4.4. **MPI**
Message Passing Interface

Ferienakademie 2009
Franz Diebold

Agenda

1. Einführung, Motivation
2. Kommunikationsmodell
3. *Punkt-Zu-Punkt*-Kommunikation
4. Globale Kommunikation
5. Vergleich MPI und OpenMP

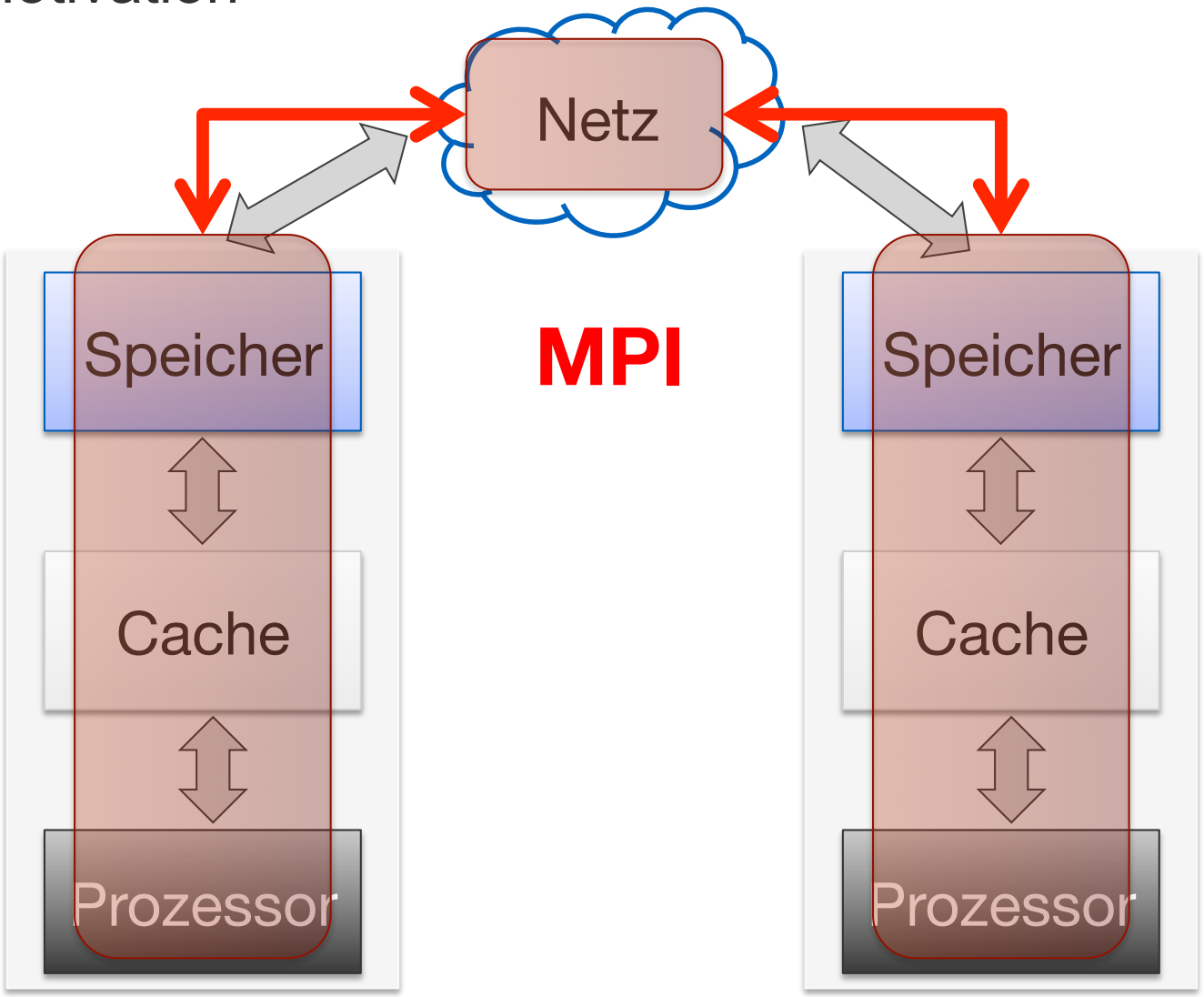
1. Einführung, Motivation

MPI = Message Passing Interface:

- *De-facto-Standard* (Spezifikation) für Nachrichtenaustausch auf verteilten Computersystemen
- Sowohl auf *verteiltem*, als auch auf *gemeinsamem Speicher* nutzbar
- Ziele:
  - Einfache Anwendbarkeit
  - Portabilität
  - Flexibiltät
  - Effizienz
- MPI 1 (1994), MPI 2 (1997)
- Implementierungen: C/C++, Fortran, Ocaml, Java, Python

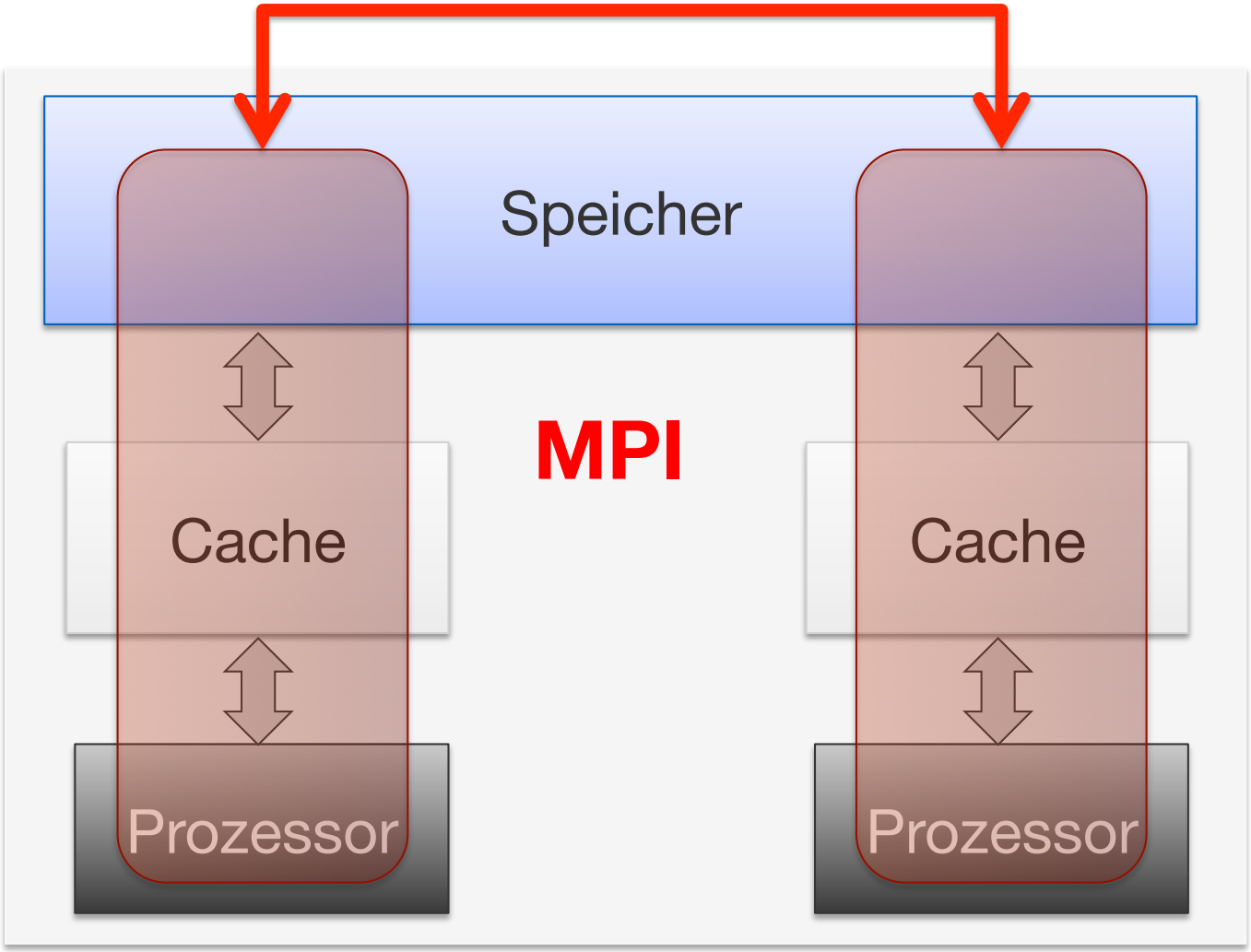
1. Einführung, Motivation

Verteilter Speicher, über ein Netz verbunden



1. Einführung, Motivation

Gemeinsamer Speicher:
z.B. SMP (Symmetrisches Multiprozessor-system)



1. Einführung, Motivation

Was MPI bietet:

- *Schnittstelle*
- Regelung des Nachrichtenverkehrs
- Verschiedene *Kommunikationsverfahren*

Was MPI **nicht** bietet:

- *Automatische Parallelisierung*
- Sicherstellung der Korrektheit der Parallelität
- Automatische *Deadlock-Erkennung & -Vermeidung*
- Automatische *Datenverteilung* über die beteiligten Prozesse

1. Einführung, Motivation

Paradigma von MPI-Programmen: **SPMD**

- Single Programm/Process Multiple Data
- Verzweigung innerhalb des Programmes, um verschiedene Rechner/Knoten unterscheiden zu können

323 Funktionen in Version 2:

- MPI-Funktionen in C:

```
MPI_Name (Parameter [, ...])
```

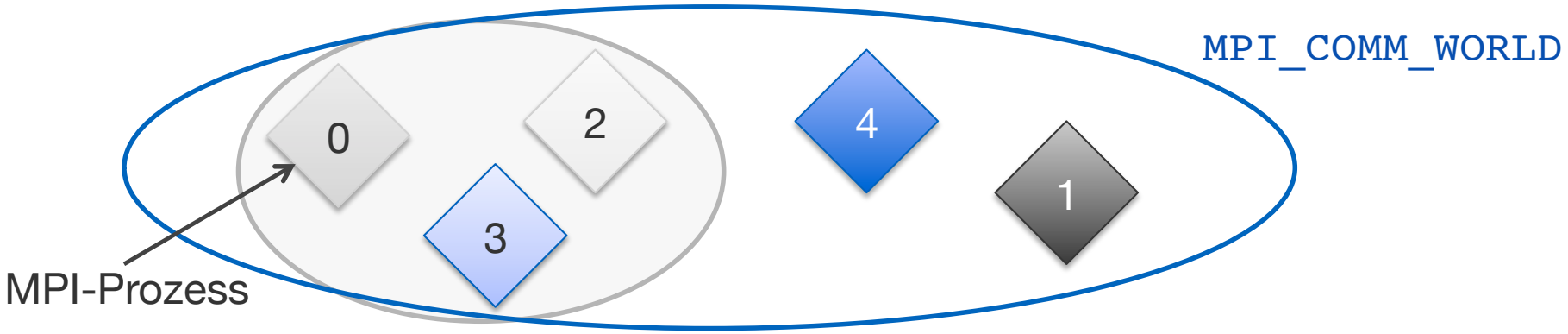
Agenda

1. Einführung, Motivation
2. Kommunikationsmodell
3. *Punkt-Zu-Punkt*-Kommunikation
4. Globale Kommunikation
5. Ausblick

2. Kommunikationsmodell: Begriffe

Kommunikator:
- Menge von MPI-Prozessen
- Definiert *Kontext* für Kommunikation
- Vordefinierte Kommunikatoren:
  - `MPI_COMM_WORLD` → alle gestartete Prozesse
- Ermittlung der Größe eines Kommunikators im Programm:

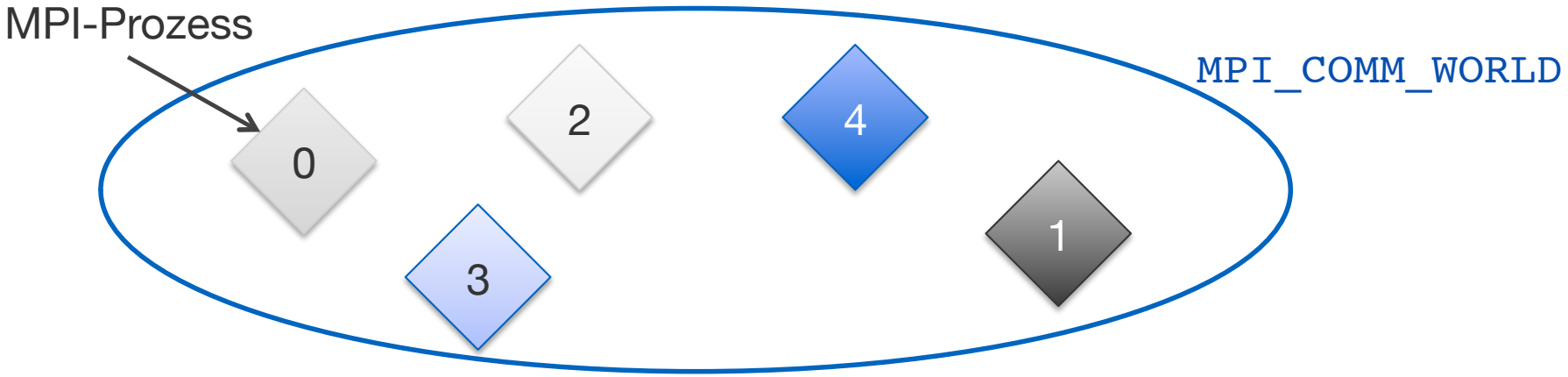
```
MPI_Comm_size (comm, &size)
```



2. Kommunikationsmodell: Begriffe

Rang:
- *Eindeutige Kennung* eines Prozesses
- Bei Initialisierung zugewiesen
- Wird als Quelle und Ziel einer Nachricht angegeben
- Ermittlung im Programm:

```
MPI_Comm_rank (comm, &rank)
```



2. Kommunikationsmodell

Grundlegende MPI-Funktionen (in C):

- Initialisierung der MPI-Umgebung: (z.B. Verbindungsaufbau)

```
MPI_Init (&argc, &argv);
```

  - Muss erste MPI-Funktion im Programm sein
  - Aufrufparameter werden an MPI-Laufzeitumgebung übergeben

- Beenden der MPI-Umgebung: (z.B. Verbindungsabbau)

```
MPI_Finalize ();
```

  - Muss letzte MPI-Funktion im Programm sein

2. Kommunikationsmodell

Grundlegender Aufbau eines MPI-Programms in C:

```
#include <mpi.h>                    // Header-Datei einbinden

// ...

MPI_Init (&argc, &argv);    // MPI initialisieren

// ...
// „MPI-Programm“ hier ...
// ...

MPI_Finalize ();                    // MPI beenden
```

2. Kommunikationsmodell

Einfaches MPI-Programm in C:

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

int main(int argc, char *argv[]) {
   int numtasks, rank;
   MPI_Init (&argc, &argv);
   MPI_Comm_size (MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   printf ("Hallo Welt! Ich bin Prozess %d von %d.\n", rank,
   numtasks);
   MPI_Finalize ();
}
```

2. Kommunikationsmodell

Einfaches MPI-Programm in C: Beispielausgabe

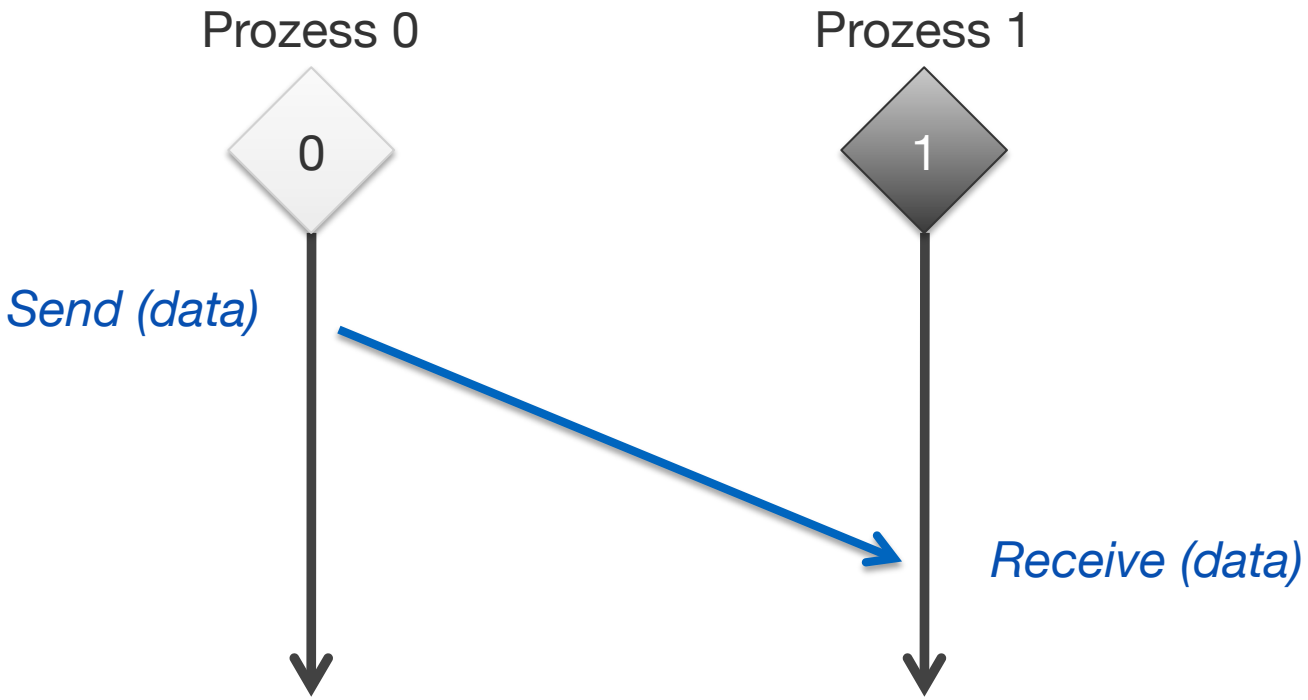
```
franz-diebold:Release Franz$ mpirun –np 5 MPI_HelloWorld
Hallo Welt! Ich bin Prozess 0 von 5.
Hallo Welt! Ich bin Prozess 3 von 5.
Hallo Welt! Ich bin Prozess 1 von 5.
Hallo Welt! Ich bin Prozess 2 von 5.
Hallo Welt! Ich bin Prozess 4 von 5.
```

- Reihenfolge nicht deterministisch!

Agenda

1. Einführung, Motivation
2. Kommunikationsmodell
3. *Punkt-Zu-Punkt*-Kommunikation
4. Globale Kommunikation
5. Vergleich MPI und OpenMP

3. *Punkt-Zu-Punkt*-Kommunikation



- Benötigt Kooperation beider Kommunikationspartner

3. *Punkt-Zu-Punkt*-Kommunikation

MPI-Funktionen zum Senden und Empfangen von Nachrichten:

- **MPI_Send** (void *message_buffer, // zu verschickende Daten
  int count, // „Länge“ der Daten
  MPI_Datatype type, // Datentyp der Daten
  int destination, // Ziel-Prozess der Nachricht
  int message_tag, // Nachrichten-Identifikator
  MPI_Comm communicator); // Kontext

- **MPI_Recv** (void *message_buffer, // zu empfangende Daten
  int count, // „Länge“ der Daten
  MPI_Datatype type, // Datentyp der Daten
  int source, // Herkunfts-Prozess der Nachricht
  int message_tag, // Nachrichten-Identifikator
  MPI_Comm communicator, // Kontext
  MPI_Status* status); // Nachrichten-Status

Datentypen in MPI (`MPI_Datatype`)

MPI-Datentyp	C-Datentyp
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_CHAR`	`signed char`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigend short`
`MPI_UNSIGNED`	`unsigend int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`

3. *Punkt-Zu-Punkt*-Kommunikation

MPI-Funktionen zum Senden und Empfangen von Nachrichten:

- Nachricht von beliebigem Prozess:
  `MPI_ANY_SOURCE`
- Beliebige Nachricht empfangen:
  `MPI_ANY_TAG`

- Prüfen, ob zu empfangende Nachricht verfügbar:

```
MPI_Probe (int source, int message_tag,
           MPI_Comm comm, MPI_Status* status);
```

  - Herkunft: `status->MPI_SOURCE`
  - Identifikator: `status->MPI_TAG`
  - Fehler: `status->MPI_ERROR`
  - „Länge“ der Daten:

```
MPI_Get_count (MPI_Status* status,
               MPI_Datatype type, int* count);
```

3. *Punkt-Zu-Punkt*-Kommunikation

Beispiel: Producer-Consumer-MPI-Programm in C:

```
int numtasks, rank, producer, consumer, data, msgtag;
MPI_Status status;
producer = 0; consumer = 1; msgtag = 134;
[...]
if (rank == producer) {                    // Producer
   data = 1;
   while (data > 0) {
        scanf("%d", &data);
        MPI_Send(&data, 1, MPI_INT, consumer, msgtag, MPI_COMM_WORLD); }
} else if (rank == consumer) {    // Consumer
   data = 1;
   while (data > 0) {
        MPI_Recv(&data, 1, MPI_INT, producer, msgtag, MPI_COMM_WORLD,
   &status); }
}
[...]
```

3. *Punkt-Zu-Punkt*-Kommunikation

Kombinierte Send- & Receive-Funktion:

```
MPI_Sendrecv (void *send_buffer,    // zu verschickende Daten
      int count_send                // „Länge“ der Daten
      MPI_Datatype send_type,       // Datentyp der Daten
      int destination,              // Ziel-Prozess der Nachricht
      int send_tag,                 // Nachrichten-Identifikator
      void *rec_buffer,             // zu empfangende Daten
      int count_rec,                // „Länge“ der Daten
      MPI_Datatype rec_type         // Datentyp der Daten
      int source,                   // Herkunfts-Prozess der Nachricht
      int rec_tag,                  // Nachrichten-Identifikator
      MPI_Comm communicator,        // Kontext
      MPI_Status* status);          // Nachrichten-Status
```

3. *Punkt-Zu-Punkt*-Kommunikation

Unterscheidung blockierend – nicht blockierend:

- **blockierend**:
  - Funktionsaufruf terminiert erst, wenn Puffer wieder verwendet werden bzw. gelesen werden darf
  - **`MPI_Send`**, **`MPI_Recv`**, **`MPI_Ssend`**, **`MPI_Bsend`**, **`MPI_Rsend`**, **`MPI_Sendrecv`**

- **nicht blockierend**:
  - Funktionsaufruf terminiert sofort (kein Warten auf Ende der Kommunikation)
  - **`MPI_Isend`**, **`MPI_Issend`**, **`MPI_Ibsend`**, **`MPI_Irecv`**
  - Puffer darf wieder verwendet werden: **`MPI_Wait`**, **`MPI_Test`**, **`MPI_Waitall`**

3. *Punkt-Zu-Punkt*-Kommunikation

Nicht-blockierendes Senden & Empfangen:

- **MPI_Isend** (void *message_buffer, // zu verschickende Daten
  int count, // „Länge“ der Daten
  MPI_Datatype type, // Datentyp der Daten
  int destination, // Ziel-Prozess der Nachricht
  int message_tag, // Nachrichten-Identifikator
  MPI_Comm communicator // Kontext
  MPI_Request* request); // Request-Objekt (handle)

- **MPI_Irecv** (void *message_buffer, // zu empfangende Daten
  int count, // „Länge“ der Daten
  MPI_Datatype type, // Datentyp der Daten
  int source, // Herkunfts-Prozess der Nachricht
  int message_tag, // Nachrichten-Identifikator
  MPI_Comm communicator, // Kontext
  MPI_Request* request); // Request-Objekt (handle)

3. *Punkt-Zu-Punkt*-Kommunikation

Beispiel: Nicht-blockierende Kommunikation:

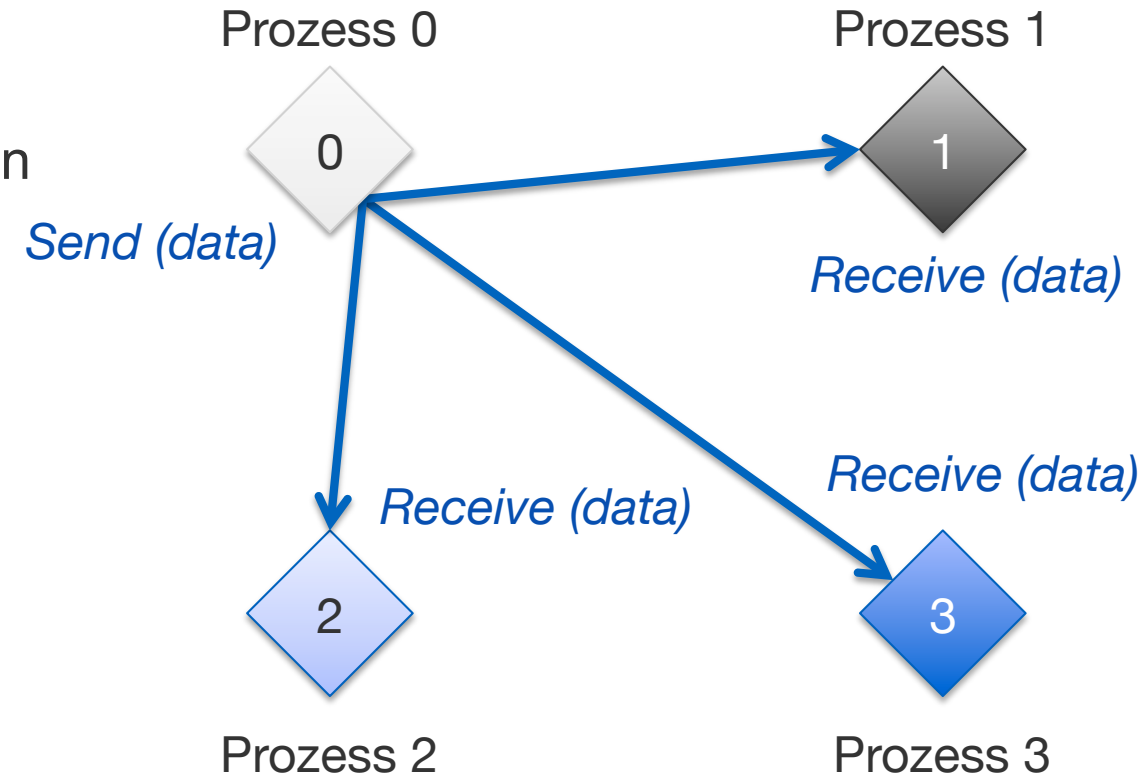
```
[...]
int rank, numtasks, left, right;
int buffer[10], buffer2[10];
MPI_Request request, request2;
MPI_Status status;
[...]
right = (rank+ 1) % numtasks;
left = rank- 1;
if (left < 0)
    left = numtasks – 1;
MPI_Irecv(buffer, 10, MPI_INT, left, 123, MPI_COMM_WORLD, &request);
MPI_Isend(buffer2, 10, MPI_INT, right, 123, MPI_COMM_WORLD, &request2);
MPI_Wait(&request, &status);
MPI_Wait(&request2, &status);
[...]
```

Agenda

1. Einführung, Motivation
2. Kommunikationsmodell
3. *Punkt-Zu-Punkt*-Kommunikation
4. Globale Kommunikation
5. Vergleich MPI und OpenMP

4. Globale Kommunikation

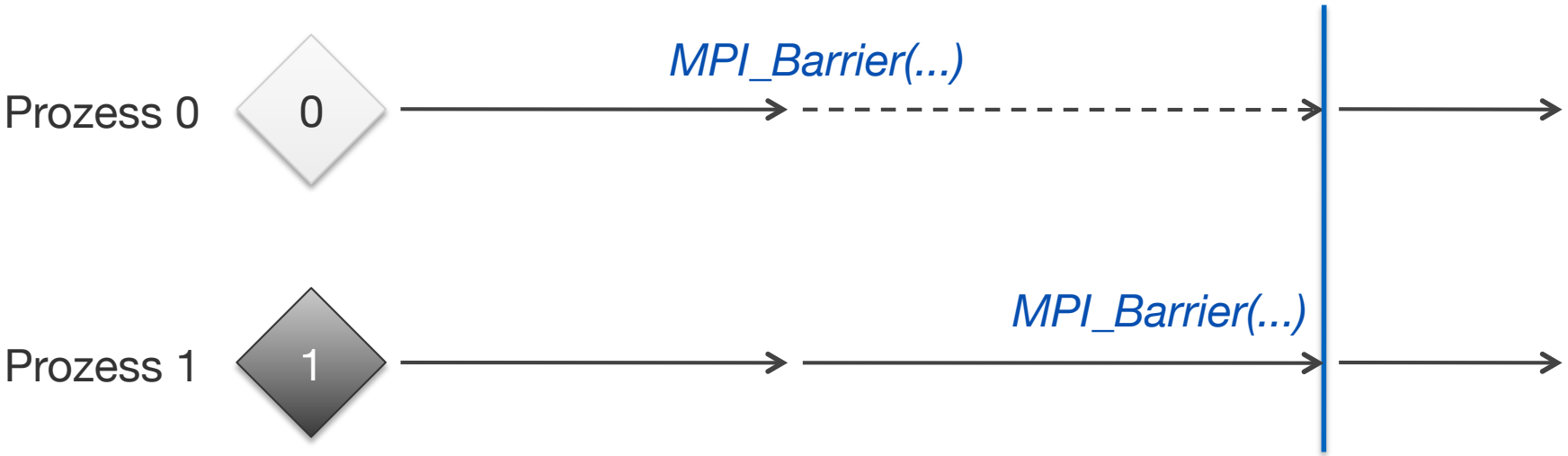
- Betrifft immer *alle* Prozesse des jeweiligen Kommunikators
- Typen:
  - Synchronisation
  - Datenübertragung
  - Gemeinsame Berechnungen
- Keine „Tags“ (Nachrichten-Identifikatoren)



4. Globale Kommunikation

Barriere:

- Blockiert Prozesse, bis alle diese Funktion aufgerufen haben

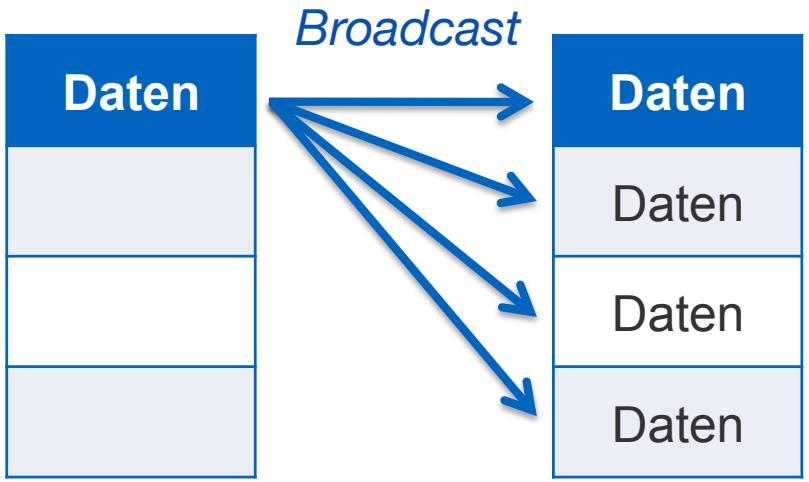


- `MPI_Barrier (MPI_Comm comm);  // Kommunikator (Kontext)`

4. Globale Kommunikation

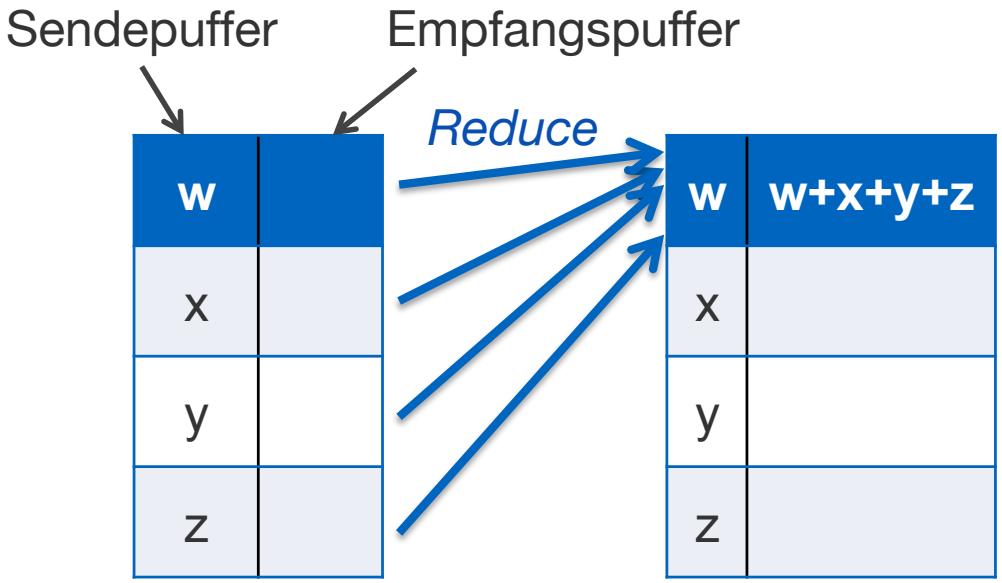
Broadcast:

- Schickt Nachricht von „root“-Prozess zu allen anderen Prozessen
- Bei Sender und Empfänger gleicher Funktionsaufruf
- Barriere (vgl. **MPI_Barrier**)



-
```
MPI_Bcast (void* buffer,          // Sende-/Empfangspuffer
           int count,             // Nachrichtenlänge
           MPI_Datatype type,     // Datentyp
           int root,              // „root“-Prozess
           MPI_Comm comm);        // Kommunikator (Kontext)
```

4. Globale Kommunikation



Reduce:

- Führt globale Operation aus
- „root“-Prozess erhält Ergebnis
- Bei Sender und Empfänger gleicher Funktionsaufruf

```
MPI_Reduce (void* sendbuffer,        // Sendepuffer
            void* recvbuffer,        // Empfangspuffer
            int count,               // Nachrichtenlänge
            MPI_Datatype type,       // Datentyp
            MPI_Op op,               // Reduktionsfunktion
            int root,                // „root“-Prozess
            MPI_Comm comm);          // Kommunikator
```

Globale Operationen in MPI (`MPI_Op`)

MPI-Op	Bedeutung
`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_SUM`	Summe
`MPI_PROD`	Produkt
`MPI_LAND / MPI_LOR / MPI_LXOR`	logisches AND / OR / XOR
`MPI_BAND / MPI_BOR / MPI_BXOR`	bitweises AND / OR / XOR
`MPI_MAXLOC`	Maximum und Index des Prozesses
`MPI_MINLOC`	Minimum und Index des Prozesses

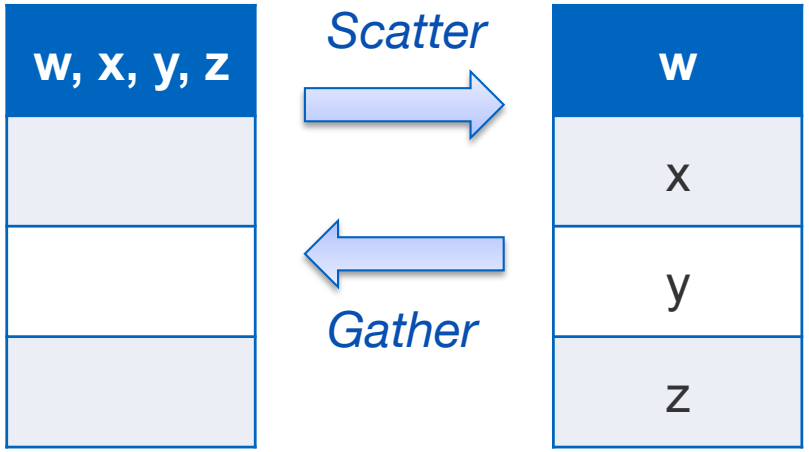
- Nutzung eigener Reduktionsfunktionen möglich mit:

```
MPI_Op_create (MPI_User_function *function, int commute, MPI_Op *op);
```

4. Globale Kommunikation

Gather / Scatter:

- Daten „einsammeln“ bzw. „verteilen“
- Bei Sender und Empfänger gleicher Funktionsaufruf



- **MPI_Gather/ MPI_Scatter**

```
(void* sendbuffer,          // Sendepuffer
 int send_count,            // Nachrichtenlänge
 MPI_Datatype send_type,    // Datentyp (Senden)
 void* recvbuffer,          // Empfangspuffer
 int send_count,            // Nachrichtenlänge
 MPI_Datatype send_type,    // Datentyp (Empfang)
 int root,                  // „root“-Prozess
 MPI_Comm comm);            // Kommunikator
```

Agenda

1. Einführung, Motivation
2. Kommunikationsmodell
3. *Punkt-Zu-Punkt*-Kommunikation
4. Globale Kommunikation
5. Vergleich MPI und OpenMP

5. Vergleich MPI und OpenMP

MPI	OpenMP
✔ mit verteiltem und gemeinsamem Speicher möglich	✔ einfacher zu programmieren und debuggen als MPI
✔ mehr Möglichkeiten als OpenMP	✔ Programm auch seriell ausführbar
✔ Rechner mit verteiltem Speicher billiger als mit gemeinsamem Speicher	✔ Direktiven inkrementell hinzufügbar
	✔ Code einfacher zu verstehen und zu warten
✗ Flaschenhals: Netzwerk	✗ nur mit gemeinsamem Speicher nutzbar
✗ schwer zu debuggen	✗ Compiler mit OpenMP-Unterstützung notwendig
✗ mehr Änderungen von seriell zu parallel	(✗ hauptsächlich für Schleifenparallelisierung)

Quellen

- http://www.tu-chemnitz.de/informatik/RA/projects/mpihelp/
- https://computing.llnl.gov/tutorials/mpi/
- http://www.mpi-forum.org/docs/mpi21-report.pdf
- http://www.rz.rwth-aachen.de/global/show_document.asp?id=aaaaaaaaaabluvl