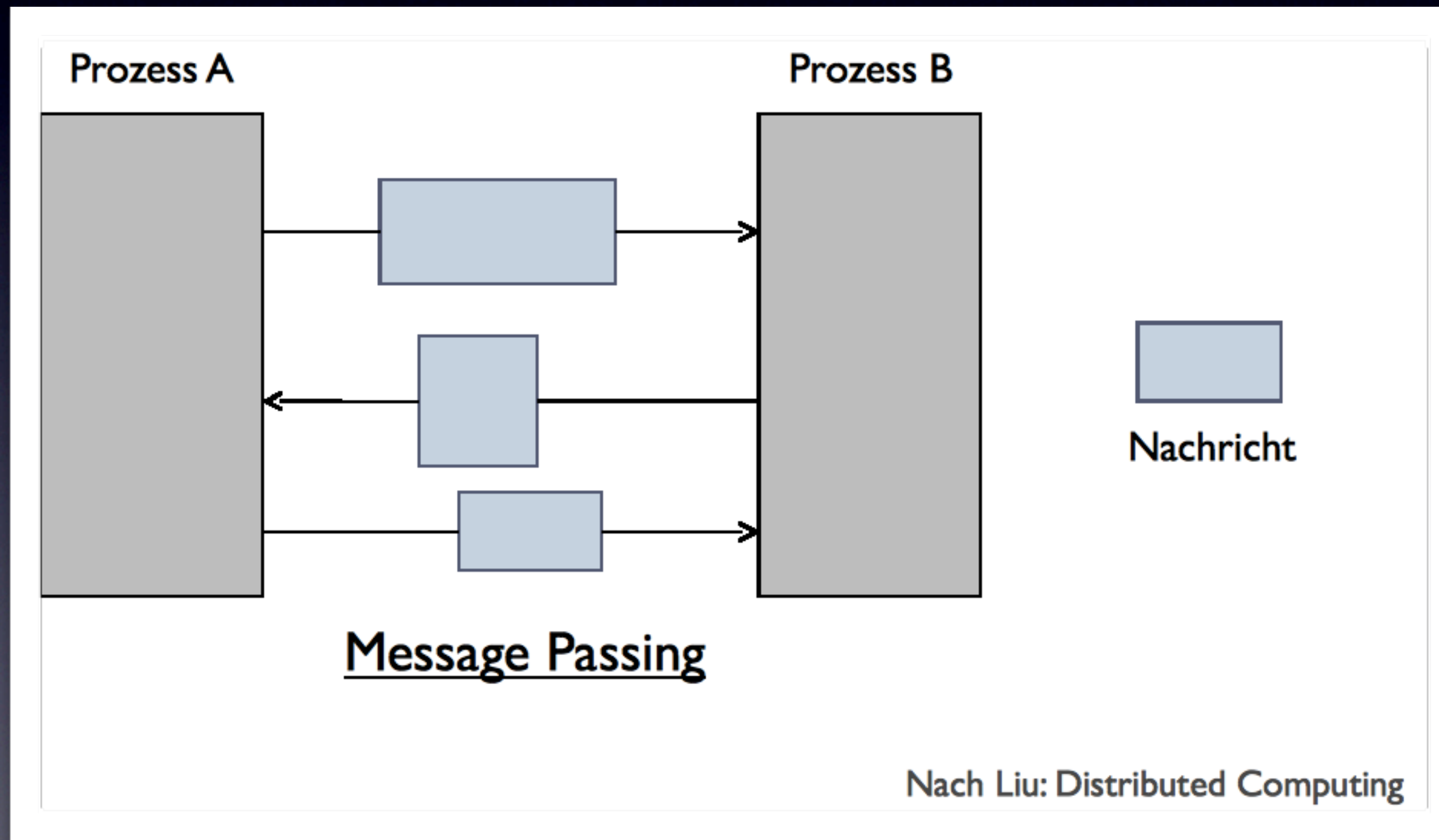
Das Message Passing Paradigma (I)



Das Message Passing Paradigma (2)

- Sehr flexibel, universell, hoch effizient
- Programm kann logisch in beliebig viele Prozesse aufgeteilt werden
- Prozesse können unterschiedlichen Code ausführen
- Prozesse können untereinander kommunizieren

Prinzipien des Message Passing Paradigma (I)

- Logische Sicht eines Message Passing Paradigma Systems
  - Besteht aus p Prozessen
  - Jeder Prozess hat exklusiven Arbeitsspeicher

Prinzipien des Message Passing Paradigma (2)

- Jedes Datenelement muss einem Speicherbereich zugeordnet werden
  - Daten explizit aufteilen
  - Daten in Speicher platzieren

Prinzipien des Message Passing Paradigma (3)

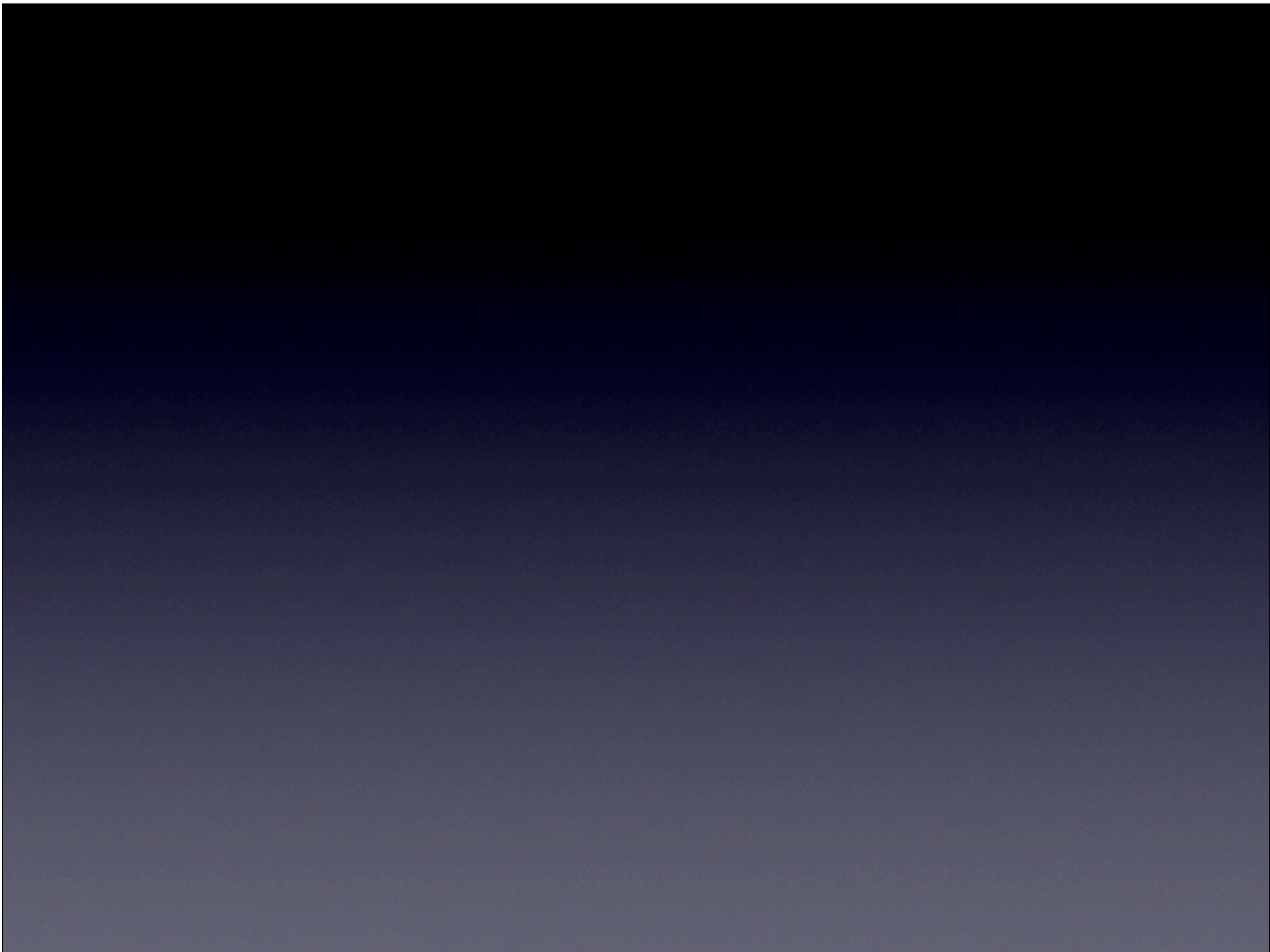
- Bei Interaktionen müssen Prozesse kooperieren
  - Prozess welcher Daten benötigt – Prozess welcher Daten hat

Prinzipien des Message Passing Paradigma (4)

- Nachrichtenversand Synchron/Asynchron möglich
  - Meist asynchron oder „lose“ synchron
- Bei asynchroner Ausführung werden alle Nachrichten asynchron versandt
- Bei lose synchroner Ausführung werden nur Teilaufgaben synchron versandt, zwischen Teilaufgaben asynchrone Kommunikation

Prinzipien des Message Passing Paradigma (5)

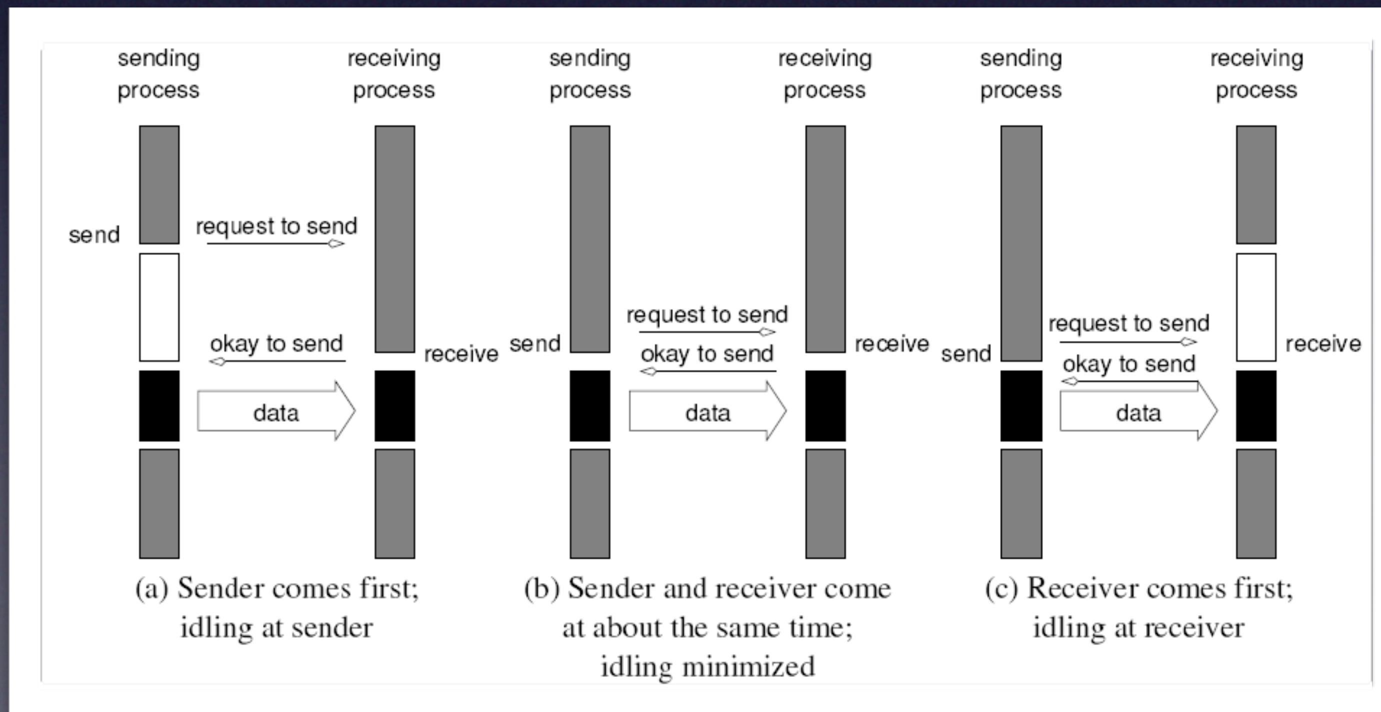
Programme meist nach SPMD (single program multiple data) Modell



Kommunikations-operationen und Arten

Ungepuffert und blockierend

- Keine Rückgabe der sendenden Methode, bis Bestätigung von Empfänger eingetroffen ist
- Probleme: Leerlauf, Deadlocks



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

Prototypen in C

- send(void *sendbuf, int nelems, int dest)
- receive(void *recvbuf, int nelems, int source)

Beispiel

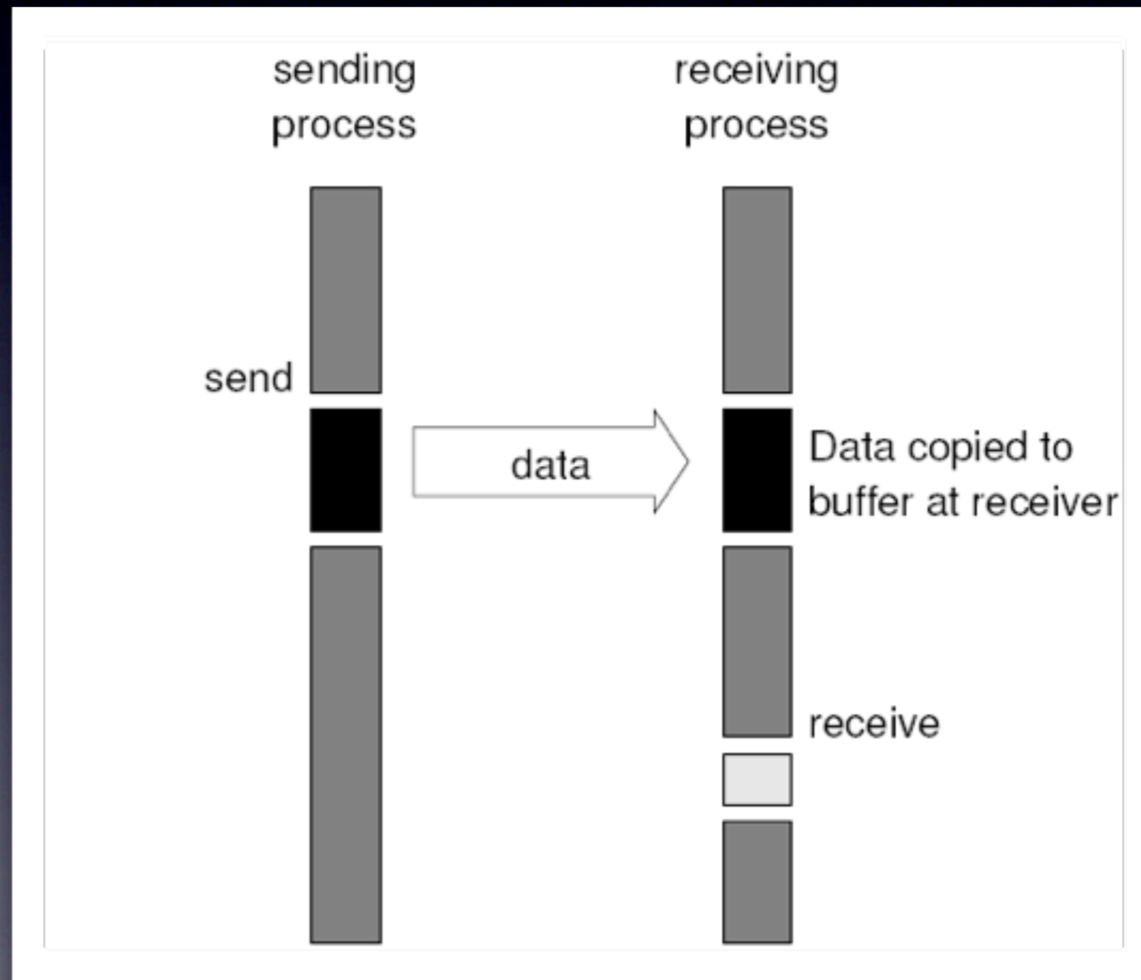
Programm 0:	Programm 1:
int a = 10;	int a = 0;
send(&a, 1, 1);	receive(&a, 1, 0);
a = 0;	printf(„%d“, a);

Was ist die Ausgabe von Programm 1?

Gepuffert und blockierend

- Lösung für Leerläufe und Deadlocks
  - Puffer bei Sender und Empfänger
- Sender kopiert Daten in Puffer und gibt zurück
- Daten werden in Puffer des Empfängers transferiert
- Problem: Leerläufe und Deadlocks werden vermindert, dafür erhöhter Overhead

Gepuffert und blockierend



Prototypen in C

- **b**send(void *sendbuf, int nelems, int dest)
- **b**receive(void *recvbuf, int nelems, int source)

Programm 0:

```
for (i = 0; i < 1000; i++) {
        produce_data(&a);
        bsend(&a, 1, 1);
}
```

Programm 1:

```
for (i = 0; i < 1000; i++) {
        breceive(&a, 1, 0);
        consume_data(&a);
}
```

Performance

- Puffergröße kann großen Einfluss auf Performance haben
- Was passiert, wenn Empfänger langsamer als Sender?
  - Bei vollem Puffer muss Sender warten
- Wie wirkt sich ein größerer Puffer aus?
  - Warten wird gemildert/verhindert, Overhead steigt

Deadlocks

- Sind im Programm unten Deadlocks möglich?

Programm 0:

```
breceive(&a, 1, 1);
bsend(&b, 1, 1);
```

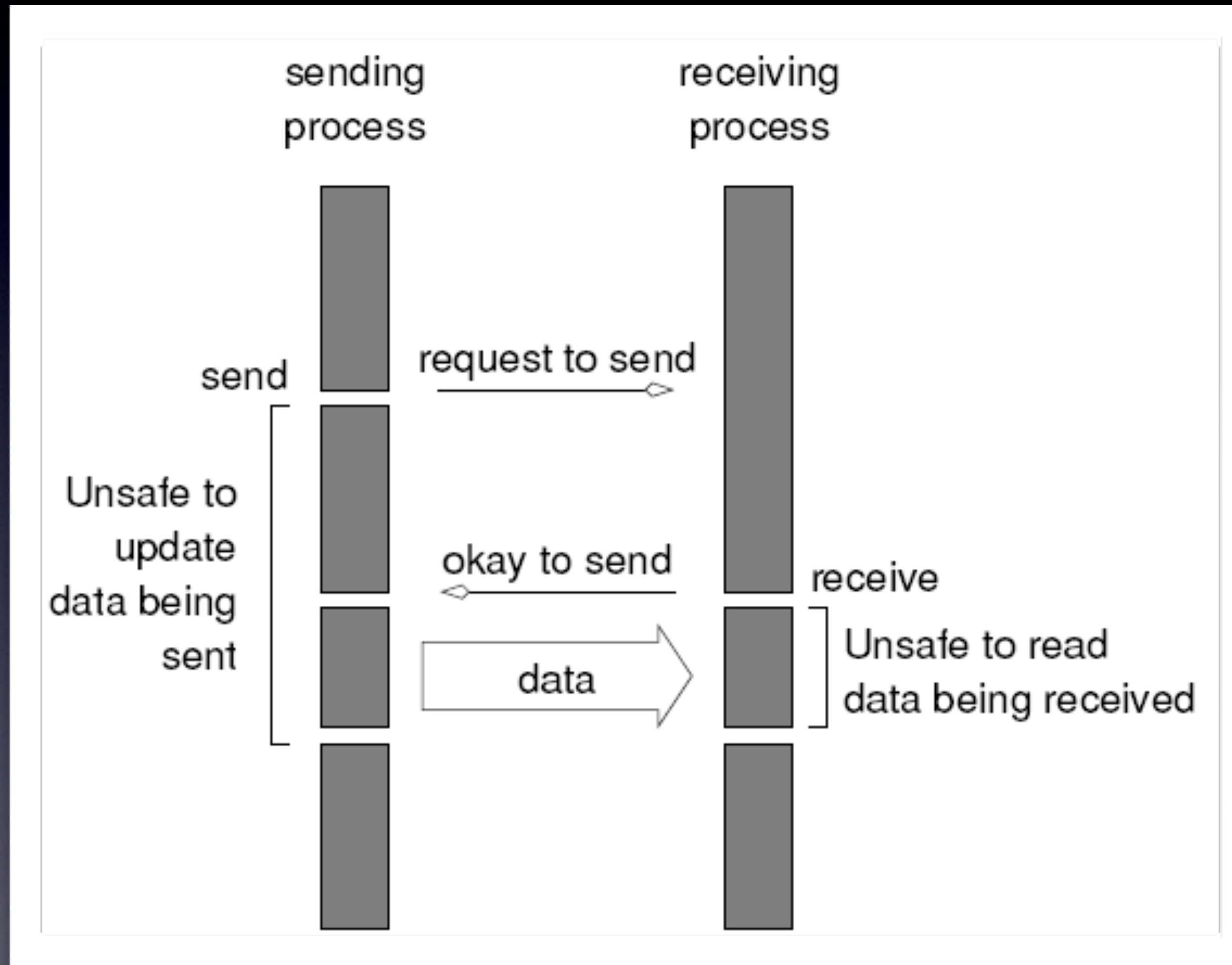
Programm 1:

```
breceive(&a, 1, 0);
bsend(&b, 1, 0);
```

Nichtblockierend

- Sendeoperation wird ausgeführt und Methode gibt sofort zurück (ohne Antwort von Empfänger)
  - Unsicheres Senden
  - Es ist unklar, wann Daten tatsächlich versendet werden

Nichtblockierend



Nichtblockierend

- Programmierer muss sich um Prozessynchronisation kümmern
- Richtig genutzt, können Leerläufe durch Berechnungen genutzt werden
- Message Passing Bibliotheken bieten i.d.R. blockierende und nicht blockierende Kommunikation an

MPI
Message Passing
Interface

Message Passing Interface (I)

- MPI definiert eine Standartbibliothek
  - Sprachenunabhängig
  - Legt Semantik fest
  - Legt Grundoperationen fest

Message Passing Interface (2)

- Eine MPI Applikation besteht meist aus mehreren Prozessen
  - Werden zu Beginn gestartet
  - Kommunizieren untereinander
- Defacto Standard – verwendet bei den meisten Clustern und Supercomputern

MPI Funktionen

`MPI_Init`	Initialisiert Anwendung
`MPI_Finalize`	Terminiert Anwendung
`MPI_Comm_Size`	Ermittelt Anzahl der Prozesse
`MPI_Comm_Rank`	Ermittelt ID des aufrufenden Prozesses
`MPI_Send`	Sendet Nachricht
`MPI_Recv`	Empfängt Nachricht

- Diese Methoden sind mindestens notwendig um eine MPI-Applikation zu erstellen

MPI Starten und Beenden

- Zuerst muss die MPI Umgebung mit `MPI_Init` initialisiert werden
  - `MPI_Init` verarbeitet auch MPI-spezifische Argumente
- Nach Beenden der Berechnung muss `MPI_Finalize` ausgeführt werden
  - Beendet die MPI Umgebung und bereinigt das System

MPI Prototypen in C

- int MPI_Init(char **args, int argc)
- int MPI_Finalize()

MPI Kommunikatoren (I)

- Kommunikator identifiziert Gruppe von Prozessen welche miteinander kommunizieren dürfen
  - Der Kommunikator MPI_COMM_WORLD umfasst alle Prozesse
  - Eigene Kommunikatoren möglich. Sie umfassen dann nur Teil der verfügbaren Prozesse

MPI Kommunikatoren (2)

- Prozess kann mehreren Kommunikatoren gehören
- Kommunikatoren sind Parameter bei allen Nachrichtenübermittelnden MPI-Methoden

MPI Kommunikatoren (3)

- Man kann einen Kommunikator nach der Anzahl seiner Prozesse fragen

- MPI_Comm_size ermittelt die Anzahl von Prozessen des Kommunikators
  - `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Beispiel:
    `int MPI_Comm_size (MPI_COMM_WORLD, &pamount)`

MPI Kommunikatoren (4)

- Jedem Prozess wird eine ID zugewiesen
- MPI_Comm_rank ermittelt die ID des aufrufenden Prozesses
  - `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Beispiel:
    ```
    int MPI_Comm_rank
    (MPI_COMM_WORLD, &prank)
    ```

Senden und Empfangen in MPI

- Befehle für das Senden und Empfangen:
  - `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- MPI hat eigene Datentypen (äquivalent zu C)

MPI: Datentypen (Auszug)

MPI Datentyp	C Datentyp
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED	unsigned int
MPI_DOUBLE	double
MPI_LONG_SOUBLE	long double
MPI_BYTE	

Erstes MPI Programm

```
#include <mpi.h>
main(int argc, char *argv[]) {
  int pamount, myrank, a[10], b[5];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &pamount);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if (myrank == 0) {
       MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
       MPI_Send(b, 5, MPI_INT, 1, 7, MPI_COMM_WORLD);
  }
  else if (myrank == 1) {
       MPI_Recv(b, 5, MPI_INT, 0, 7, MPI_COMM_WORLD);
       MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
  } ...
  MPI_Finalize();
}
```

Anwendungsgebiete

Nutzungsszenarien (I)

- Scatter – Gather (Verteilen und Aufsammeln)
  - Für einfache Problemstellungen
  - Problem wird zerteilt, Teilprobleme an Prozesse gesendet
  - Ergebnisse werden nachher gesammelt und Zusammengefügt/Ausgewertet
  - Bsp. Brute-Force oder Integralberechnung

Nutzungsszenarien (2)

- Komplexe Physikalische Berechnungen
  - Teilprobleme werden an Prozesse gesendet
  - Prozesse kommunizieren untereinander
  - Tauschen Teilergebnisse aus
  - Bsp. Wettersimulationen, Auswertungen der Daten des LHC

Beispiel Anwendungsplattform

- Cluster „Jugene“ im Forschungszentrum Jülich
- Drittschnellster Rechner der Welt (Stand Juni 2009)
- 1002TFlops Maximalleistung