# *Parallele Programmierung*
# Cluster Parallelisierung

Vorlesung 12

**Prof. Dr. Luc Bläser**

# Letzte Vorlesung - Quiz

*Welche Performance-Aspekte mussten wir bei der GPU-Parallelisierung beachten?*

# Inhalt Heute

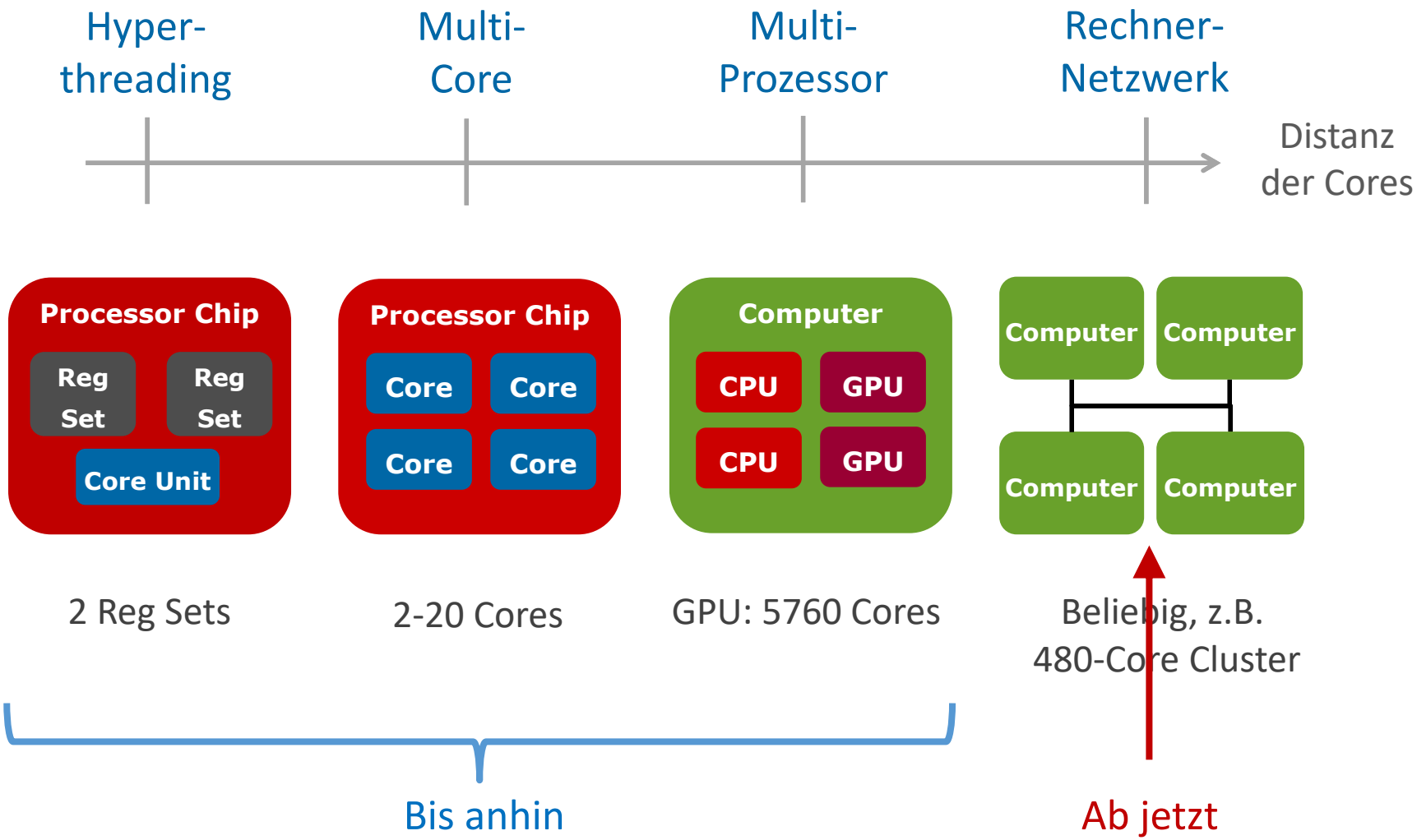
Cluster-Parallelisierung

- HPC Cluster Architektur
- MPI (Message Passing Interface)

# Lernziele

- Cluster für verteilte Parallelisierung nutzen können
- Parallelprogrammierung mit MPI kennenlernen

# Stufen der Parallelisierung



# Motivation

- Möglichst hohe parallele Beschleunigung
  - Faktor 100 und mehr
- General-Purpose Programming
  - Viele CPU Cores statt nur viele GPU Cores
  - GPUs sind wegen SIMD oft zu einschränkend
  - Kombination auch möglich

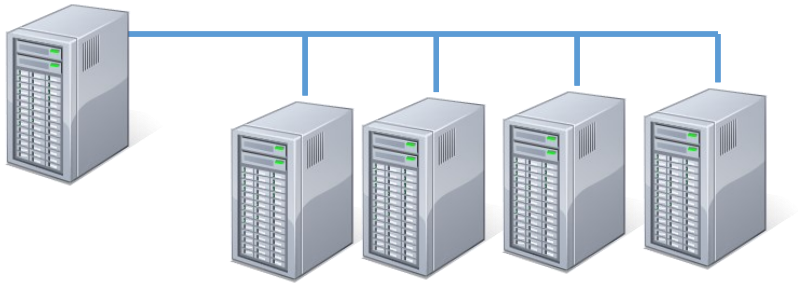
# Computer Cluster

Verbund leistungsfähiger Rechenknoten

- Meist gleichartige Rechnerknoten
- Fest verbunden an einem Standort
- Sehr schnelles Interconnect, z.B. 100GBit/s Switch

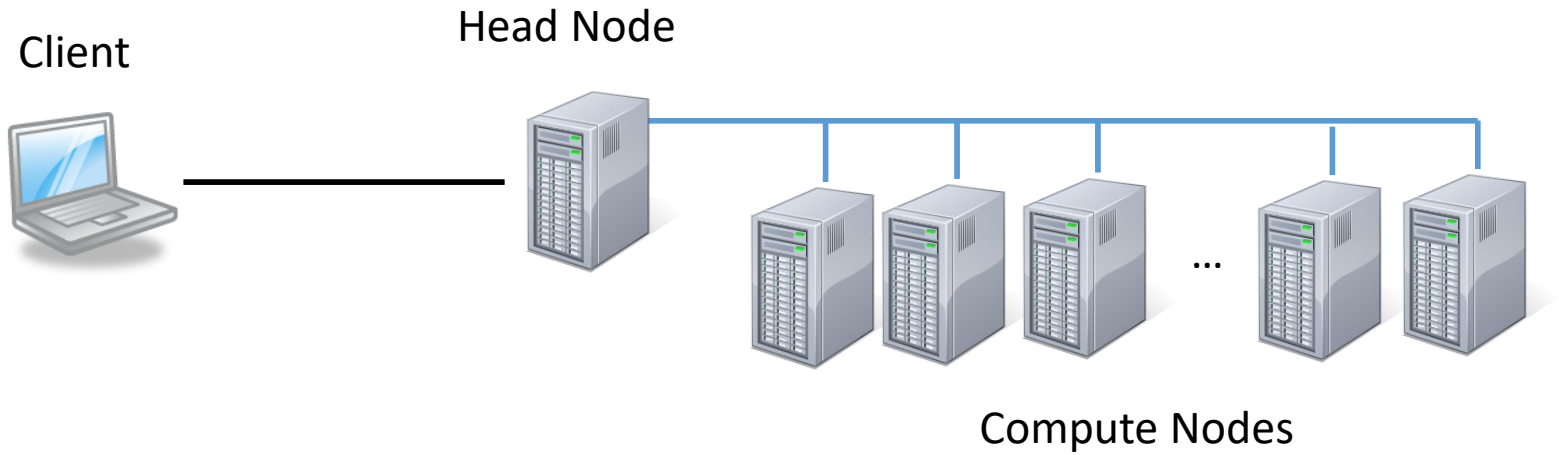
Domäne: wissenschaftliches Rechnen

- HPC: High-Performance Computing
- Grosse Simulationen: Fluids, Meteo, Traffic etc.



# OST Cluster (ehemals HSR)

- 24 Compute Nodes mit je 20 CPU Cores zu je 2.2GHz
- Total 480 echte Cores, 960 Hyperthreading Cores
- Linux Cluster



# HPC Anleitung

1. Programcode uploaden
   - `HelloCluster.c` zu File-Share
2. Auf Cluster kompilieren
   - `mpicc HelloCluster.c`
3. HPC-Job lancieren (z.B. 32 Cores)
   - `psub-parprog –f a.out –c 32`
4. Job-Ende abwarten
   - `squeue`
5. Job-Resultat anschauen
   - Slurm-Datei auf File-Share

# Cluster-Parallelisierung

## Ziel: Ein Programm auf mehreren Nodes ausführen

*Wieso kann man ein Java, C#, C/C++ Programm nicht automatisch auf mehreren Nodes verteilt ausführen?*

# Verteiltes Programmiermodel

Programm auf mehreren Nodes ausführen

- Kein Shared Memory (NUMA) zwischen Nodes
- Shared Memory (SMP) für Cores innerhalb Node

# Message Passing Interface (MPI)

Verteiltes Programmiermodell

- MPI basiert auf Actor/CSP-Prinzip (späteres Thema)
- Übliche Wahl für Parallelisierung auf Cluster

Industrie-Standard einer Library

- Meist für C, Fortran, auch für .NET, Java u.a.
- Diverse Implementierungen (z.B. Open MPI, MPICH)
- Aktuell: Version 3.1 (2015)
- Release Candidate 4.0 (2020)

https://www.mpi-forum.org

# MPI – Erstes Programm

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

int main(int argc, char* argv[]){
  MPI_Init(&argc, &argv);

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("MPI process %i", rank);

  MPI_Finalize();

  return 0;
}
```

MPI Initialisierung

Prozess Identifikation

MPI Finalisierung

# Mit Prozessor-Namen

```
int main(int argc, char* argv[]){
  MPI_Init(&argc, &argv);

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  char name[MPI_MAX_PROCESSOR_NAME];
  int len;
  MPI_Get_processor_name(name, &len);

  printf("MPI process %i on %s\n", rank, name);

  MPI_Finalize();

  return 0;
}
```

# Kompilieren & Ausführen

```
mpicc HelloCluster.c
```

GCC mit OpenMpi

```
psub-parprog –f a.out –c 32
```

OST-Cluster-spezifisch, führt folgendes aus:
`mpiexec -n 32 a.out`

# MPI Programmausführung

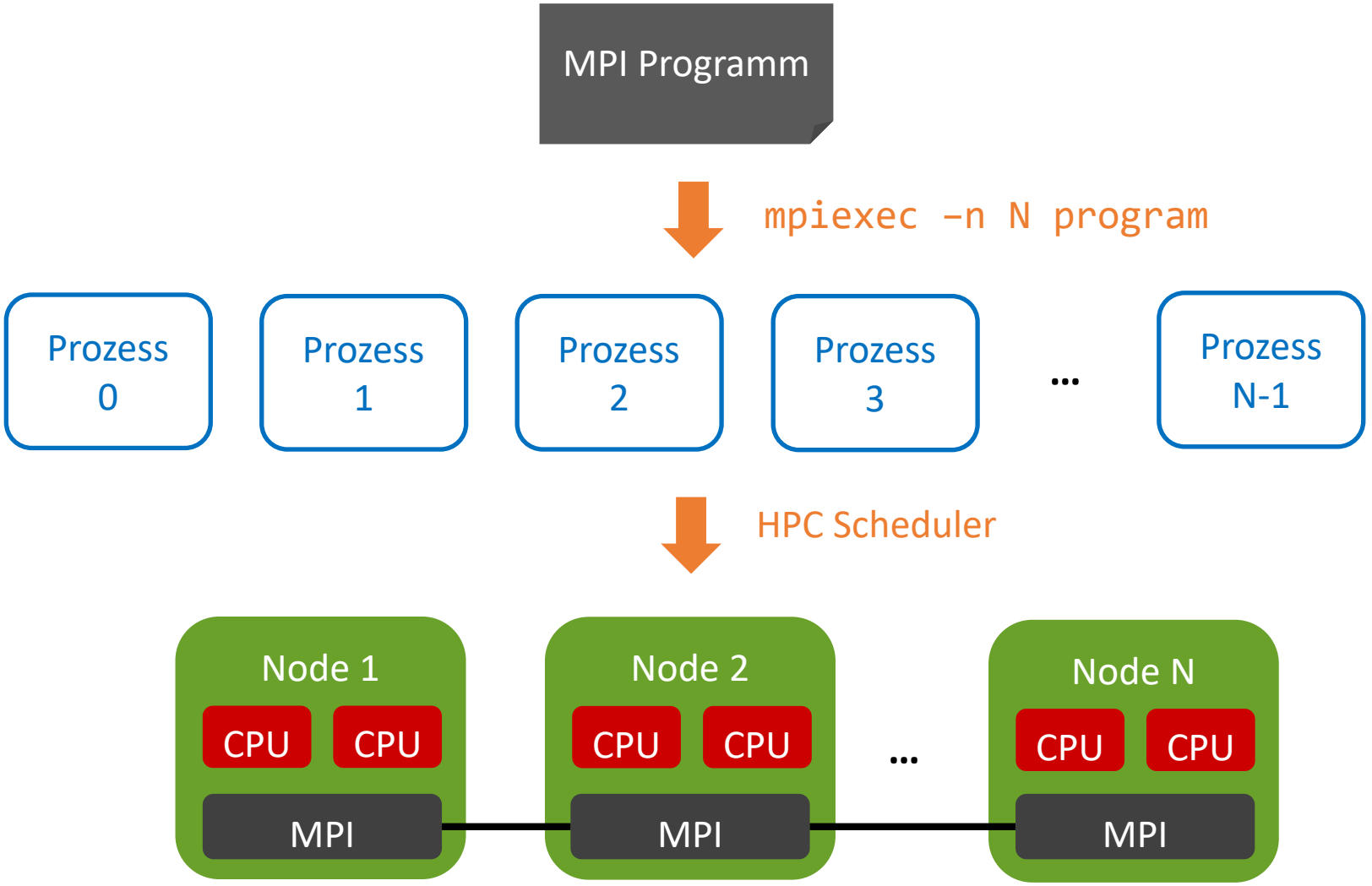
```
mpiexec -n 32 a.out
```

32 Prozessinstanzen

```
MPI process 31 on node03
MPI process 29 on node03
MPI process 13 on node02
MPI process 11 on node02
...
MPI process 1 on node02
MPI process 20 on node03
MPI process 0 on node02
```

Auf beliebige Nodes & Cores verteilbar

# Verteilte Ausführung



# Single Program Multiple Data

MPI Programm wird in mehrere Prozesse gestartet

- Jeder Prozess hat seine Identifikation (Rank)
- Prozess arbeiten unabhängig in ihrem Adressraum
- Alle Prozesse starten und terminieren synchron

Prozesse können untereinander kommunizieren

- Senden und Empfangen von Nachrichten
- Synchronisation mit Barrieren

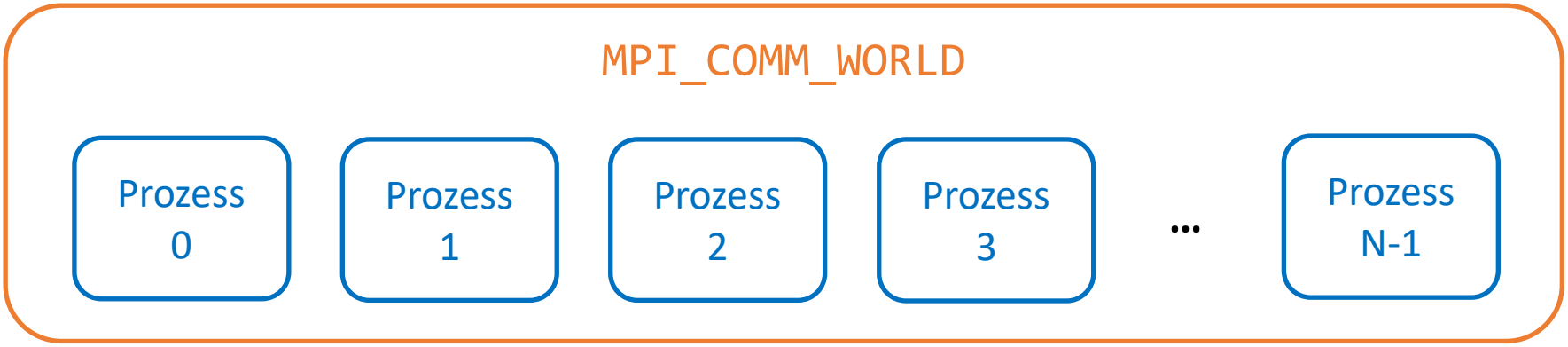
# Communicator

Gruppe von MPI-Prozessen

- Erlaubt Kommunikation zwischen Prozessen

Communicator World

- Alle Prozesse einer MPI-Programmausführung
- Eigene Gruppen definierbar



# Prozess Identifikation

- Rank = Nummer innerhalb einer Gruppe
  - Aufsteigende Nummerierung von 0
- Eindeutige Identifikation = (Rank, Communicator)

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```
Prozess-Nummer (0..size-1)

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```
Gesamtanzahl der Prozesse

# Direkte Kommunikation

Send

```
MPI_Send(&value, 1, MPI_INT, receiverRank, tag,
                                          MPI_COMM_WORLD);
```

Receive

```
MPI_Recv(&value, 1, MPI_INT, senderRank, tag,
                          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Hier Senden/Empfangen eines Int-Wertes (`int value`)
tag: Frei wählbare Nummer für Nachrichtenart (>= 0)
status: Fehlerinformationen (hier ignoriert)

# Beispiel: Senden und Empfangen

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (rank == 0) {
  int value = rand(); int to;
  for (to = 1; to < size; to++) {
    MPI_Send(&value, 1, MPI_INT, to, 0, MPI_COMM_WORLD);
  }
} else {
  int value;
  MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                                     MPI_STATUS_IGNORE);
  printf("%i received by %i", value, rank);
}
```

*Was bewirkt dieses MPI-Programm?*

# Verschiedene Verhalten

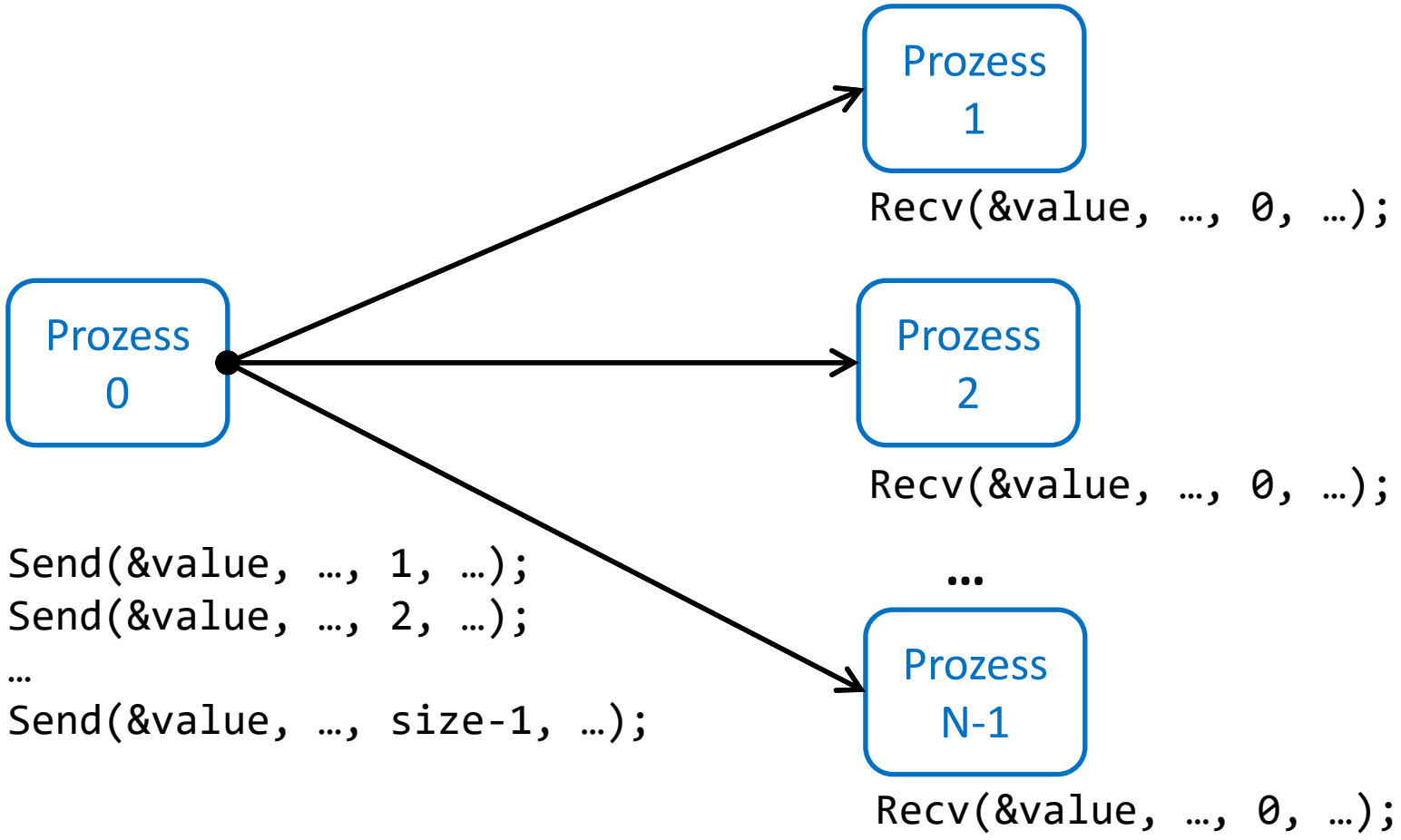
Prozess 0

```
int value = rand(); int to;
for (to = 1; to < size; to++) {
  MPI_Send(&value, …, to, …);
}
```

Prozesse 1 bis N-1

```
int value;
MPI_Recv(&value, …, 0, …);
printf(…);
```

# Kommunikationsablauf



# MPI-Datentypen

| MPI Datentyp | C Datentyp |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | short |
| MPI_INT | int |
| MPI_LONG | long |
| MPI_LONG_LONG | long long |
| MPI_UNSIGNED | unsigned int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| … | |

Es können auch massgeschneiderte MPI-Typen definiert werden (MPI_Type_contiguous)

# Array übermitteln

```
int array[LENGTH];
```

Send

```
MPI_Send(array, LENGTH, MPI_INT, receiverRank, tag,
                                                MPI_COMM_WORLD);
```

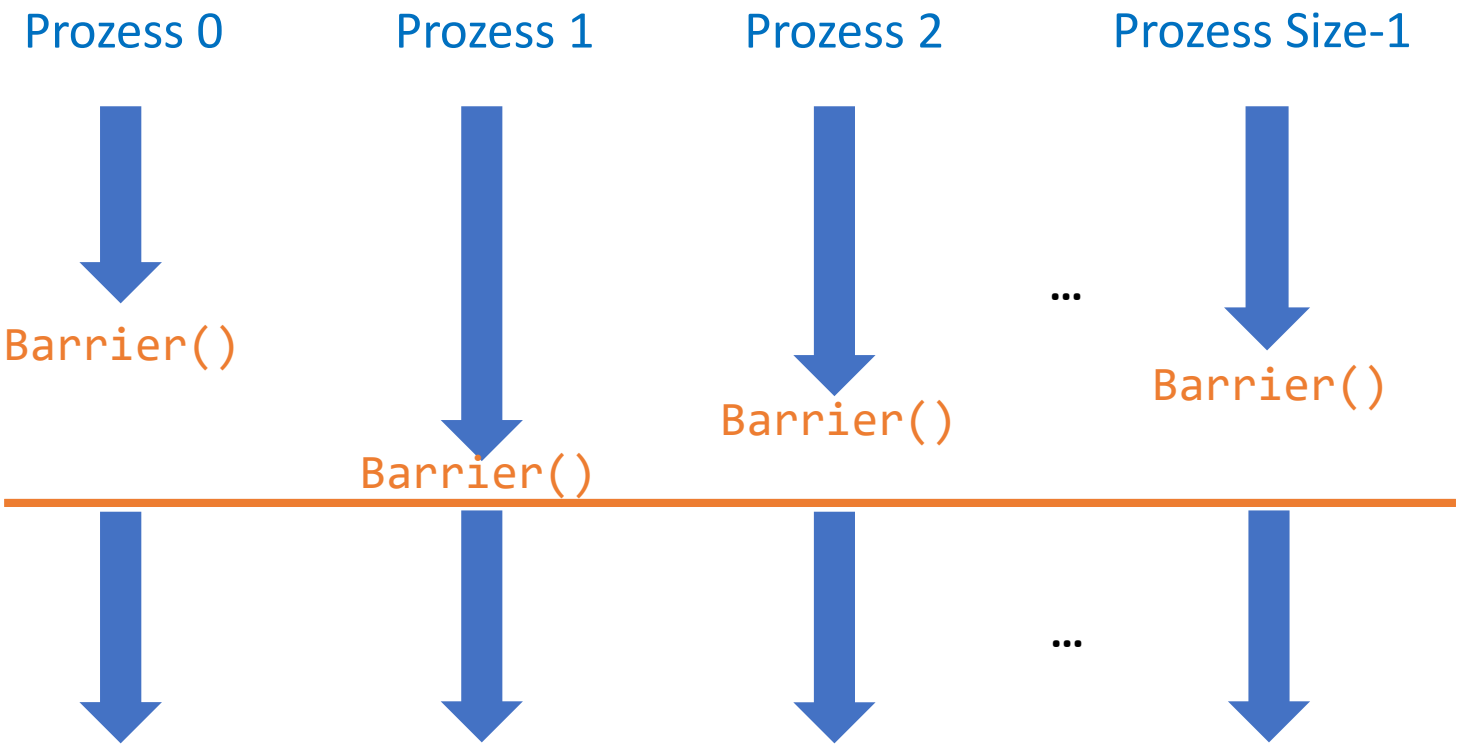
Receive

```
MPI_Recv(array, LENGTH, MPI_INT, senderRank, tag,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

# MPI Barriere

`MPI_Barrier(MPI_COMM_WORLD)`

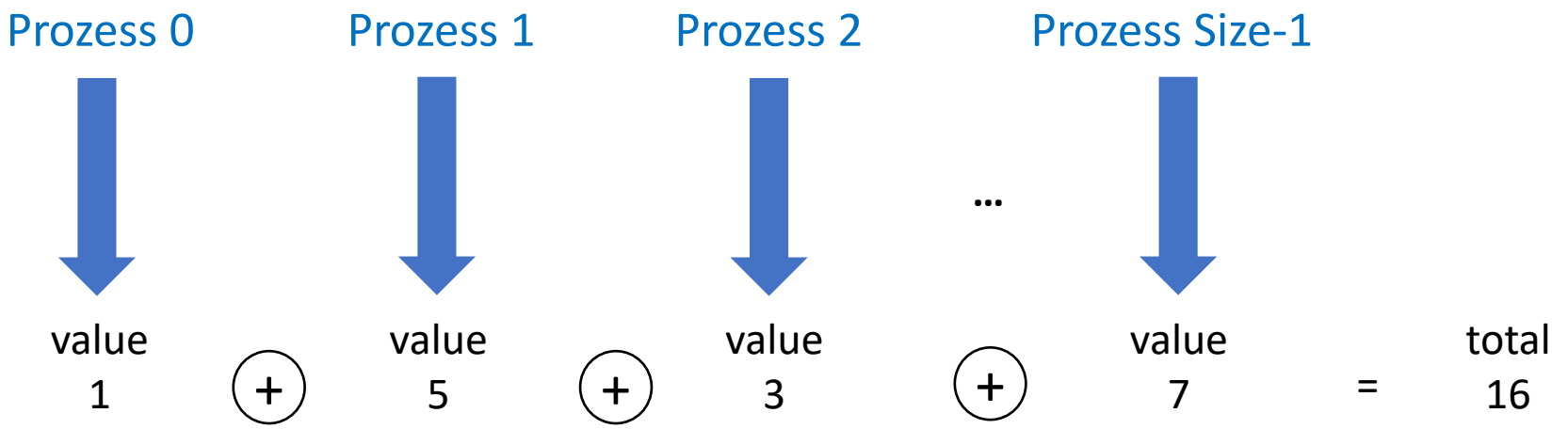
- Warte, dass alle Prozesse Barriere erreichen



# Reduktion

Aggregation von Teilresultate zwischen Prozessen

- `MPI_Allreduce(&value, &total, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD)`



# Aggregatoren

| MPI-Operator | |
|---|---|
| MPI_MAX | Max() |
| MPI_MIN | Min() |
| MPI_SUM | + |
| MPI_PROD | * |
| … | |

# Reduktions-Funktionen

## AllReduce

- Jeder erhält das Gesamtresultat als Rückgabewert
- Implizite Barriere/Broadcast über alle Prozesse

## Reduce

- Nur ein Prozess (rank) sieht das Gesamtresultat
- Effizienter als Allreduce, weil kein Broadcast

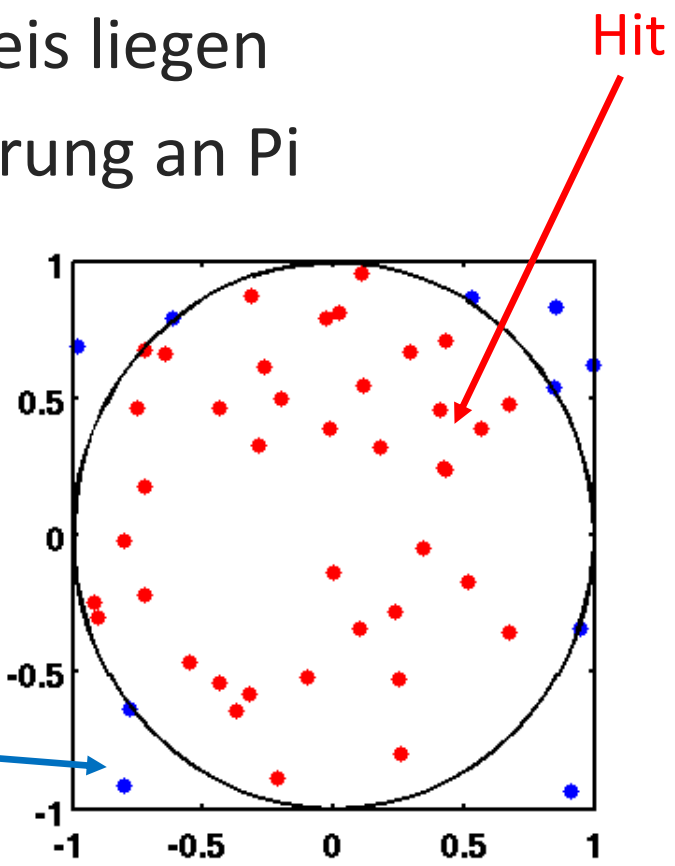
```
MPI_Reduce(&value, &total, 1, MPI_INT, MPI_SUM,
           receiverRank, MPI_COMM_WORLD)
```

# Monte Carlo Pi Simulation

Randomisierte Berechnung von Pi

- Generiere zufällige Punkte aus Fläche
- Schaue, ob sie im Einheitskreis liegen
- Trefferrate gibt eine Annäherung an Pi

$$\frac{\pi}{4} = \frac{\#Hit}{\#Points}$$




©wikipeadia.org

# Sequentieller Algorithmus

```
long count_hits(long trials) {
  long hits = 0, i;
  for (i = 0; i < trials; i++) {
    double x = (double)rand()/RAND_MAX;
    double y = (double)rand()/RAND_MAX;
    if (x * x + y * y <= 1) { hits++; }
  }
  return hits;
}
```

```
long hits = count_hits(TRIALS);
double pi = 4 * ((double)hits / TRIALS);
```

*Wie lässt sich das mit MPI parallelisieren?*

# MPI Parallelisierung

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Jeder Prozess hat anderen willkürlichen Seed

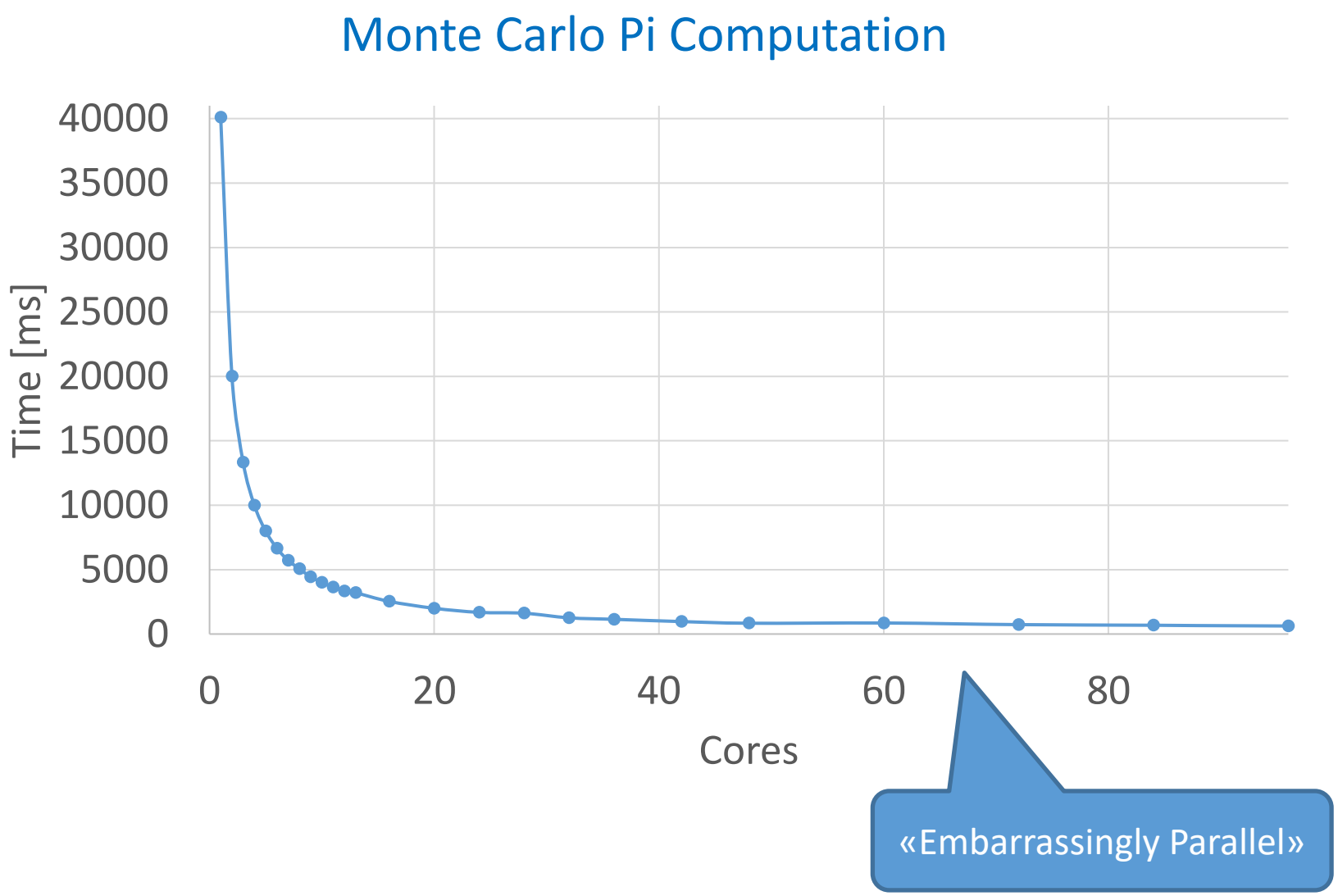
Jeder Prozess rechnet Bruchteil

Prozess 0 erhält Gesamtwert

```
srand(rank * 4711);
long hits = count_hits(TRIALS / size);
long total;
MPI_Reduce(&hits, &total, 1, MPI_LONG, MPI_SUM, 0,
                                       MPI_COMM_WORLD);

if (rank == 0) {
   double pi = 4 * ((double)total / TRIALS);
   …
}
```

# Performance-Analyse



# Rückblick: Lernziele

- Cluster für verteilte Parallelisierung nutzen können
- Parallelprogrammierung mit MPI kennenlernen

# Literatur zum Nachschlagen

MPI Forum

- Standard Spezifikation (für C), aktuell Version 3.1
- https://www.mpi-forum.org/
- MPI 4.0 RC: https://www.mpi-forum.org/docs/drafts/mpi-2020-draft-report.pdf

MPI Tutorial

- Lawrence Livermore National Lab
- https://computing.llnl.gov/tutorials/mpi/

# MPI lokal für Windows

Installation (bei Interesse):

- Microsoft MPI 10.1.2
  - MPI Runtime mit mpiexec
- Microsoft MPI SDK 10.1.2
  - Compilation Header & Libraries, z.B. für Visual Studio

# Selbststudium

# Besondere Kommunikationen

Broadcast (gehabt)

- Einer sendet gleichen Wert an alle

All to all

- Jeder sendet verschiedenen Wert an alle anderen

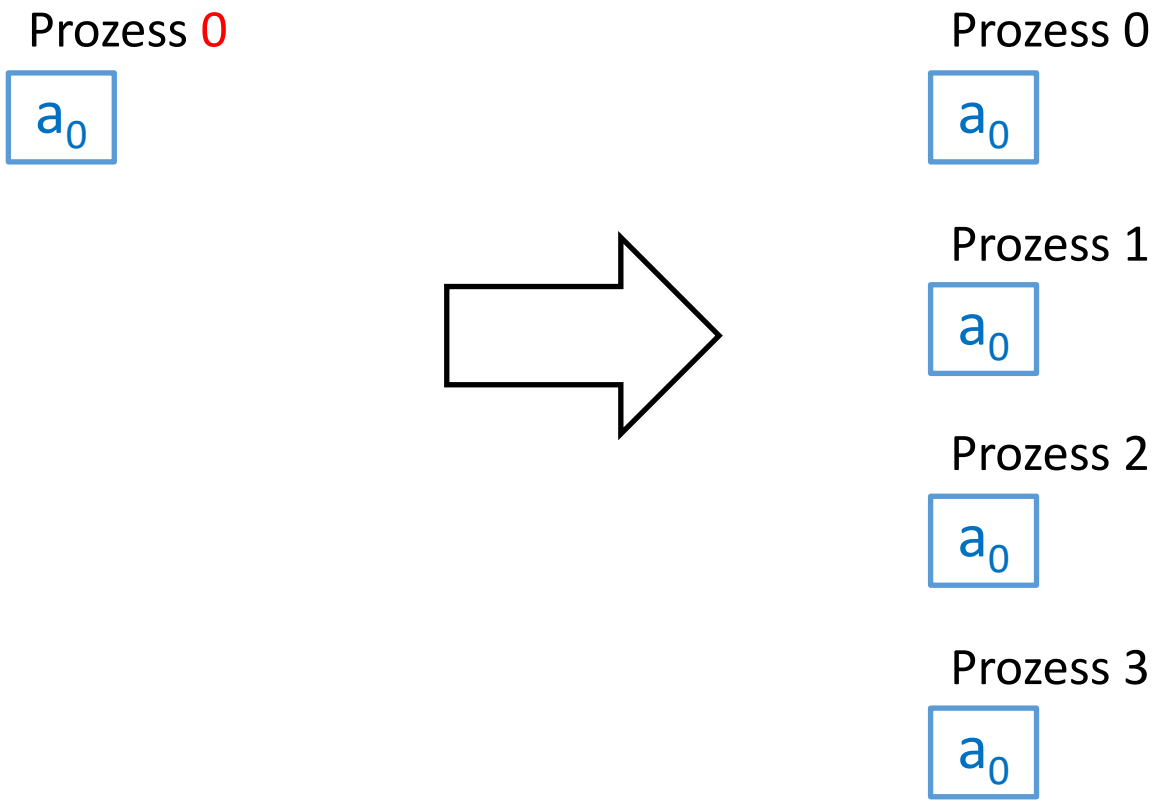
Scatter

- Einer verteilt verschiedene Werte an alle anderen

Gather

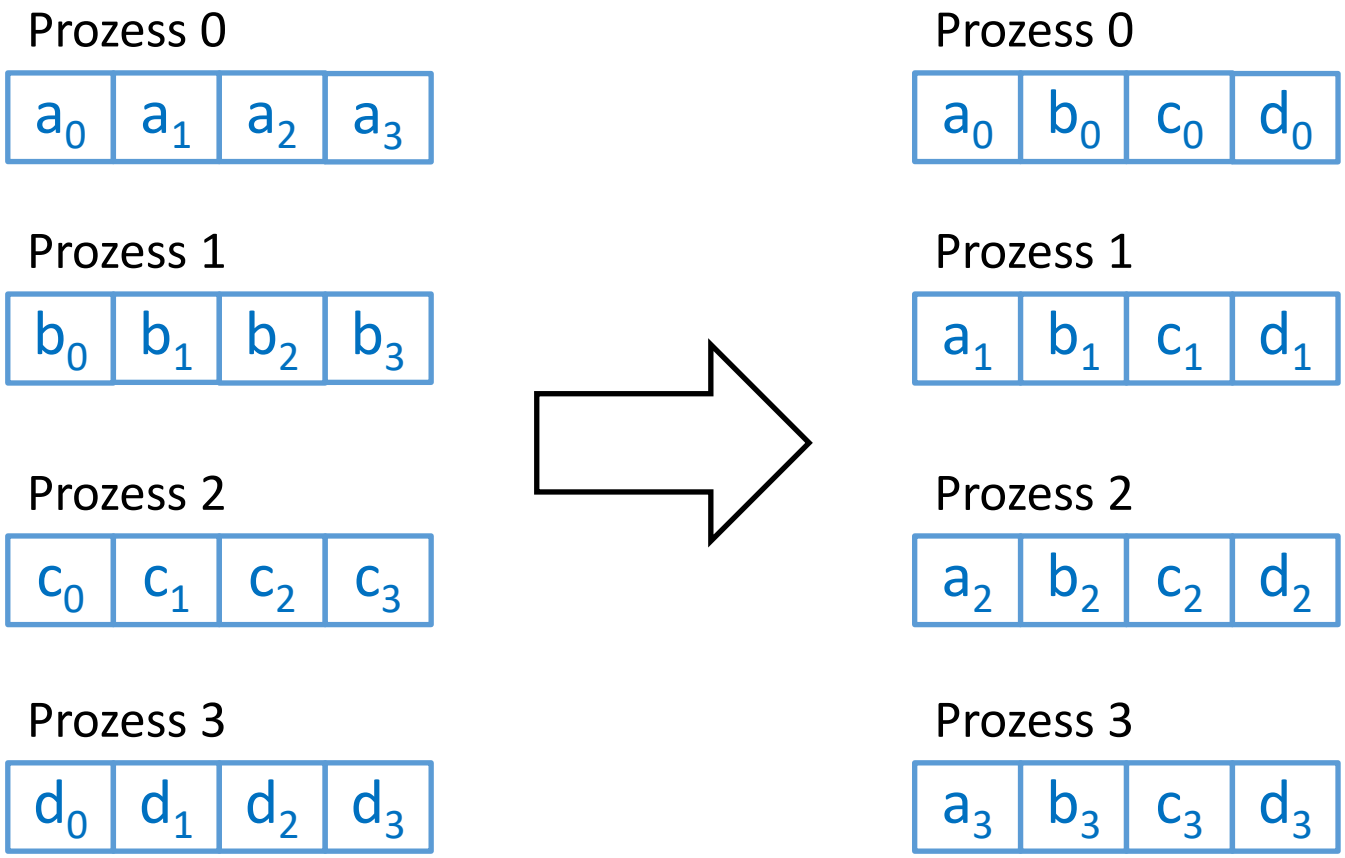
- Alle senden verschiedenen Wert an einen

# Broadcast



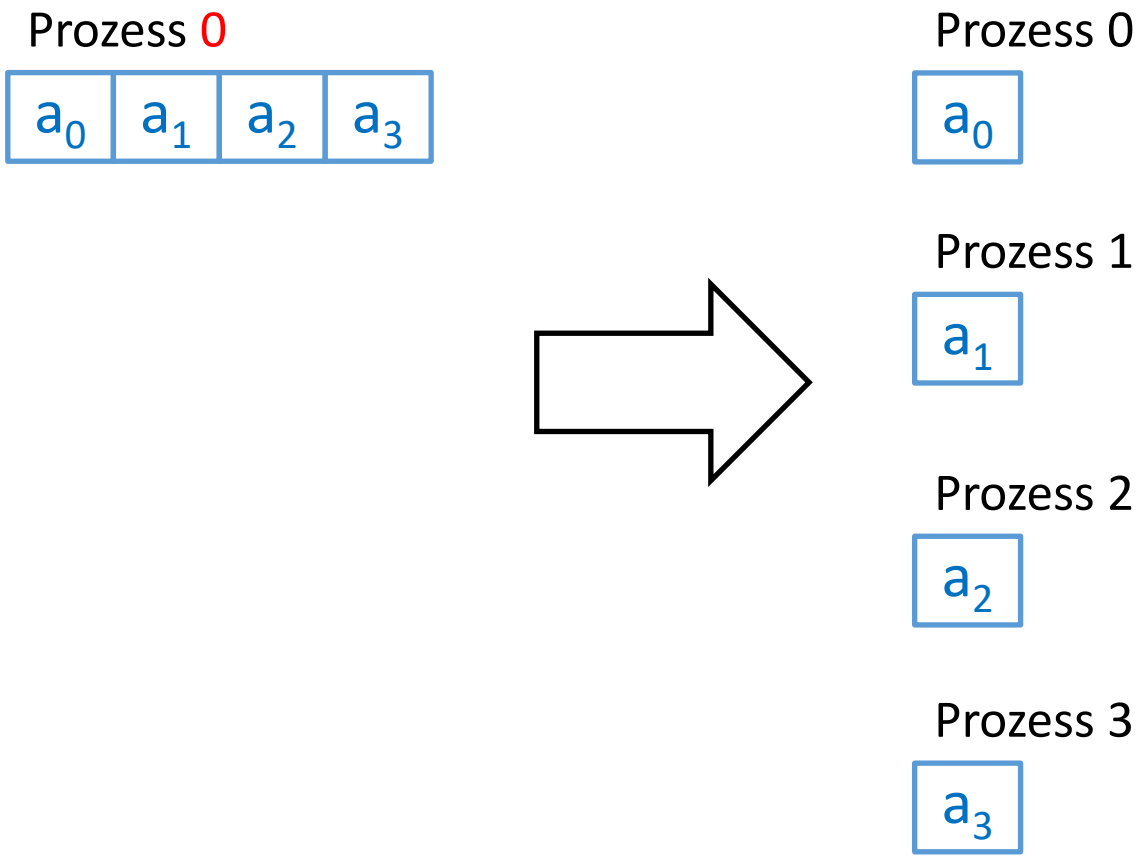

```
MPI_Bcast(&input, 1, MPI_INT, 0, MPI_COMM_WORLD)
```

# All to All



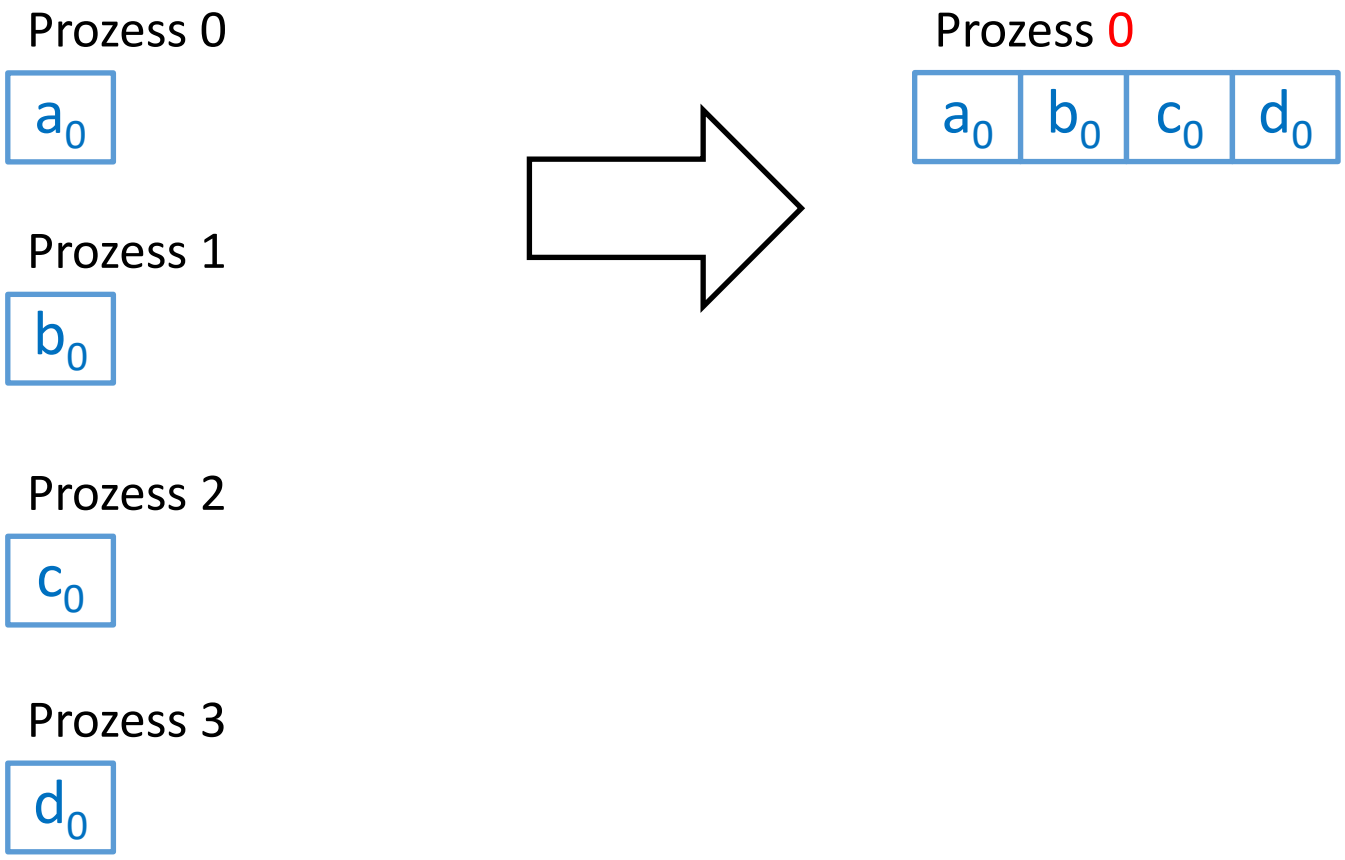

```
MPI_All(&input_array, size, MPI_INT,
&output_array, size, MPI_INT, MPI_COMM_WORLD)
```

# Scatter




```
MPI_Scatter(&input_array, size, MPI_INT,
&output_value, 1, MPI_INT, 0, MPI_COMM_WORLD)
```

# Gather




```
MPI_Gather(&input_value, 1, MPI_INT, &output_array, 1,
           MPI_INT, 0, MPI_COMM_WORLD)
```