req, MPI_Status *status)`

Example: Send and wait

RWTH AACHEN UNIVERSITY

```
[…]
        message=42;
        MPI_Request req;
        MPI_Issend(&message, 1, MPI_INT, 1, tagSend, MPI_COMM_WORLD,
                &req);

        int flag=0;
        while (1)
        {
                MPI_Test(&req,&flag,MPI_STATUS_IGNORE);

                if (1 == flag)
                        break;

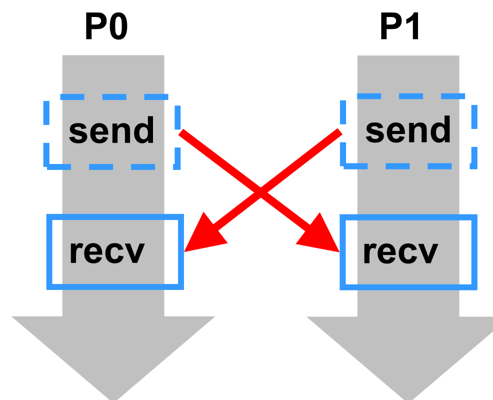
                printf("wait ...\n");
                sleep(1);
        }

        printf("Message sent \n");
[…]
```

Example: Receive and wait

RWTH AACHEN UNIVERSITY

```
[…]
        MPI_Request req;
        MPI_Irecv(&message, 1, MPI_INT, 0, tagSend, MPI_COMM_WORLD,
                &req); // no status

        int flag=0;
        while (1)
        {
                MPI_Test(&req,&flag,MPI_STATUS_IGNORE);

                if (1 == flag)
                        break;

                printf("wait ...\n");
                sleep(1);
        }

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

Synchronous and Asynchronous

RWTH AACHEN UNIVERSITY

- **Relation between Sender and Receiver**
- **Synchronous:**
  - send call will only start when the destination has started synchronous receive
  - send operation will complete only after acknowledgement that the message was safely received by the receiving process (destination has copied data out of incoming buffer into memory)
- **Asynchronous (buffered):**
  - a send operation may "complete" even though the receiving process has not actually received the message
  - only know that message has left

Variants of Communication

RWTH AACHEN UNIVERSITY

semantics mode	standard	synchronous	Asynchronous (buffered)
blocking	MPI_Send MPI_Recv	MPI_Ssend	MPI_Bsend
non-blocking	MPI_Isend MPI_Irecv	MPI_Issend	MPI_Ibsend

- **Note: There is only one receive function for both blocking and non-blocking send functions**

Deadlocks with MPI_Send

- `MPI_Send` can be synchronous, asynchronous or both (not declared in the MPI standard)

- Behavior is implementation dependent (typical: asynchronous for small messages and synchronous for large messages)

- Depending on the system, this can deadlock or not:



Example: Deadlock

```
[…]
        // force sync. send, wait for rcv. in any case
        // MPI_Send(&messageS, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD);
        MPI_Ssend(&messageS, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD);
        MPI_Recv(&messageR, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);

        printf("proc %d finished, message %d \n",procRank,messageR);
[…]
```

Example: No Deadlock

```
[…]
        // force sync. send, wait for rcv. in any case
        // MPI_Send(&messageS, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD);
        MPI_Bsend(&messageS, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD);
        MPI_Recv(&messageR, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);

        printf("proc %d finished, message %d \n",procRank,messageR);
[…]
```

Send and Receive in one Operation (1)

- `MPI_Sendrecv` **combines an asynchronous send and receive**
- **Send buffer and receive buffer must be disjoint, and may have different lengths and data types**
- **A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation**
- **A send-receive operation can receive a message sent by a regular send operation**
- **Useful for data exchange**

Send and Receive in one Operation (2)

- **Send and receive buffer must not overlap**

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

```
MPI_Sendrecv_replace(void* buf, int count,
MPI_Datatype datatype, int dest, int sendtag, int
source, int recvtag, MPI_Comm comm, MPI_Status
*status);
```

Example: Sendreceive

```
[…]
        if (0 == procRank)
        {
                messageMaster = 42;

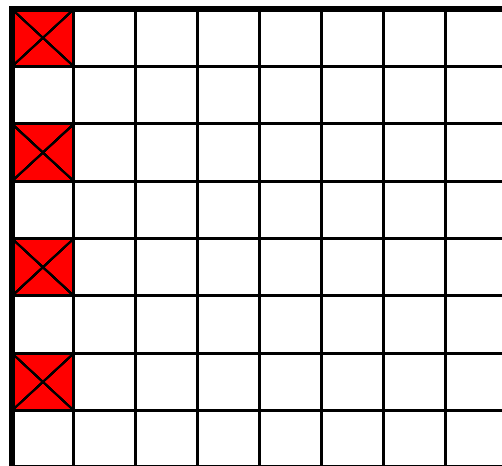
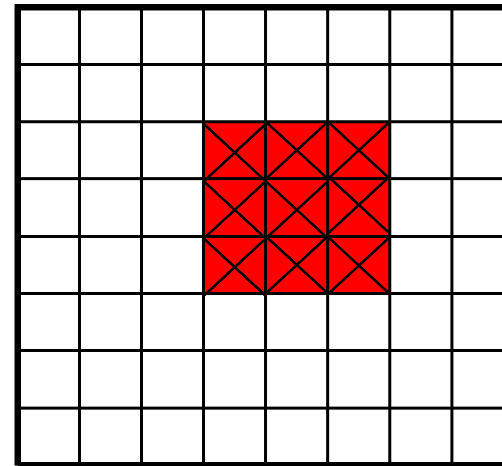
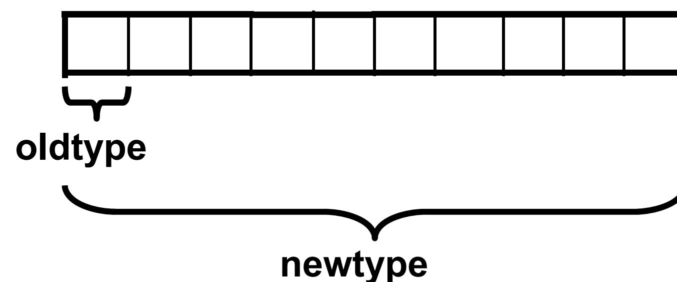
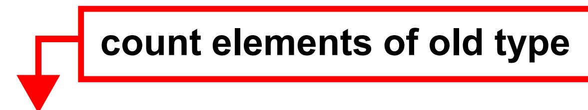
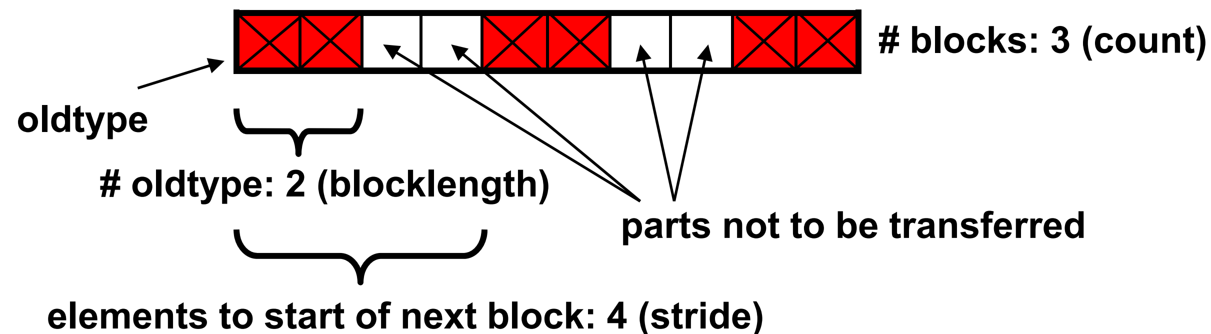
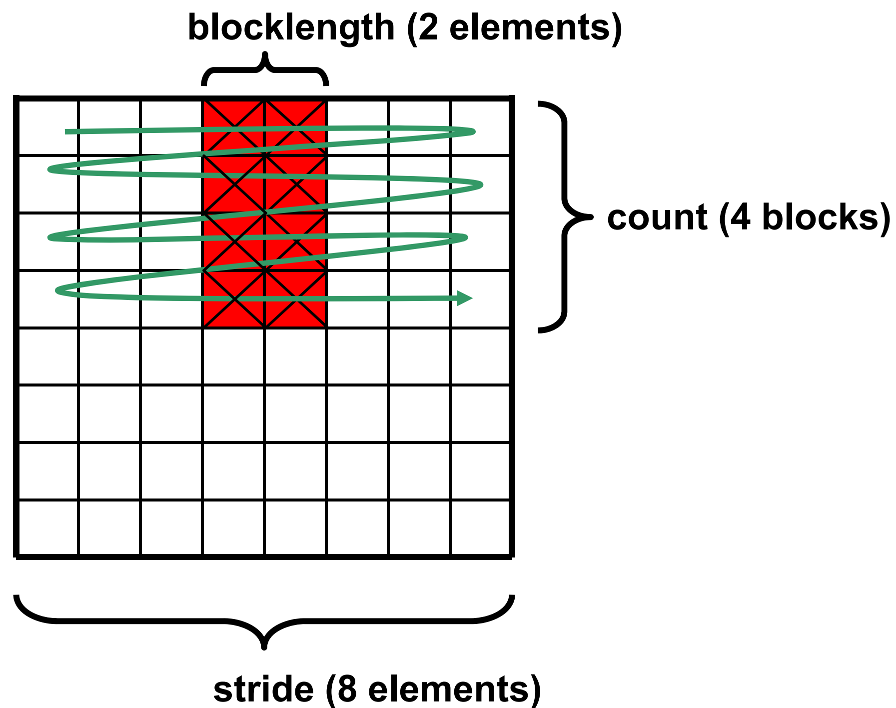
                MPI_Sendrecv(&messageMaster, 1, MPI_INT, 1, tagSendMaster,
                        &messageWorker, 1, MPI_INT, 1, tagSendWorker,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else
        {
                messageWorker = 43;

                MPI_Sendrecv(&messageWorker, 1, MPI_INT, 0, tagSendWorker,
                        &messageMaster, 1, MPI_INT, 0, tagSendMaster,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        printf("proc %d, master %d, worker %d \n",
                procRank, messageMaster, messageWorker);
[…]
```

Example: Sendreceive Replace

```
[…]
        if (0 == procRank)
        {
                message = 42;

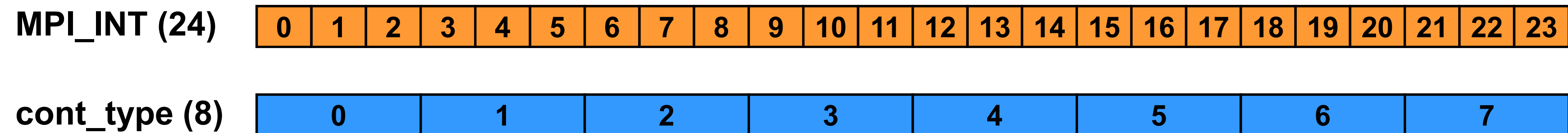
                MPI_Sendrecv_replace(&message, 1, MPI_INT, 1, tagSendMaster,
                        1, tagSendWorker, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else
        {
                message = 43;

                MPI_Sendrecv_replace(&message, 1, MPI_INT, 0, tagSendWorker,
                        0, tagSendMaster, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

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

Test for Incoming Messages

RWTH AACHEN UNIVERSITY

- **Also possible in a non-blocking mode:**
  - `int MPI_Iprobe(int src, int tag, MPI_Comm comm, int* flag, MPI_Status* status)`

  - flag: non-zero, if there is a matching message
  - status: source, tag and size of the message

  - Not necessary to receive the message immediately after it has been probed for
  - The same message may be probed for several times before it is received

15. Test-Receive the measurement

MPI – Part II - Contents

RWTH AACHEN UNIVERSITY

- Different Types of Communication
- **Complex Data Structures**
- Self-defined Communicators
- Load Balancing
- Case Study: Game of Life

Motivation

- **Using predefined data types would mean to send very small messages many times (e.g. communication of sub arrays)**





Derived Data Types

RWTH AACHEN UNIVERSITY

- **Data types in MPI: basic (already known) and derived**

- **MPI provides data type constructor functions to create derived data types**

- **Kinds of data type constructors in MPI:**
  - **contiguous**
  - **vector**/hvector
  - indexed/hindexed
  - struct

Committing and Freeing a Data Type

RWTH AACHEN UNIVERSITY

- Before a data type handle is used in communication, it needs to be committed
  - `int MPI_Type_commit(MPI_Datatype *datatype)`
- After use, a self defined data type can be deallocated
  - `int MPI_Type_free(MPI_Datatype *datatype)`

Contiguous Data Type

- **Simplest derived data type**

- **Creates a new data type consisting of contiguous elements of another data type**




```
int MPI_Type_contiguous(int count,
MPI_Datatype oldtype,MPI_Datatype *newtype)
```

Transfer of Contiguous Data

- "New" method (derived data type):




```
MPI_Type_contiguous(count, datatype, &newtype)
MPI_Type_commit(&newtype)
MPI_Send(&buffer, 1, newtype, dest, tag, comm)
```

- "Old" method (simple data type)

```
MPI_Send(&buffer, count, datatype, dest, tag, comm)
```

Vector Data Type

- More general constructor

- Allows replication of a data type into locations that consist of equally spaced blocks



❖ `int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)`

Example: Transfer of 2D Array Data

- `MPI_Type_vector` is fine for sending rectangular blocks from 2D arrays



Note: in Fortran: different memory layout

Example: Contiguous Data (1 / 2)

```
[…]
        MPI_Type_contiguous(3, MPI_INT, &cont_type); // make type: 24 = 8*3
        MPI_Type_commit(&cont_type);

        if (0 == procRank)
        {
                for (int i=0; i<24; i++)
                        buffer[i] = i;

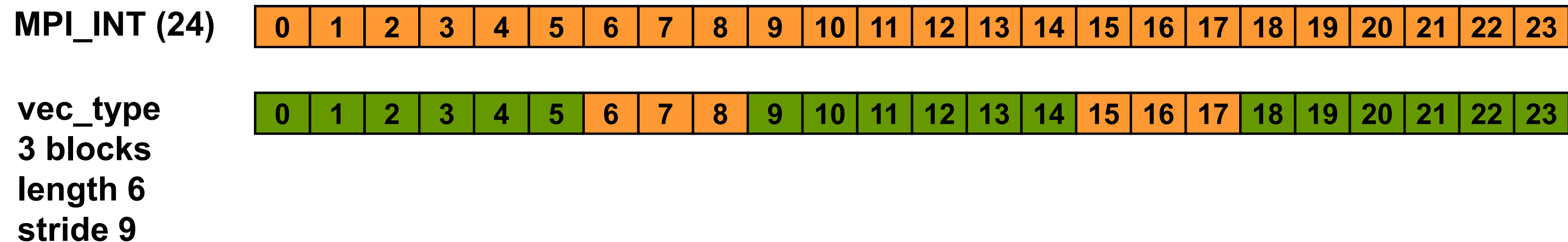
                // send data with new data type to worker
                MPI_Send(buffer, 8, cont_type, 1,  tagSendBuffer, MPI_COMM_WORLD);
        }
        else
        {
                // receive data with new data type from master
                MPI_Recv(buffer, 8, cont_type, 0, tagSendBuffer, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);

                for (int i=0; i<24; i++)
                        printf("buffer[%d] = %d\n", i, buffer[i]);
        }
[…]
```

Example: Contiguous Data (2 / 2)

RWTH AACHEN UNIVERSITY

- **The contiguous data type created in the source code on the slide before looks like this:**



Example: Vector Data of Basic Type (1 / 2)

```
[…]
          MPI_Type_vector(3, 6, 9, MPI_INT, &vec_type); // 3 blocks, length 6, stride 9
          MPI_Type_commit(&vec_type);

          if (0 == procRank)
          {
                    for (int i=0; i<24; i++)
                              buffer[i] = 2*i;

                    // send data with new data type to worker
                    MPI_Send(buffer, 1, vec_type, 1,  tagSendBuffer, MPI_COMM_WORLD);
          }
          else
          {
                    for (int i=0; i<24; i++)
                              buffer[i] = -1;

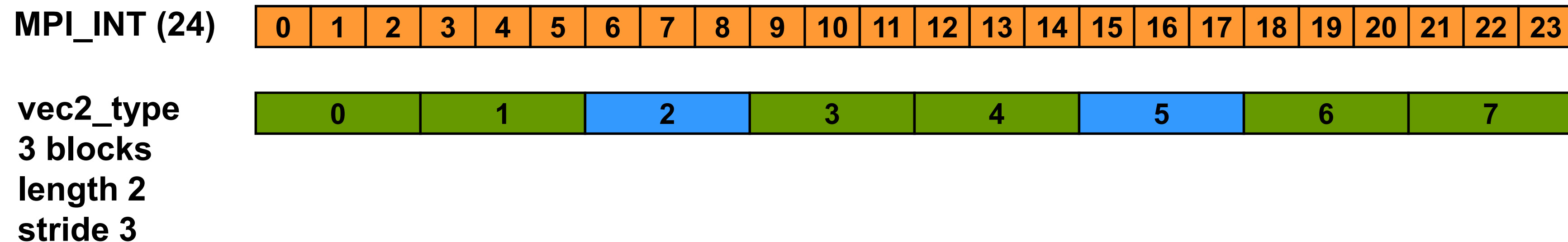
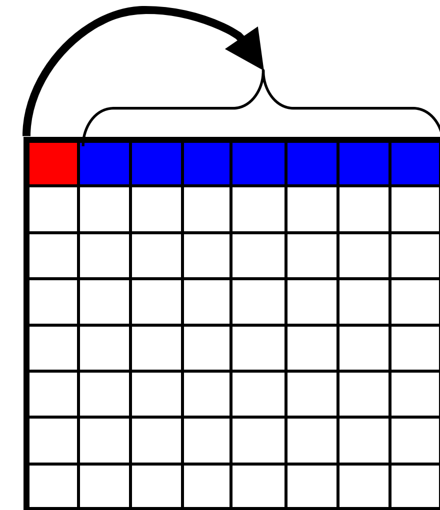
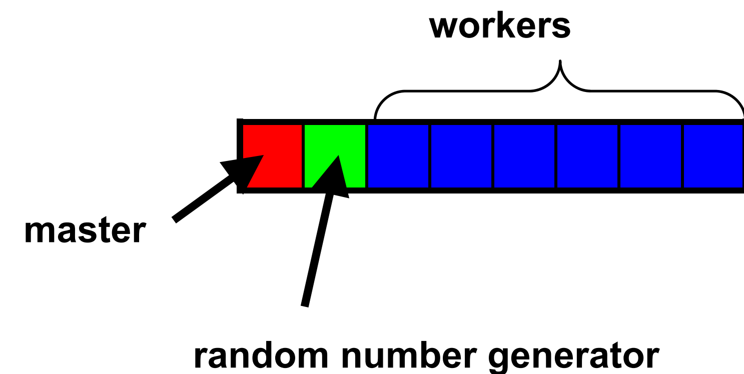
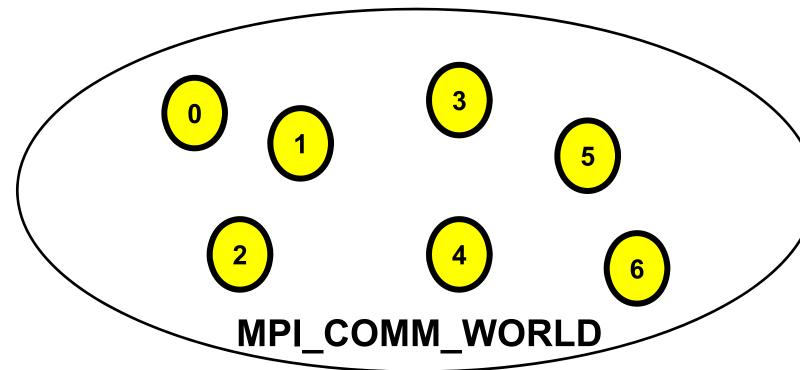
                    // receive data with new data type from master
                    MPI_Recv(buffer, 1, vec_type, 0, tagSendBuffer, MPI_COMM_WORLD,
                              MPI_STATUS_IGNORE);

                    for (int i=0; i<24; i++)
                              printf("buffer[%d] = %d\n", i, buffer[i]);
          }
[…]
```

Example: Vector Data of Basic Type (2 / 2)

- **The vector data type based on a basic data type created in the source code on the slide before looks like this:**



Example: Vector Data of own Data Type (1 / 2)

RWTH AACHEN UNIVERSITY

```
[…]
          MPI_Type_vector(3, 2, 3, cont_type, &vec2_type); //3 blocks, length 2, stride 3
          MPI_Type_commit(&vec2_type);

          if (0 == procRank)
          {
                    for (int i=0; i<24; i++)
                              buffer[i] = 3*i;

                    // send data with new data type to worker
                    MPI_Send(buffer, 1, vec2_type, 1,  tagSendBuffer, MPI_COMM_WORLD);
          }
          else
          {
                    for (int i=0; i<24; i++)
                              buffer[i] = -1;

                    // receive data with new data type from master
                    MPI_Recv(buffer, 1, vec2_type, 0, tagSendBuffer, MPI_COMM_WORLD,
                              MPI_STATUS_IGNORE);

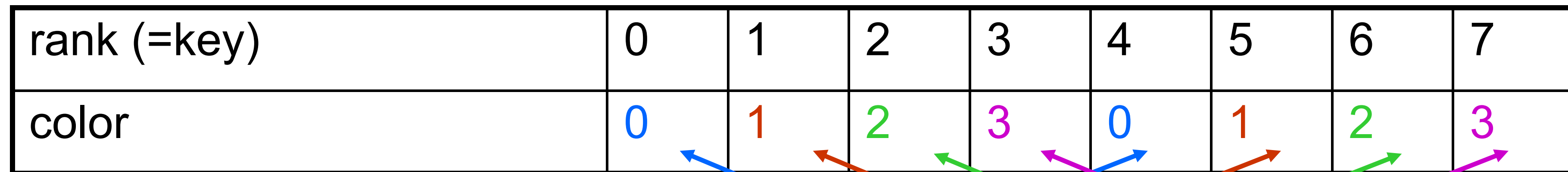
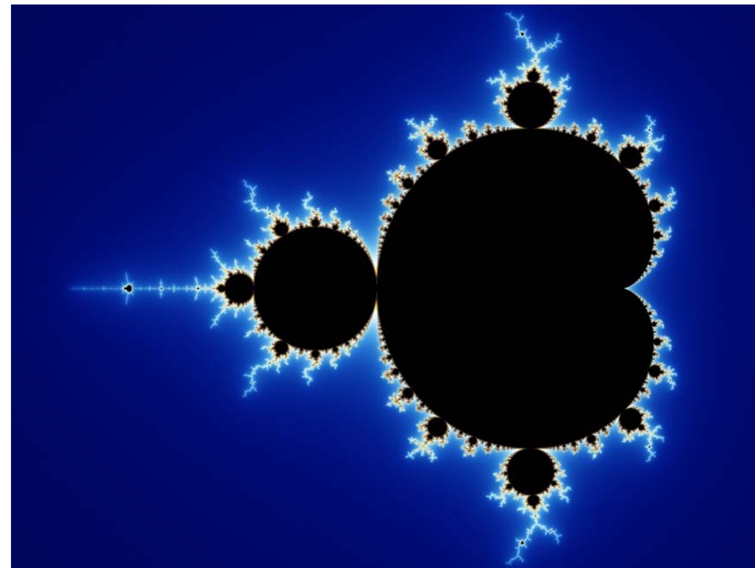
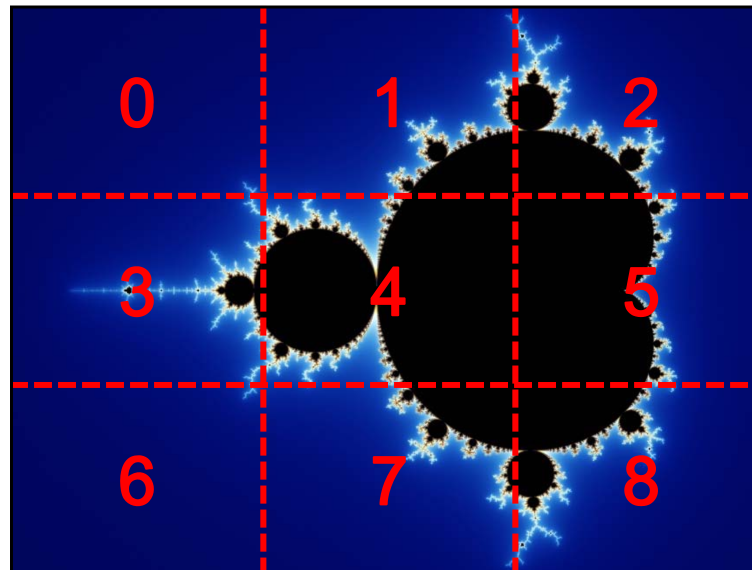
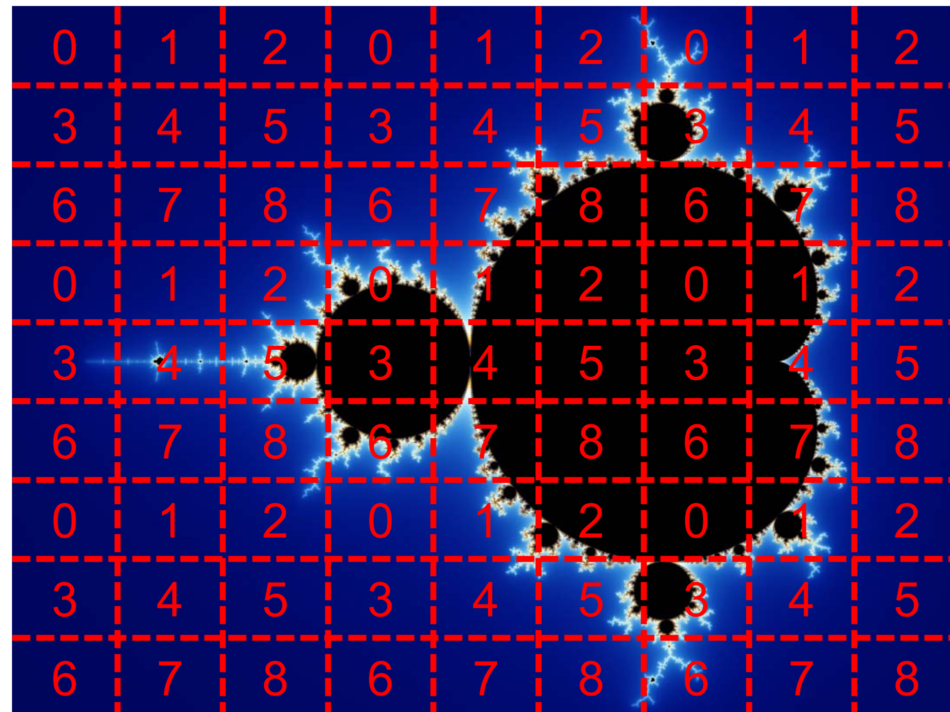
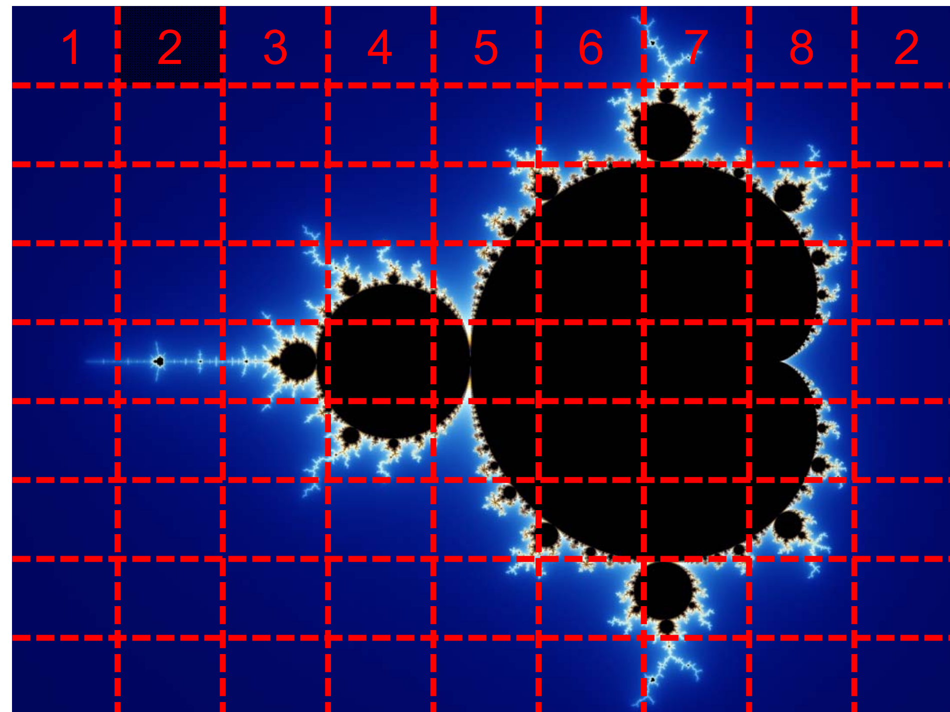
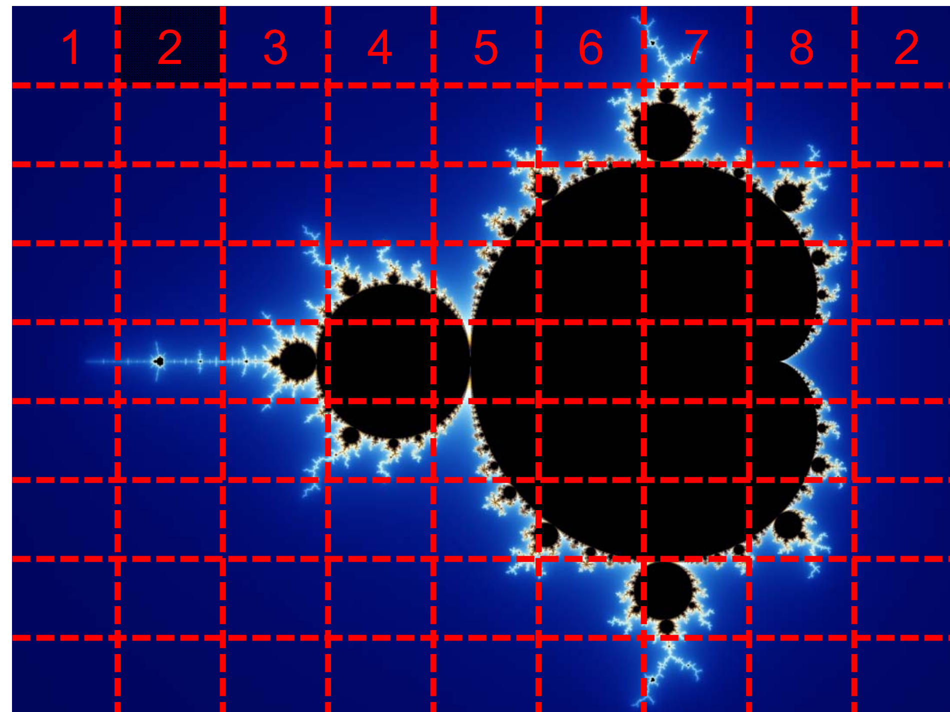
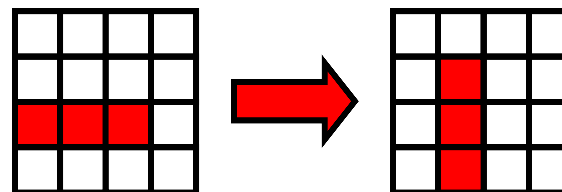
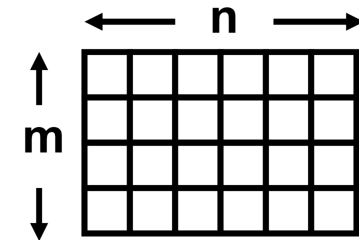
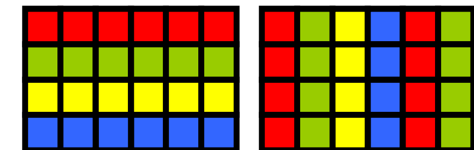
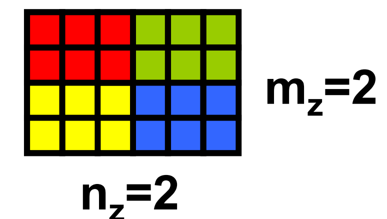
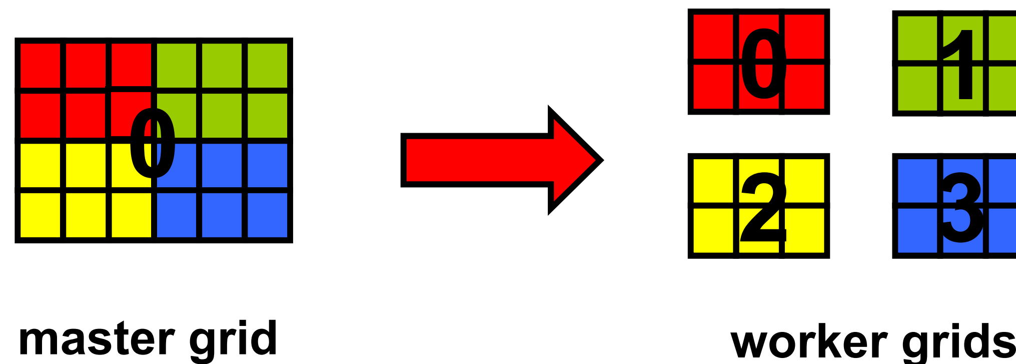
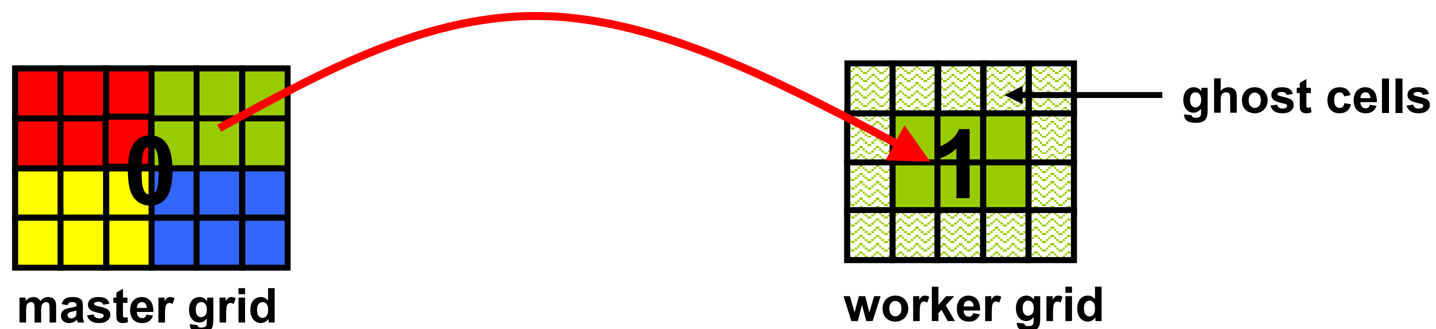
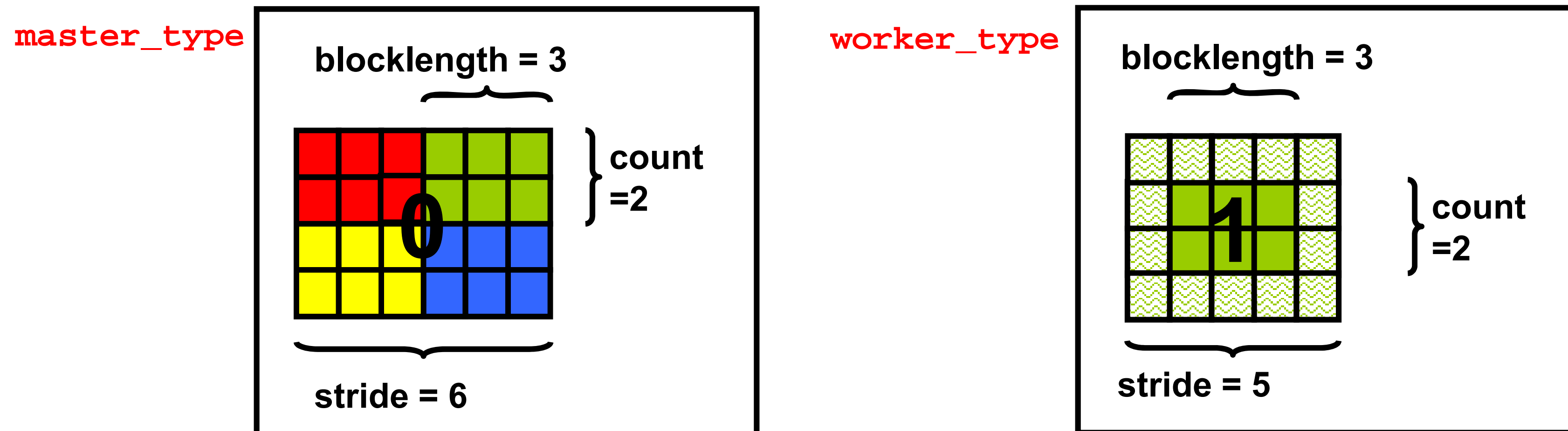
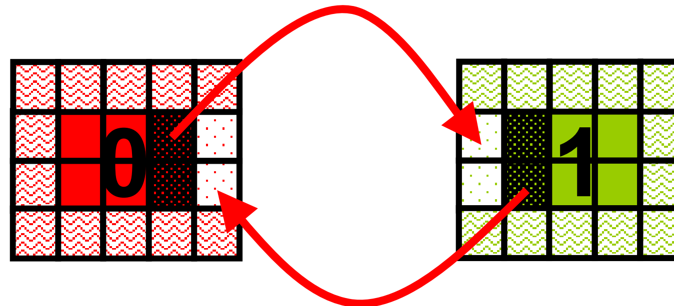
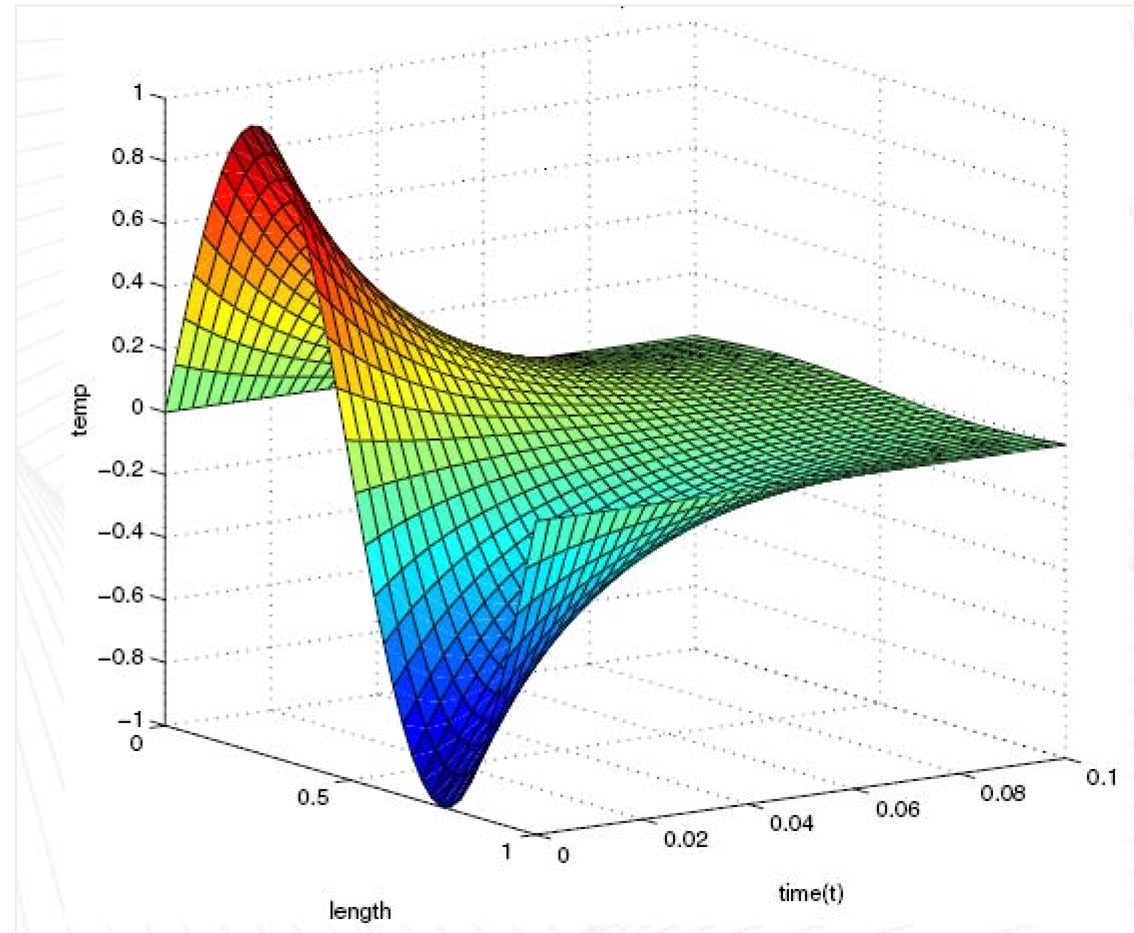
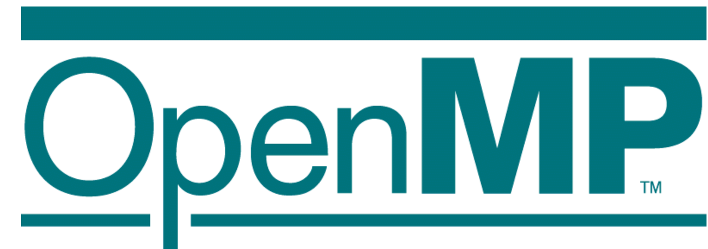
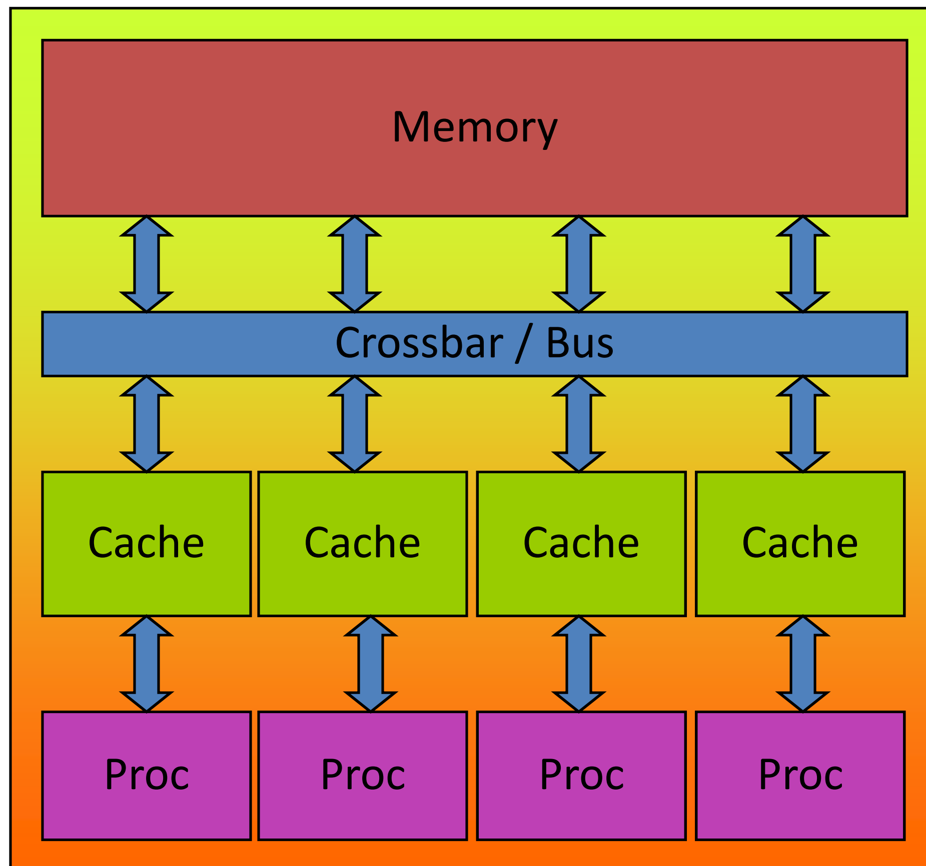
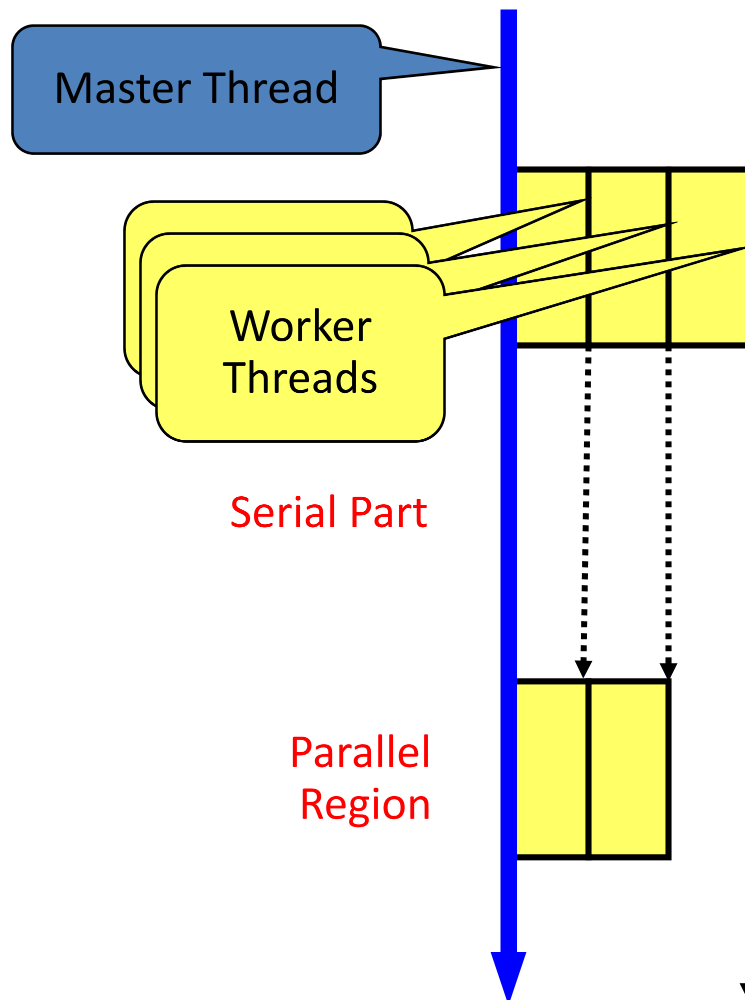
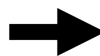
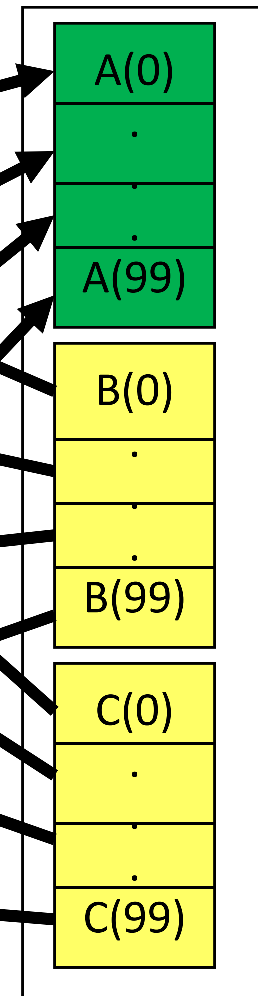
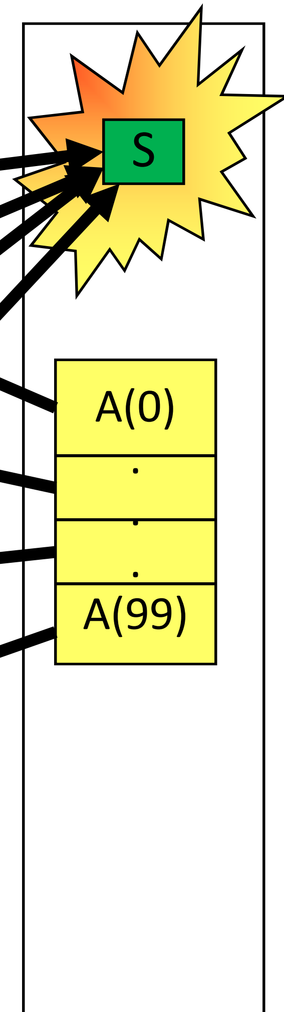
                    for (int i=0; i<24; i++)
                              printf("buffer[%d] = %d\n", i, buffer[i]);
          }
[…]
```

Example: Vector Data of own Data Type (2 / 2)

- **The vector data type based on a self-defined data type created in the source code on the slide before looks like this:**



16. Vector data type

MPI – Part II - Contents

- Different Types of Communication
- Complex Data Structures
- **Self-defined Communicators**
- Load Balancing
- Case Study: Game of Life

Why new Communicators?

RWTH AACHEN UNIVERSITY

- Communication in a single row of processes instead of the whole matrix



- Different tasks for processes



- **Base communicator for all MPI communicators is predefined:** `MPI_COMM_WORLD`



- **One can**
  - either construct a new communicator (means to form a new group)

  or

  - split a communicator into a group of new communicators

Construct new Communicator

RWTH AACHEN UNIVERSITY

- Extract the "process group" from the communicator with
  - `int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)`
- Use MPI_Group-Commands to alter groups
  - `int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)`
  - `int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)`
- Create a communicator around a new group with
  - `int MPI_Comm_create(MPI_Comm oldcomm, MPI_Group newgroup, MPI_Comm *newcomm)`

    collective command - processes of the old communicator not included get a dummy value

Example: Construct new Communicator

RWTH AACHEN UNIVERSITY

```
[…]
        // remove two processes
        int loser[2]; //have to leave the world_group
        MPI_Group world_group, win_group;
        MPI_Comm win_comm;

        // first and last have to go
        loser[0]=0;
        loser[1]=procCount-1;

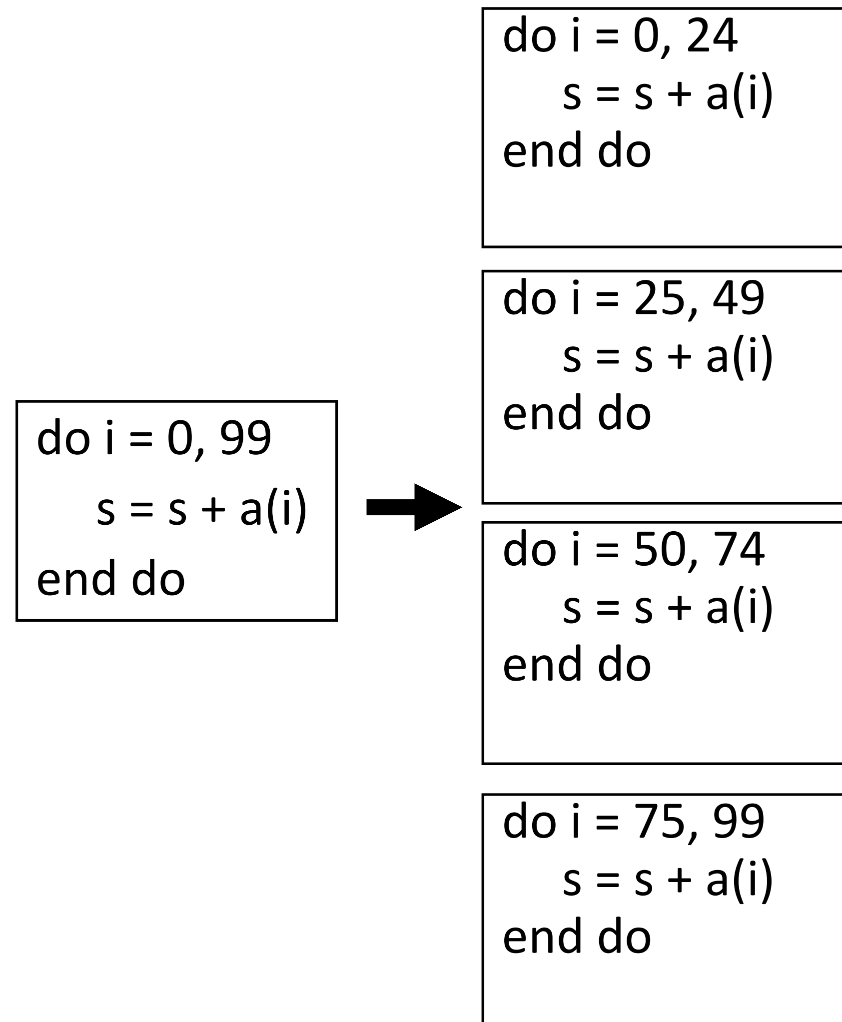
        // return group of communicator
        MPI_Comm_group(MPI_COMM_WORLD, &world_group);

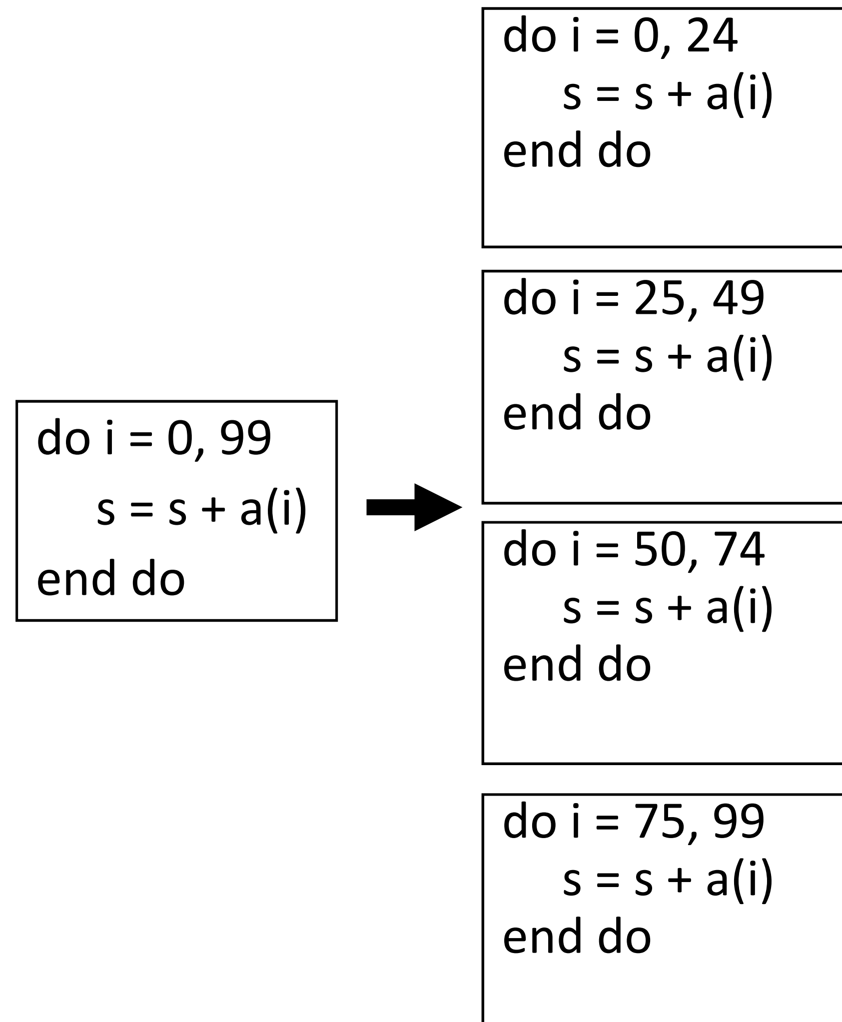
        // create new group without loser
        MPI_Group_excl(world_group, 2, loser, &win_group);

        // create communicator (subset of group of comm)
        MPI_Comm_create(MPI_COMM_WORLD, win_group, &win_comm);
[…]
```

Split a Communicator

RWTH AACHEN UNIVERSITY

- `int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)`

- A new communicator *newcomm* is created for each value of color
- Color value `MPI_UNDEFINED` allowed, in which case newcomm returns `MPI_COMM_NULL`
- Within the new communicators, the processes are ranked in the order(!) defined by the value of the argument key
- Collective call, but each process can provide different values for color and key

Example: Split in 4 Groups (1 / 2)

RWTH AACHEN UNIVERSITY

```
[…]
        int color = procRank%4;

        MPI_Comm newcomm;
        MPI_Comm_split(MPI_COMM_WORLD, color, procRank,
                       &newcomm);
[…]
```

Example: Split in 4 Groups (2 / 2)



rank (=key)	0	1	2	3	4	5	6	7
color	0	1	2	3	0	1	2	3

4 new communicators

new comm	0	1	2	3	0	1	2	3
new rank	0	0	0	0	1	1	1	1

Communicator Destructor

RWTH AACHEN UNIVERSITY

- `int MPI_Comm_free(MPI_Comm *comm)`

- Collective operation
- Marks the communication object for deallocation
- Any pending operations that use this communicator will complete normally
- The object is actually deallocated only if there are no other active references to it

17. Define Communicators

MPI – Part II - Contents

RWTH AACHEN UNIVERSITY

- Different Types of Communication
- Complex Data Structures
- Self-defined Communicators
- **Load Balancing**
- Case Study: Game of Life

Example: Mandelbrot Set

- **The Mandelbrot set is a set of complex values**

- **A complex number c is in the Mandelbrot set if, when starting with $x_0 = 0$ and applying the iteration $x_{n+1} = x_n^2 + c$ repeatedly, the absolute value of x_n never exceeds a certain number**



Parallel Calculation of the Set

RWTH AACHEN UNIVERSITY

- Each process calculates one part of the area
- Process 4 has much more work to do than process 0 and 6



⇨ Load balancing problem

Solution #1

- Redesign the distribution of your processes
- The over-all workload is nearly equal for all processes
- The distribution can also be determined by random numbers



Master-Worker-Pattern: First Result

- **Assume process 2 is ready first**
  - gives the results to the master and asks for more work
  - process 2 gets the next free block
  - and so on …



Master-Worker-Pattern: Last Results

RWTH AACHEN UNIVERSITY

- **Now, let any process be ready:**
  - no more blocks to calculate
  - master sends a message saying: finish your work (e.g. with a special tag)
  - when all workers have finished their work, the master finishes, too
  - `MPI_Finalize` is a collective operation
  - ⇨ the program ends, when all processes have called it



MPI – Part II - Contents

RWTH AACHEN UNIVERSITY

- Different Types of Communication
- Complex Data Structures
- Self-defined Communicators
- Load Balancing
- **Case Study: Game of Life**

Game of Life

- **"cellular automaton"**
  - iterations over a 2d-array
  - cells can be live (1) or dead (0)
  - each cell interacts with its 8 neighbors

0	0	0
1	1	0
0	1	0

- **Rules for each iteration (all cells at the same time!):**
  - 0-1 neighbors live: new state=0 (living cells die)
  - 2 neighbors live: new state=old state
  - 3 neighbors live: new state=1 (dead cells come to live)
  - 4-8 neighbors live: new state=0 (living cells die)
  - Example:



Domain Decomposition (1 / 2)

- **game of life takes place on a m x n grid**



- **distribute the grid on z processors (domain decomposition)**
  - simplest way: row wise or column wise
  - more general approach: rectangular areas (checkerboard partitioning)



  - constraints:
    - $m \,\%\, m_z = 0$
    - $n \,\%\, n_z = 0$
    - $m_z \cdot nz = z$



  - get the best (most compact) distribution with `MPI_Dims_Create`

Domain Decomposition (2 / 2)

- **Initial (master) grid is in process 0**
- **Parts must get distributed to the other processes**



Ghost Cells

- **For updating the cells, we need all the neighbours of all the cells**
  - “ghost cells” around each block are necessary

- **This means**
  - cells are not continuous in memory; neither in the master nor in the worker grid
  - they are arranged in different ways



New Data Types

- Define new MPI datatypes with
  - `MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)`



- The “type signature” of both types matches
- It is possible to send `master_type` and receive `worker_type`

Data Exchange

RWTH AACHEN UNIVERSITY

- **Exchange between subdomains to fill the ghost cells**



- **Each process exchanges borders with all its 4 neighbours**

- **Use again new MPI data types:**
  - rows: `MPI_Type_contiguous`
  - columns: `MPI_Type_vector`

- **Data exchange with `MPI_Sendrecv`**

Iterations and Visualization

RWTH AACHEN UNIVERSITY

- **The calculation is a sequence of iterations and data exchanges between subdomains**
- **When the data is to be visualized, all the worker data must be copied back to the master grid**
- **More about GOL visualization …**
  http://golly.sourceforge.net

Summary of Part II

RWTH AACHEN UNIVERSITY

- **Blocking/non-blocking , synchronous/asynchronous**
- **Derived data types**
- **New communicators**
- **Load balancing**

Example / Exercise: Heat-Diffusion

RWTH AACHEN UNIVERSITY

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2}$$



$$\frac{u_i^{(j+1)} - u_i^{(j)}}{\Delta t} = \frac{u_{i+1}^{(j)} - 2u_i^{(j)} + u_{i-1}^{(j)}}{(\Delta x)^2}$$

Contents

- **Lecture „Introduction to Parallel Programming“**
  - Introduction
  - Java-Threads
  - MPI - Part I – Basics
  - MPI - Part II – Advanced Topics
  - Introduction to OpenMP
- **Exercises**
  - Java-Threads
  - MPI
  - OpenMP
  - Hybrid

RWTH AACHEN UNIVERSITY

Introduction to OpenMP

Worksharing, Scoping, Synchronization, Advanced Worksharing, Tasking, Miscellaneous

RWTH AACHEN UNIVERSITY

- **Introduction to OpenMP**
  - Worksharing
  - Scoping
  - Synchronization
  - Advanced Worksharing
  - Miscellaneous
- **Tasks**
  - Case Study: Traversing a Tree

History

- **De-facto standard for Shared-Memory Parallelization.**
  - 1997: OpenMP 1.0 for FORTRAN
  - 1998: OpenMP 1.0 for C and C++
  - 1999: OpenMP 1.1 for FORTRAN (errata)
  - 2000: OpenMP 2.0 for FORTRAN
  - 2002: OpenMP 2.0 for C and C++
  - 2005: OpenMP 2.5 now includes both programming languages.
  - 08/2007: OpenMP 3.0 draft
  - 05/2008: OpenMP 3.0 release



http://www.OpenMP.org

OpenMP Overview (1 / 2)

RWTH AACHEN UNIVERSITY

- **OpenMP: Shared-Memory Parallel Programming Model.**



All processors/cores access a shared main memory.

Parallelization in OpenMP employs threads.

OpenMP Overview (2 / 2)



- OpenMP programs start with just one thread: The *Master*.
- *Worker* threads are spawned at *Parallel Regions*. Together with the Master they form a *Team*.
- In between Parallel Regions the Worker threads are put to sleep.
- Concept: *Fork-Join*.
- Allows for an incremental parallelization!

Directives and Structured Blocks

RWTH AACHEN UNIVERSITY

- **The parallelism has to be expressed explicitly.**

C/C++

```
#pragma omp parallel
{
    ...
    structured block
    ...
}
```

- ***Structured Block***
  - Exactly one entry point at the top
  - Exactly one exit point at the bottom
  - Branching in or out is not allowed
  - Terminating the program is allowed (abort)

Starting OpenMP programs

RWTH AACHEN UNIVERSITY

- **Specification of number of threads:**
  - Environment variable: `OMP_NUM_THREADS`
  - **Or**: Via `num_threads` clause:

    ```
    #pragma omp parallel num_threads(num) {…}
    ```

  - **Or**: Via explicit runtime call:

    ```
    #include <omp.h>
    omp_set_num_threads(num);
    ```

RWTH AACHEN UNIVERSITY

- **Introduction to OpenMP**
  - Worksharing
  - Scoping
  - Synchronization
  - Advanced Worksharing
  - Miscellaneous
- **Tasks**
  - Case Study: Traversing a Tree

Worksharing (1 / 3)

RWTH AACHEN UNIVERSITY

- **If only the *parallel* construct is used, each thread executes the Structured Block.**
  - Sharing work between the threads to achieve speedup
- **OpenMP's most common Worksharing construct: *for***

C/C++

```
int i;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
   a[i] = b[i] + c[i];
}
```

  - Distribution of loop iterations over all threads in a Team.
  - Scheduling of the distribution can be influenced.
- **Typically loops account for most of the program runtime!**

Worksharing (2 / 3)

Pseudo-Code
Here: 4 Threads

```
do i = 0, 99
    a(i) = b(i) + c(i)
end do
```



```
do i = 0, 24
    a(i) = b(i) + c(i)
end do
```

```
do i = 25, 49
    a(i) = b(i) + c(i)
end do
```

```
do i = 50, 74
    a(i) = b(i) + c(i)
end do
```

```
do i = 75, 99
    a(i) = b(i) + c(i)
end do
```

Memory



Worksharing (3 / 3)

- ***for*-construct: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the *schedule* clause:**
  - `schedule(static [, chunk])`: Iteration space divided into blocks of *chunk* size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.
  - `schedule(dynamic [, chunk])`: Iteration space divided into blocks of *chunk* (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.
  - `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to *chunk*.

- **Default on most implementations is `schedule(static)`**

RWTH AACHEN UNIVERSITY

- **Introduction to OpenMP**
  - Worksharing
  - Scoping
  - Synchronization
  - Advanced Worksharing
  - Miscellaneous
- **Tasks**
  - Case Study: Traversing a Tree

Scoping (1 / 2)

RWTH AACHEN UNIVERSITY

- **Challenge of Shared-Memory parallelization: Managing the Data Environment.**
- ***Scoping* in OpenMP: Dividing variables in *shared* and *private*:**
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - Default is *shared*
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for each thread
    - *firstprivate*: Initialization with Master‘s value
    - *lastprivate*: Value of last loop iteration is written back to Master
  - Static variables are *shared*

Scoping (2 / 2)

- **Global / static variables can be privatized with the *threadprivate* directive**
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region

C/C++

```
static int i;
#pragma omp threadprivate(i)
```

Introduction to OpenMP - Contents

- **Introduction to OpenMP**
  - Worksharing
  - Scoping
  - Synchronization
  - Advanced Worksharing
  - Miscellaneous
- **Tasks**
  - Case Study: Traversing a Tree

Synchronization (1 / 5)

- **Can all loops be parallelized with `for`-constructs? No!**
  - Simple test: If the results differ when the code is executed backwards, the loop iterations are not independent. BUT: This test alone is not sufficient:

C/C++

```
int i;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
   s = s + a[i];
}
```

- ***Data Race*: If between two synchronization points at least one thread writes to a memory location from which at least one other thread reads, the result is not deterministic (race condition).**

Synchronization (2 / 5)

Pseudo-Code
Here: 4 Threads

```
do i = 0, 99
    s = s + a(i)
end do
```

```
do i = 0, 24
    s = s + a(i)
end do
```

```
do i = 25, 49
    s = s + a(i)
end do
```

```
do i = 50, 74
    s = s + a(i)
end do
```

```
do i = 75, 99
    s = s + a(i)
end do
```



Synchronization (3 / 5)

- **A *Critical Region* is executed by all threads, but by only one thread simultaneously (*Mutual Exclusion*).**

C/C++

```
#pragma omp critical (name)
{
    ... structured block ...
}
```

- **Do you think this solution scales well?**

C/C++

```
int i;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
#pragma omp critical
    {  s = s + a[i];  }
}
```

Synchronization (4 / 5)

Very bad scalability:

```
#pragma omp parallel
{

#pragma omp for
    for (i = 0; i < 99; i++)
    {
        #pragma omp critical
        {
            sp  = sp   + a[i];
        }
    }

} // end parallel
```

Good scalability:



Synchronization (4 / 5)

Very bad scalability:

```
#pragma omp parallel
{
   double sp;
#pragma omp for
    for (i = 0; i < 99; i++)
    {
       #pragma omp critical
       {
          sp  = sp + a[i];
       }
    }
#pragma omp critical
{  s += sp; }

} // end parallel
```

Good scalability:



Synchronization (5 / 5)


```
#pragma omp parallel
{


#pragma omp for
   for (i = 0; i < 99; i++)
   {

         s     = s     + a[i];

   }



} // end parallel
```

Synchronization (5 / 5)

```
#pragma omp parallel
{


#pragma omp for reduction(+:s)
   for (i = 0; i < 99; i++)
   {

         s    = s    + a[i];

   }




} // end parallel
```

Introduction to OpenMP - Contents

RWTH AACHEN UNIVERSITY

- **Introduction to OpenMP**
  - Worksharing
  - Scoping
  - Synchronization
  - Advanced Worksharing
  - Miscellaneous
- **Tasks**
  - Case Study: Traversing a Tree

Advanced Worksharing (1 / 5)

RWTH AACHEN UNIVERSITY

- **Code in the `single`-construct is executed by one thread only.**
  - The thread selection is implementation-defined.
  - The thread might change over time and from run to run.

C/C++

```
#pragma omp parallel
{
    ...
#pragma omp single
{
    ...
}

} /* end parallel */
```

- **The `master`-construct can be used to execute a code block only by the Master thread.**

Advanced Worksharing (2 / 5)

RWTH AACHEN UNIVERSITY

- ***Orphaning* allows to syntactically separate the Worksharing constructs from the Parallel Region.**
  - Parallelization overhead is reduced if multiple consecuting worksharing constructs are combined in one Parallel Region.

- **The `barrier` construct implements a synchronization point: All threads wait until each single thread of a team has reached the barrier.**

C/C++
```
#pragma omp barrier
```

RWTH AACHEN UNIVERSITY

- **The *sections*-construct is used to execute code parts by different threads:**

C/C++

```
#pragma omp parallel sections
{

#pragma omp section
 {
   ... structured block ...
 }

#pragma omp section
 {
   ... structured block ...
 }

}
```

- **The first `section` directive may be omitted.**

Advanced Worksharing (4 / 5)

- **There is an implicit Barrier at Worksharing constructs:**

C/C++

```
int i;
#pragma omp for
for (i = 0; i < 100; i++)
{
   a[i] = b[i] + c[i];
}
/* implicit #pragma omp barrier */
```

- **The Barrier can be omitted by specifying the `nowait` clause:**

C/C++

```
int i;
#pragma omp for nowait
for (i = 0; i < 100; i++)
{
   a[i] = b[i] + c[i];
}
/* no barrier here */
```

Advanced Worksharing (5 / 5)

RWTH AACHEN UNIVERSITY

- **All Worksharing constructs contain an implicit barrier at the end, that can be omitted by specifying the `nowait` clause.**
  - The `master` construct is not a worksharing construct, `single` is.

C/C++

```
#pragma omp parallel {

#pragma omp for nowait
for (i = 0; i < 100; i++) {
   a[i] = b[i] + c[i];
}

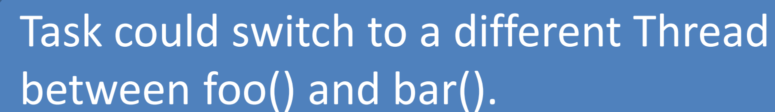
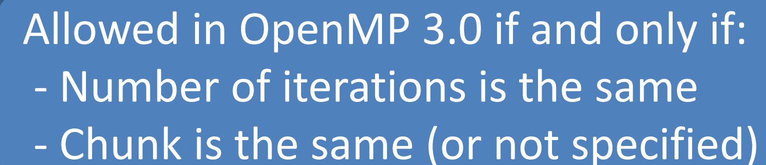
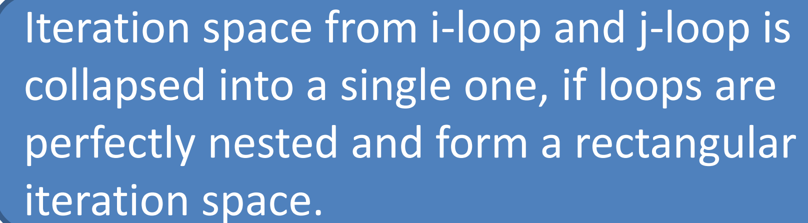
#pragma omp for nowait
for (i = 0; i < 100; i++) {
   b[i] = c[i];
}

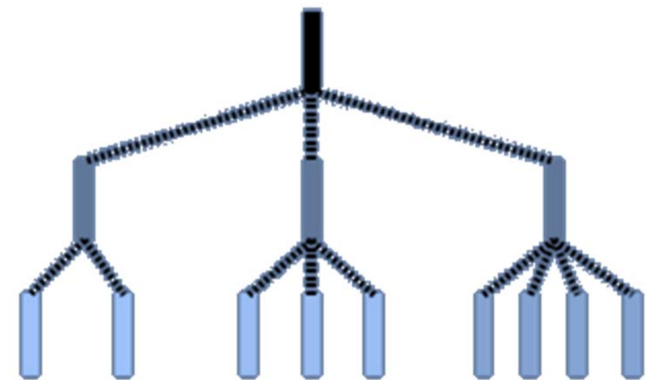
} // end omp parallel
```

Be careful using `nowait`: This code is only *correct* when both parallel loops have identical boundaries and use the same scheduling scheme.

There is always a Barrier at the end of a Parallel Region.

Introduction to OpenMP - Contents

RWTH AACHEN UNIVERSITY

- **Introduction to OpenMP**
  - Worksharing
  - Scoping
  - Synchronization
  - Advanced Worksharing
  - Miscellaneous
- **Tasks**
  - Case Study: Traversing a Tree

Runtime Library

- **C and C++:**
  - If OpenMP is enabled during compilation, the preprocessor symbol `_OPENMP` is defined.
  - To use the OpenMP runtime library, the header `omp.h` has to be included.
  - `omp_set_num_threads(int)`: The specified number of threads will be used for the parallel region encountered next.
  - `int omp_get_num_threads()`: Returns the number of threads in the current team.
  - `int omp_get_thread_num()`: Returns the number of the calling thread in the team, the Master has always the id 0.

- **Additional functions are available, e.g. to provide locking functionality.**

```
#pragma omp parallel for
for (int i = 0; i < 99; i++) {
   a[i] = b[i] + c[i];
}
```

Use the Runtime Library to distribute the work manually:

```
#pragma
for (int i = omp_get_thread_num(); i < 99;
     i += omp_get_num_threads()) {
   a[i] = b[i] + c[i];
}
```

- **Note: if only OpenMP constructs are used with suitable formatting, the program can still be compiled with a non OpenMP-aware compiler.**

Introduction: Runtime Library

```
#pragma omp parallel for
for (int i = 0; i < 99; i++) {
   a[i] = b[i] + c[i];
}
```

Use the Runtime Library to distribute the work manually:

```
#pragma omp parallel
for (int i = omp_get_thread_num(); i < 99;
     i += omp_get_num_threads()) {
   a[i] = b[i] + c[i];
}
```

- **Note: if only OpenMP constructs are used with suitable formatting, the program can still be compiled with a non OpenMP-aware compiler.**

The ordered construct

RWTH AACHEN UNIVERSITY

- **Allows to execute a structured block within a parallel loop in sequential order**

```
C/C++

#pragma omp ordered
... structured block ...
```

- **In addition, an `ordered` clause has to be added to the *Parallel Region* in which this construct occurs**

- **Can be used e.g. to enforce ordering on printing of data**

- **May help to determine whether there is a data race**

OpenMP Environment Variables

RWTH AACHEN UNIVERSITY

- `OMP_NUM_THREADS`: Controls how many threads will be used to execute the program.
- `OMP_SCHEDULE`: If the schedule-type *runtime* is specified in a schedule clause, the value specified in this environment variable will be used.
- `OMP_DYNAMIC`: The OpenMP runtime is allowed to smartly guess how many threads might deliver the best performance. If you want full control, set this variable to *false*.
- `OMP_NESTED`: Most OpenMP implementations require this to be set to *true* in order to enabled nested Parallel Regions.
  Remember: Nesting Worksharing constructs is not possible.

OpenMP API: Locks

- **OpenMP provides a set of low-level locking routines, similar to semaphores:**
  - `void omp_func_lock (omp_lock_t *lck)`, with func:
    - init / init_nest: Initialize the lock variable
    - destroy / destroy_nest: Remove the lock variable association
    - set / set_nest: Set the lock, wait until lock acquired
    - test / test_nest: Set the lock, but test and return if lock could not be acquired
    - unset / unset_nest: Unset the lock
  - Argument is address to an instance of `omp_lock_t` type
  - Simple lock: May not be locked if already in a locked state
  - Nested lock: May be locked multiple times by the same thread

Memory Model of OpenMP

RWTH AACHEN UNIVERSITY

- **OpenMP: Shared-Memory model**
  - All threads share a common address space (shared memory)
  - Threads can have private data (explicit user control)
  - Fork-Join execution model
- **Weak memory model**
  - Temporary View: Memory consistency is guaranteed only after certain points, namely implicit and explicit `flushes`
- **Any OpenMP `barrier` includes a `flush`**
- **Entry to and exit from `critical` regions include a `flush`**
- **Entry to and exit from lock routines (OpenMP API) include a `flush`**

The flush directive

RWTH AACHEN UNIVERSITY

C/C++

```
#pragma omp flush [(list)]
```

- **Enforces shared data to be consistent (but be cautious!)**
  - If a thread has updated some variables, their values will be flushed to memory, thus accessible to other threads
  - If a thread has not updated a value, the construct will ensure that any local copy will get latest value from memory

- **BUT: Do not use this for thread synchronization**
  - Compiler optimization might come in your way
  - Rather use OpenMP lock functions for thread synchronization

Summary

RWTH AACHEN UNIVERSITY

- OpenMP is a parallel programming model for Shared-Memory machines. That is, all threads have access to a *shared* main memory. In addition to that, each thread can have *private* data.
- The parallelism has to be expressed explicitly by the programmer. The base construct is a *Parallel Region*: A *Team* of threads is created by the runtime system.
- Using the available *Worksharing* constructs, the work can be distributed among the threads of a team, influencing the scheduling is possible.
- To control the parallelization, thread exclusion and synchronization constructs can be used.

Introduction to OpenMP - Contents

RWTH AACHEN UNIVERSITY

- **Introduction to OpenMP**
  - Worksharing
  - Scoping
  - Synchronization
  - Advanced Worksharing
  - Miscellaneous

- **Tasks**
  - Case Study: Traversing a Tree

How to parallelize a While-Loop?

RWTH AACHEN UNIVERSITY

- **How would you parallelize this code?**

```
typedef list<double> dList;
dList myList;
/* fill myList with tons of items */

dList::iterator it = myList.begin();
while (it != myList.end())
{
    *it = processListItem(*it);
    it++;
}
```

- **One possibility: Create a fixed-sized array containing all list items and a parallel loop running over this array**
  Concept: *Inspector / Executor*

How to parallelize a While-Loop?

RWTH AACHEN UNIVERSITY

- Or: Use Tasking in OpenMP 3.0

```
#pragma omp parallel
{
   #pragma omp single
   {
     dList::iterator it = myList.begin();
     while (it != myList.end())
     {
            #pragma omp task
            {
                   *it = processListItem(*it);
            }
            it++;
     }
   }
}
```

- All while-loop iterations are independent from each other!

Biggest change in OpenMP 3.0: Tasks

RWTH AACHEN UNIVERSITY

- **Tasks allow to parallelize irregular problems, e.g.**
  - unbounded loops
  - recursive algorithms
  - Producer / Consumer patterns
  - and more …
- ***Task*: A work unit which execution may be deferred**
  - Can also be executed immediately
- **Tasks are composed of**
  - Code to execute
  - Data environment
  - Internal control variables (ICV)

Tasks in OpenMP: Overview

RWTH AACHEN UNIVERSITY

- **Tasks are executed by the threads of the *Team***
- **Data environment of a *Task* is constructed at creation time**
- **A Task can be tied to a thread – only that thread may execute it – or untied**
- **Tasks are either implicit or explicit**
- **Implicit tasks: The thread encountering a *Parallel* construct**
  - Creates as many implicit Tasks as there are threads in the Team
  - Each thread executes one implicit Task
  - Implicit Tasks are tied

  → Different description than in 2.5, but equivalent semantics!

The task directive

C/C++

```
#pragma omp task [clause [[,] clause] ... ]
... structured block ...
```

- **Each encountering thread creates a new Task**
  - Code and data is being packaged up
  - Tasks can be nested
    - Into another Task directive
    - Into a Worksharing construct
- **Data scoping clauses:**
    - `shared`(*list*)
    - `private`(*list*)
    - `firstprivate`(*list*)
    - `default`(*shared | none*)

Task synchronization (1 / 2)

RWTH AACHEN UNIVERSITY

- **At OpenMP `barrier` (implicit or explicit)**
  - All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit

- **Task barrier: `taskwait`**
  - Encountering Task suspends until child tasks are complete
    - Only direct childs, not descendants!

C/C++

```
#pragma omp taskwait
```

Task synchronization (2 / 2)

RWTH AACHEN UNIVERSITY

- **Simple example of Task synchronization in OpenMP 3.0:**

```
#pragma omp parallel num_threads(np)
{
#pragma omp task
   function_A();
#pragma omp barrier
#pragma omp single
   {
#pragma omp task
      function_B();
   }
}
```

np Tasks created here, one for each thread

All Tasks guaranteed to be completed here

1 Task created here

B-Task guaranteed to be completed here

Tasks in OpenMP: Data Scoping

RWTH AACHEN UNIVERSITY

- **Some rules from *Parallel Regions* apply:**
  - Static and Global variables are shared
  - Automatic Storage (local) variables are private

- **If no `default` clause is given:**
  - Orphaned Task variables are `firstprivate` by default!
  - Non-Orphaned Task variables inherit the `shared` attribute!

  → Variables are `firstprivate` unless `shared` in the enclosing context

- **So far no verification tool is available to check Tasking programs for correctness!**

Tasks in OpenMP: Scheduling

- **Default: Tasks are *tied* to the thread that first executes them → not neccessarily the creator. Scheduling constraints:**
  - Only the Thread a Task is tied to can execute it
  - A Task can only be suspended at a suspend point
    - Task creation, Task finish, `taskwait`, `barrier`
  - If Task is not suspended in a barrier, executing Thread can only switch to a direct descendant of all Tasks tied to the Thread

- **Tasks created with the `untied` clause are never tied**
  - No scheduling restrictions, e.g. can be suspended at any point
  - But: More freedom to the implementation, e.g. load balancing

Tasks in OpenMP: if clause

RWTH AACHEN UNIVERSITY

- **If the expression of an `if` clause on a *Task* evaluates to false**
  - The encountering Task is suspended
  - The new Task is executed immediately
  - The parent Task resumes when new Tasks finishes

  → Used for optimization, e.g. avoid creation of small tasks

- **If the expression of an `if` clause on a *Parallel Region* evaluates to false**
  - The Parallel Region is executed with a Team of one Thread only

  → Used for optimization, e.g. avoid going parallel

- **In both cases the OpenMP data scoping rules still apply!**

Task pitfalls (1 / 3)

- **It is the user's responsability to ensure data is alive:**

```
// within Parallel Region
void foo() {
    int a[LARGE_N];
    #pragma omp task
    {
        bar1(a);
    }
    #pragma omp task
    {
        bar2(a);
    }
}
```

Variable a has to be shared in order to prevent copying to task (default firstprivate).

If not shared: Parent Task may have exited foo() by the time bar() accesses a: a is variable of automatic storage duration and thus is disposed when foo() is exited.

Task pitfalls (2 / 3)

- **It is the user's responsability to ensure data is alive:**

```
// within Parallel Region
void foo() {
    int a[LARGE_N];
   #pragma omp task shared(a)
    {
       bar1(a);
    }
   #pragma omp task shared(a)
    {
       bar2(a);
    }
#pragma omp taskwait
}
```

Task pitfalls (3 / 3)

- **Examing your code thoroughly before using `untied` Tasks:**

```
int dummy;
#pragma omp threadprivate(dummy)

void foo() {dummy = …; }
void bar() {… = dummy; }

#pragma omp task untied
{
   foo();
   bar();
}
```



Other news in OpenMP 3.0 (1 / 5)

- **Static schedule guarantees**

```
#pragma omp for schedule(static) nowait
   for(i = 1; i < N; i++)
      a[i] = …
#pragma omp for schedule(static)
   for (i = 1; i < N; i++)
      c[i] = a[i] + …
```



Other news in OpenMP 3.0 (2 / 5)

- **Loop collapsing**

```
#pragma omp for collapse(2)
   for(i = 1; i < N; i++)
      for(j = 1; j < M; j++)
         for(k = 1; k < K; k++)
            foo(i, j, k);
```



Other news in OpenMP 3.0 (3 / 5)

RWTH AACHEN UNIVERSITY

- **New variable types allowed in `for`-*Worksharing***

```
#pragma omp for
for (unsigned int i = 0; i < N; i++)
    foo(i);


vector v;
vector::iterator it;
#pragma omp for
for (it = v.begin(); it < v.end(); it++)
    foo(it);
```

Legal in OpenMP 3.0:
- Unsigned integer types
- Pointer types
- Random access iterators (C++)

Other news in OpenMP 3.0 (4 / 5)

- **Improvements in the API for Nested Parallelism:**
  - How many nested Parallel Regions?
    - `int omp_get_level()`
  - How many nested Parallel Regions are active?
    - `int omp_get_active_level()`
  - Which thread-id was my ancestor, in given level?
    - `int omp_get_ancestor_thread_num(int level)`
  - How many Threads were in my ancestor‘s team, at given level?
    - `int omp_get_team_size(int level)`

- **This is now well-defined in OpenMP 3.0:**

```
omp_set_num_threads(3);
#pragma omp parallel {
  omp_set_num_threads(omp_get_thread_num() + 2);
  #pragma omp parallel {
     foo();
} }
```



Other news in OpenMP 3.0 (5 / 5)

RWTH AACHEN UNIVERSITY

- **Improved definition of environment interaction**
  - Env. Var. OMP_MAX_NESTED_LEVEL + API functions
    - Controls the maximum number of active parallel regions
  - Env. Var. OMP_THREAD_LIMIT + API functions
    - Controls the maximum number of OpenMP threads
  - Env. Var. OMP_STACKSIZE
    - Controls the stack size of child threads
  - Env. Var. OMP_WAIT_POLICY
    - Control the thread idle policy:
      - active: Good for dedicated systems
      - passive: Good for shared systems (e.g. in batch mode)

Introduction to OpenMP - Contents

RWTH AACHEN UNIVERSITY

- **Introduction to OpenMP**
  - Worksharing
  - Scoping
  - Synchronization
  - Advanced Worksharing
  - Miscellaneous
- **Tasks**
  - Case Study: Traversing a Tree

Traversing a Tree with Tasks

RWTH AACHEN UNIVERSITY

```
[…]
struct node
{
    public:
         node *left;
         node *right;
         int value;
};

void process(node*);

void traverse(node* p)
{
     if (p->left)
     {
               #pragma omp task
               traverse(p->left);
     }

     if (p->right)
     {
               #pragma omp task
               traverse(p->right);
     }

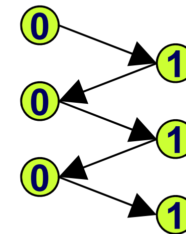
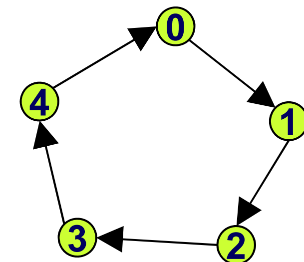
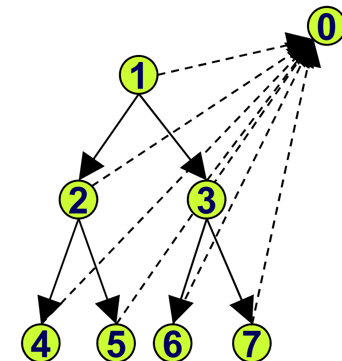
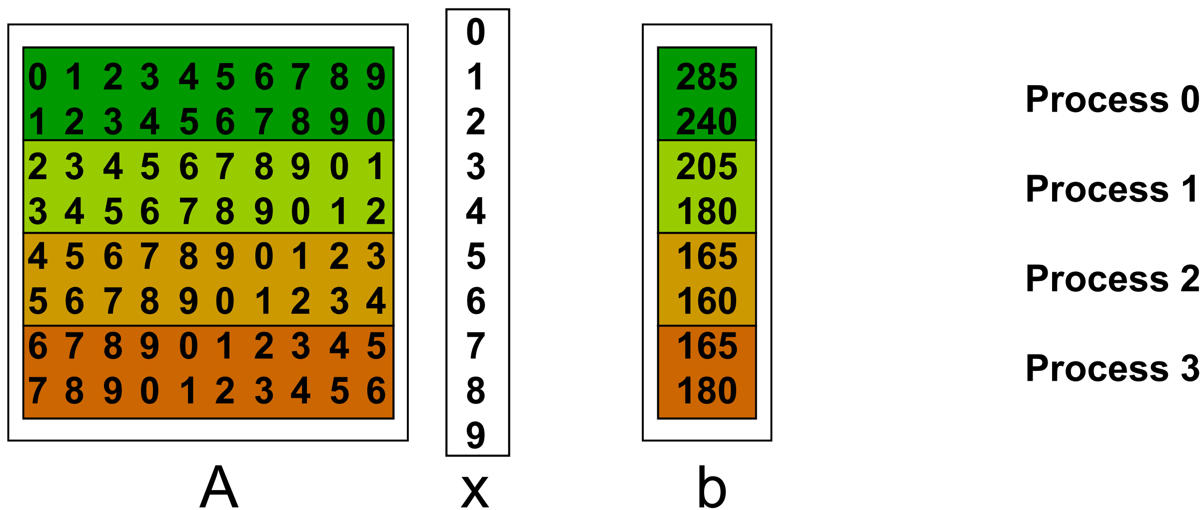
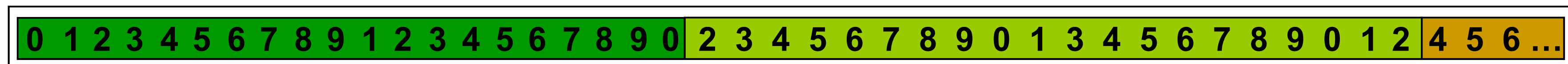
     process(p);
}
[…]
```

Traversing a Tree with Tasks: Postorder

RWTH AACHEN UNIVERSITY

```
[…]
struct node
{
    public:
        node *left;
        node *right;
        int value;
};

void process(node*);

void traverse(node* p)
{
    if (p->left)
    {
            #pragma omp task
            traverse(p->left);
    }

    if (p->right)
    {
            #pragma omp task
            traverse(p->right);
    }

    #pragma omp taskwait
    process(p);
}
[…]
```

Exercises

Java-Threads, MPI, OpenMP

Contents

- Lecture „Introduction to Parallel Programming“
  - Introduction
  - MPI - Part I – Basics
  - MPI - Part II – Advanced Topics
  - Introduction to OpenMP
- **Exercises**
  - Java-Threads
  - MPI
  - OpenMP
  - Hybrid

Exercise 1 – Prime Search I

RWTH AACHEN UNIVERSITY

- Write a program that reads one integer after another from the user via the keyboard. For each of this integers let the program check whether it is prime. Perform this check threaded so that reading integers from the keyboard can continue while checking. It is up to you which technique you use to realize the threaded check. Print for each integer whether it is prime or not.

- *Hint for all Java-Threads exercises:*
  - *Try to use large numbers so that the check takes a while. See what happens for the different solutions.*

Exercise 2 – Prime Search II

RWTH AACHEN UNIVERSITY

- Write another program similar to the first one. This time use only one thread, that performs the prime check in the background. This thread is started at the beginning of the program. Each new integer is passed to the thread, which checks all passed integers one after another and prints the result. Think on your own how you can realize this behavior and implement it in at least **two** different ways.

- Do **not** use a thread pool.

Exercise 3 – Prime Search III

RWTH AACHEN UNIVERSITY

- Write a third program that behaves identically to that one from the second task, except that it uses several threads to check if the entered numbers are prime. This time use a threadpool to solve the exercise.

Contents

RWTH AACHEN UNIVERSITY

- **Lecture „Introduction to Parallel Programming“**
  - Introduction
  - MPI - Part I – Basics
  - MPI - Part II – Advanced Topics
  - Introduction to OpenMP
- **Exercises**
  - Java-Threads
  - MPI
  - OpenMP
  - Hybrid

Exercise 1 – Using cluster-linux

RWTH AACHEN UNIVERSITY

- Download and install a SSH-client
  - http://pp.bastian-kueppers.de
  - Pay attention on choosing the right platform!
- Log in to the RWTH Aachen Linux Cluster
  - Host: cluster-linux-xeon.rz.rwth-aachen.de
  - User: Your TIM-ID
  - Password: Your password
  - Hint: If you cannot log in, you first have to enable the cluster-account in TIM
- Create a directory for your exercises in your home directory

Exercise 2 – Hello World

RWTH AACHEN UNIVERSITY

Implement the classical "Hello World" C-program:

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

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

- Create a `hello.c´ file containing the source code above.
- Compile, link and run your program; name it `hello´.
  - Use *gcc hello.c –o your_executable* to compile your program
- Run your program
  - Use *./your_executable* to run your program

Exercise 3 – The first MPI-Program

RWTH AACHEN UNIVERSITY

- **Enhance your "Hello World" program and use the MPI lib:**
  - Start with your "Hello World" program from Exercise 02.
  - 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

RWTH AACHEN UNIVERSITY

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

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

- **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

RWTH AACHEN UNIVERSITY

- **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.

Exercise 8 – Integer Array I

RWTH AACHEN UNIVERSITY

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

RWTH AACHEN UNIVERSITY

- **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.

Exercise 12 – Numerical Integration

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

RWTH AACHEN UNIVERSITY

- **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´.

Exercise 13 – Matrix Vector Multiplication

RWTH AACHEN UNIVERSITY

- **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! */

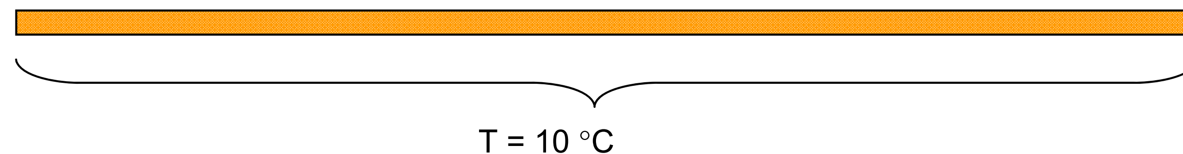
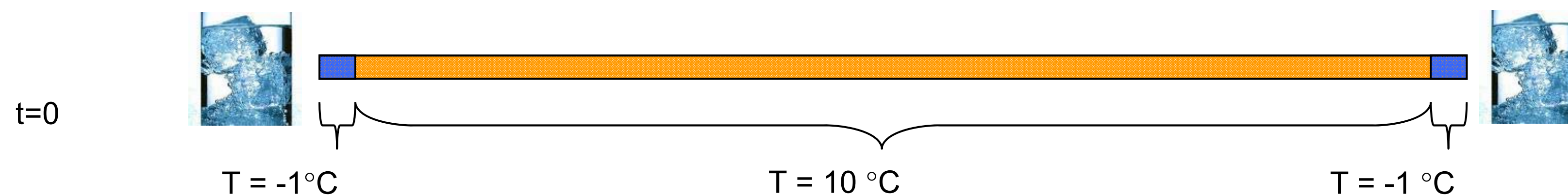
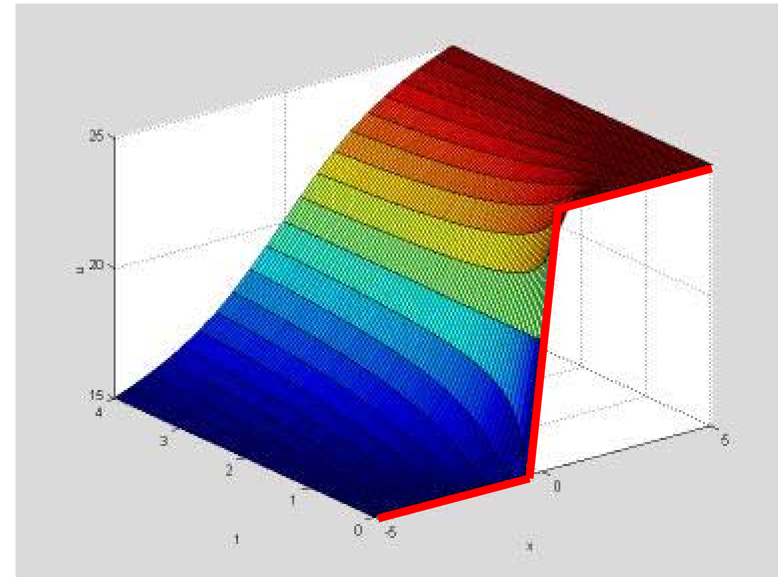
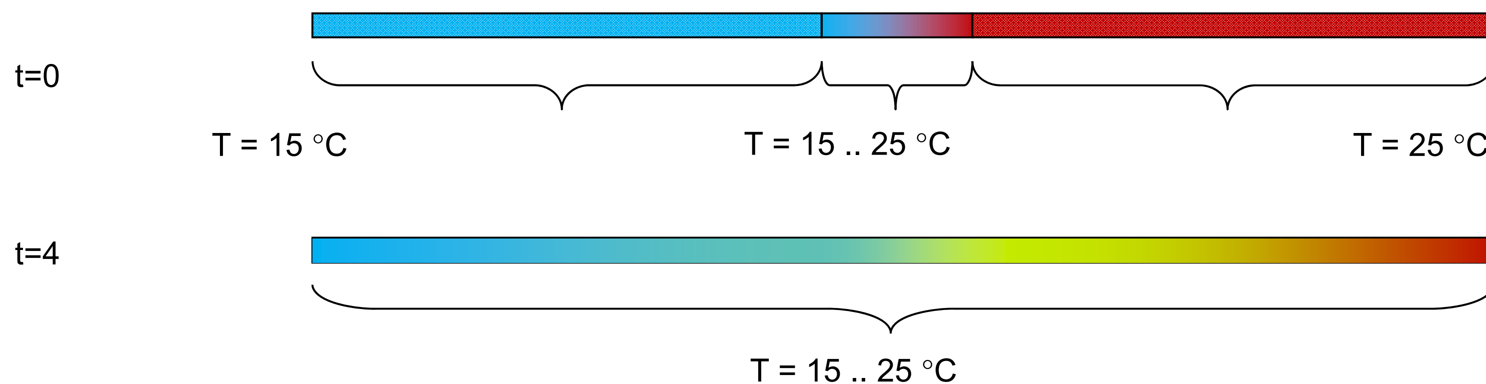
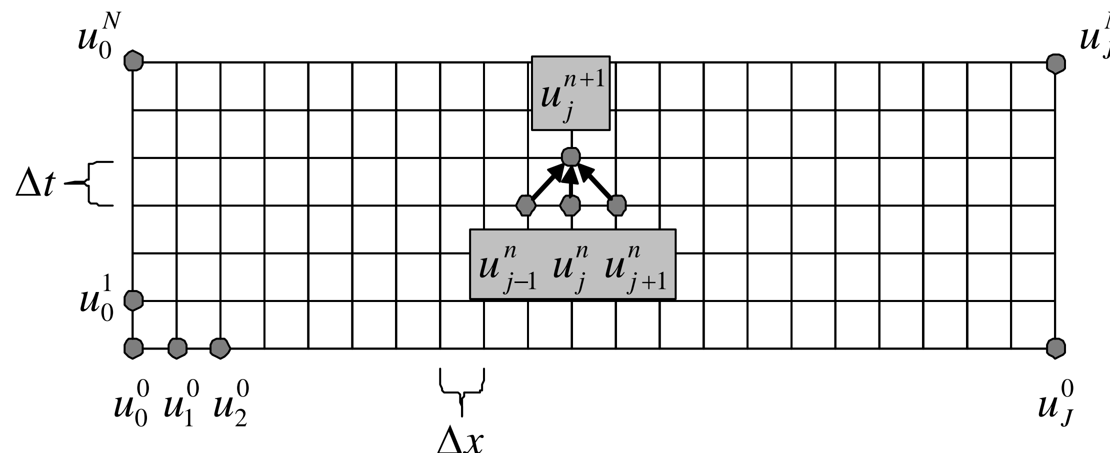
#include <mpi.h>
#include <stdio.h>

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);

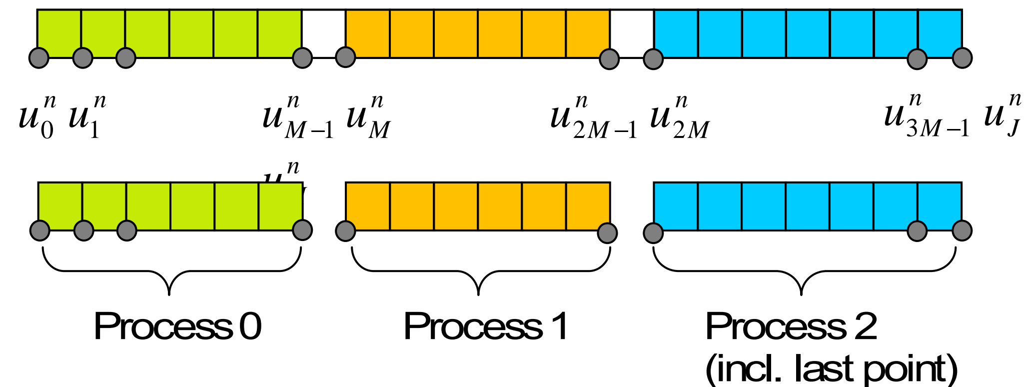
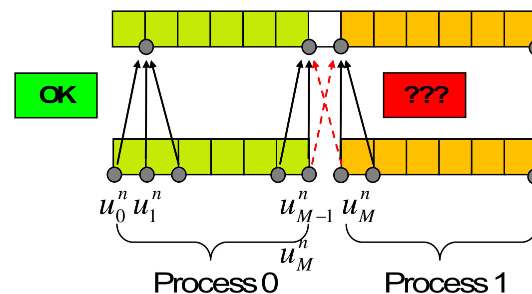
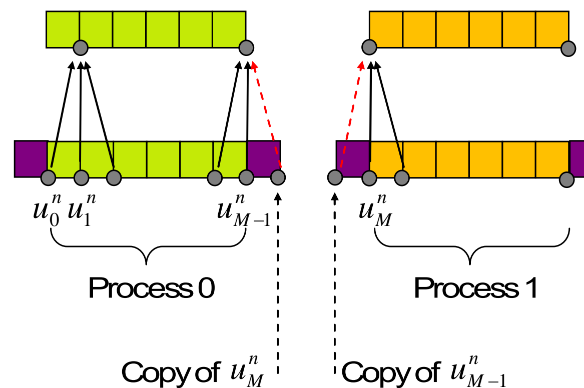
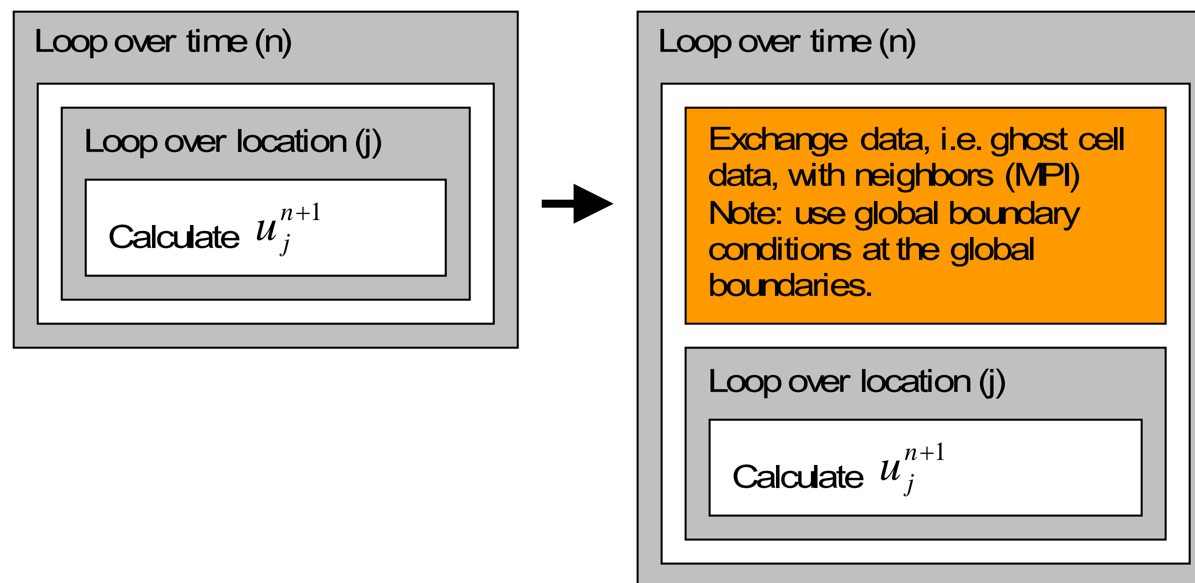
    if (data != 399) fprintf(stdout, "Whoa! The data is incorrect\n");
    else             fprintf(stdout, "Whoa! Got the message ... \n");

    return 0;
}
```

Exercise 15 – Measure Test-Receive Speed

RWTH AACHEN UNIVERSITY

- **Count how often you may call MPI_Test (polling) in one second waiting for a message:**
  - Use two processes. One sends a message the other waits for the message in non-blocking mode.

Exercise 16 – Vector Datatype

RWTH AACHEN UNIVERSITY

- **Rewrite your solution of Exercise 13 using self-defined datatypes:**
  - Define a MPI-datatype \`row_type´ that represents a complete row in A.
  - Send elements of this \`row_type´ instead of doubles, i.e. use this type for scatter and gather operations.

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	0
2	3	4	5	6	7	8	9	0	1
3	4	5	6	7	8	9	0	1	2
4	5	6	7	8	9	0	1	2	3
5	6	7	8	9	0	1	2	3	4
6	7	8	9	0	1	2	3	4	5
7	8	9	0	1	2	3	4	5	6

One element of type \`row_type´!

Exercise 17 - Communicator

RWTH AACHEN UNIVERSITY

- **Define Communicators:**
  - Define a communicator with exactly 20 processes and divide them into groups of 5 (the first communicator contains (old) rank 0..4, etc.)
  - Start your program with 25 processes.

Exercise 18 – Heat Equation (1 / 10)

RWTH AACHEN UNIVERSITY

- The heat equation is a so called partial differential equation describing the distribution of heat or variation in temperature in a given region over time. It reads

$$u_t = k^2 u_{xx}$$

- Here k denotes some material constant (set $k = 1.0$ here) and $u(x, t)$ a function of one spatial variable x and time variable t, representing the temperature at a point x at time t.

Exercise 18 – Heat Equation (2 / 10)

RWTH AACHEN UNIVERSITY

- **Example 1:**
  - Imagine a thin rod that is given an initial temperature distribution.



  - Now the ends of the rod are kept at a fixed (and different) temperature; e.g., suppose at the start of the experiment ($t = 0$), both ends are immediately plunged into ice water or held against something cold (boundary conditions).



  - We are interested in how the temperatures along the rod vary with time, i.e. we look for $u(x, t)$ for $t > 0$.

RWTH AACHEN UNIVERSITY

- **Example 2:**
  - $u(x, t)$ as 2D-function





- **Numerical Solution:**
  - On the computer we can only keep track of the temperature *u* at a discrete set of times and a discrete set of positions …

$$u_j^n = u(j\Delta x, n\Delta t)$$

  - … for *j* in *{0,…,J}* and *n* in *{0,…,N}* with some constants *N* and *J*.



Exercise 18 – Heat Equation (5 / 10)

- **Numerical Solution:**
  - Rewriting the partial differential equation in terms of *finite difference approximations* to the derivatives, we get

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = k\,\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2}$$

  - These are the simplest approximations. Thus if for a particular n, we know the values of for all j, we can solve the equation above to find for each j (see figure above):

$$u_j^{n+1} = u_j^n + k\,\frac{\Delta t}{\Delta x^2}\,(u_{j+1}^n - 2u_j^n + u_{j-1}^n)$$

  - In other words, this equation tells us how to find the temperature distribution at time step $n+1$ given the temperature distribution at time step n. At the endpoints $j = 0$ and $j = J$ we ignore the equation above and apply the boundary conditions instead.

Exercise 18 – Heat Equation (6 / 10)

RWTH AACHEN UNIVERSITY

Code-Snippet:

Serial Code

```
const double TempLeft  = -1.0;
const double TempRight = -1.0;
const double TempMid   = 10.0;

const int N = 1000; // time steps
const int J =  100; // space discretization
[…]
double dx = 1.0/J;
double dt = 0.5*dx*dx;
[…]
double* uk0 = malloc((J+1)*sizeof(double)); // take right point into account
double* uk1 = malloc((J+1)*sizeof(double)); // we have u[0],u[1],...,u[J]
double* ukt; // swap fields

InitFields(uk0,uk1,J+1, 0,TempLeft, J,TempRight, TempMid);
AppendFile("data0", uk1,0.0, 0,J+1);

for (int n=1; n<=N; ++n)  // all time steps
{
  for (int j=1; j<J; ++j) // all locations, 1..J-1
  {
    uk1[j] = uk0[j] + dt/(dx*dx) * (uk0[j-1]-2*uk0[j]+uk0[j+1]);
  }

  ukt = uk0; uk0 = uk1; uk1 = ukt;
}
```

RWTH AACHEN UNIVERSITY

Parallelization:

- Obviously, the inner loop (j) can be easily parallelized to P processes (assume the number of cells is divisible without remainder). We set $M = J/P$.



- Example above: $J = 21, P = 3, M = 7$

- Chosen that partition scheme every process can compute all inner points individually!

Exercise 18 – Heat Equation (8 / 10)

- **Parallelization:**
  - There is only one problem left, namely getting the endpoint-values, or local boundary points, of each partition.



  - One solution to this problem is to introduce additional cells, so-called *ghost cells*, being copies of the boundary cells of the neighbour processes, cf. the following figure.



Exercise 18 – Heat Equation (9 / 10)

- **Parallelization:**
  - Then we have to change the loops as follows:



Exercise 18 – Heat Equation (10 / 10)

RWTH AACHEN UNIVERSITY

- **Exercise:**
  - Extend the serial code such that the problem can be computed parallel.
  - Use non-blocking communication.

Contents

RWTH AACHEN UNIVERSITY

- **Lecture „Introduction to Parallel Programming"**
  - Introduction
  - MPI - Part I – Basics
  - MPI - Part II – Advanced Topics
  - Introduction to OpenMP
- **Exercises**
  - Java-Threads
  - MPI
  - OpenMP
  - Hybrid

Exercise 1 – Hello World revisited (1 / 2)

RWTH AACHEN UNIVERSITY

- **Implement the classical "Hello World" C-program:**

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

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

  - Create a `hello.c` file containing the source code above
  - Extend it using the OpenMP-Library so that it runs with 4 threads
  - Let each thread print out "Hello World from thread #*threadid* of #*num_threads* threads"

Exercise 1 – Hello World revisited (1 / 2)

RWTH AACHEN UNIVERSITY

- **Implement the classical "Hello World" C-program:**
  - In which order do you expect the threads to print out their Hello World message?
  - What are the three different ways of forcing OpenMP to use 4 threads? In which order do they apply?

Exercise 2 – Numerical Integration revisited

RWTH AACHEN UNIVERSITY

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:

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

- Parallelize the given Sourcecode with OpenMP. Look out for the computationally most intensive part on your own.

Exercise 3 – Counting Numbers

RWTH AACHEN UNIVERSITY

- **Count the occurrence of Numbers in an Array**
  - Parallelize the given Sourcecode with OpenMP. Look out for the computationally most intensive part on your own
  - Is it possible to parallelize the output of the given numbers? If you think it's possible, do it using OpenMP. Otherwise state why you think it's not possible.

Execise 4 – Finding the Maximum

RWTH AACHEN UNIVERSITY

- **Find the maximum over all numbers within an Array**
  - Parallelize the given sourcecode using OpenMP.
    - Hint: Think about different ways of doing it.
  - Explain why using a critical region for parallelizing the program most likely is not a good idea.

Exercise 5 – OpenMP Puzzles (1 / 3)

RWTH AACHEN UNIVERSITY

- **The following declarations and definitions occur in all exercises before the Parallel Region:**

  - ```
    int i;
    double A[N] = { ... }, B[N] = { ... }, C[N], D[N];
    const double c = ...;
    const double x = ...;
    double y;
    ```

- **Insert missing OpenMP directives to parallelize this loop:**

  - ```
    for (i = 0; i < N; i++)
    {
        y = sqrt(A[i]);
        D[i] = y + A[i] / (x * x);
    }
    ```

Exercise 5 – OpenMP Puzzles (2 / 3)

RWTH AACHEN UNIVERSITY

- **Insert missing OpenMP directives to make both loops run in parallel:**

```
#pragma omp parallel
{

  for (i =_________________________; i < N; i +=_______________)
  {
    D[i] = x * A[i] + x * B[i];
  }

  #pragma omp for
  for (i = 0; i < N; i++)
  {
    C[i] = c * D[i];
  }

} // end omp parallel
```

Exercise 5 – OpenMP Puzzles (3 / 3)

RWTH AACHEN UNIVERSITY

- Can you parallelize this loop – if yes how, if not why?
  - ```
    #pragma omp parallel for
    for (int i = 1; i < N; i++)
    {
        A[i] = B[i] - A[i - 1];
    }
    ```
    ```

Exercise 6 – Fibonacci

RWTH AACHEN UNIVERSITY

- **Compute a Fibonacci-sequence**
  - Parallelize the given Sourcecode with OpenMP. Think of the special case of parallelizing a recursive algorithm.

Contents

RWTH AACHEN UNIVERSITY

- **Lecture „Introduction to Parallel Programming“**
  - Introduction
  - MPI - Part I – Basics
  - MPI - Part II – Advanced Topics
  - Introduction to OpenMP
- **Exercises**
  - Java-Threads
  - MPI
  - OpenMP
  - Hybrid

Exercise 1 – Primesearch (1 / 2)

- **Write a program that searches for primes. It should obey the following restrictions:**
  - The user specifies a maximum value n. All $i \in \mathbb{N}, 2 \leq i \leq n$ shall be checked if they are prime.
  - The program should implement a master-worker-pattern (MPI) that works the following way:
    - The master sends all numbers to be checked to the workers, collects the results of their computations and prints these results altogether at the end of the program.

Exercise 1 – Primesearch (2 / 2)

- **Write a program that searches for primes. It should obey the following restrictions:**
  - The worker should check if a number that was sent by the master is prime or not. This should be done in a multithreaded way (OpenMP). The user should be able to pass the number of threads to be used to the program. At the end of the computation the worker sends back to the master if the given number is prime or not.
  - Think of an efficient way for the communication between the master and the workers.