4. Parallele Programmiermodelle

1. Klassifikation paralleler Programmiermodelle
2. Beispiele für parallele Programmiermodelle
3. Message Passing Interface (MPI)
4. OpenMP

4. Parallele Programmiermodelle

1. **Klassifikation paralleler Programmiermodelle**
2. Beispiele für parallele Programmiermodelle
3. Message Passing Interface (MPI)
4. OpenMP

Parallele Programmiermodelle

- Parallele Programmiermodelle vereinfachen die Erstellung paralleler Programme (→ Hochsprachen).
  - In der Literatur wird eine Vielzahl (>100) unterschiedlicher paralleler Programmiermodelle beschrieben.
- Ein paralleles Programmiermodell kann als eine abstrakte (parallele) Maschine aufgefasst werden, welche die Komplexität und Unterschiede konkreter Parallelrechner-architekturen verbirgt.
- Ein paralleles Programmiermodell bildet die Schnittstelle zwischen der Programm- und der Implementierungsebene.
  - Programmebene: Möglichst ausdrucksstarke Primitive
  - Implementierungsebene: Möglichst effiziente Abbildung auf unterschiedliche Parallelrechnerarchitekturen

Anforderungen an parallele Programmiermodelle

- Effizienz
  - Minimierung des Overheads der aus Abstraktion resultiert
- Einfache Programmerstellung
  - Reduzierung der Komplexität
- Einfache Erlernbarkeit
  - Wenige mächtige und orthogonale Primitive
- Plattformunabhängigkeit
  - Abstraktion von Plattform spezifischen Eigenschaften

- Oftmals lassen sich nicht alle Anforderungen gleichzeitig erfüllen.

Klassifikation paralleler Programmiermodelle nach Foster

Klassifikation erfolgt entlang der folgenden Achsen:

- Taskparallelität oder Datenparallelität
  - Primär unterstützte Dekompositionstechnik
- Spracherweiterung oder neuartige Sprache
  - Möglichkeiten der Spracherweiterung:
    - Compiler: z.B. High Performance Fortran
    - Bibliothek: z.B. MPI
- Programmiersprache oder Koordinationssprache
  - Intrusiv: Eigenständige Sprache mit parallelen Konstrukten
  - Nicht-Intrusiv: (Skript-) Sprache zur Steuerung sequentieller Programme.
- Architekturspezifisch oder architekturunabhängig
  - z.B. Occam für Transputer

Klassifikation paralleler Programmiermodelle nach Skillicorn

- Klassifizierung nach dem **Grad der Abstraktion**
- Zur Klassifikation wird bestimmt, welche der folgenden Aspekte der Parallelisierung explizit vom Programmierer betrachtet werden müssen:
  - Dekomposition
  - Task Mapping
  - Kommunikation
  - Synchronisation
- **Hohe Abstraktionsebene:** Viele Aspekte sind für den Programmierer transparent.
- **Niedrige Abstraktionsebene:** Viele Aspekte sind vom Programmierer explizit zu betrachten.

Klassifikation paralleler Programmiermodelle nach Skillicorn

- **Abstraktionsebene 5**
  - Parallelität tritt im Programm nicht explizit auf.
- **Abstraktionsebene 4**
  - Mögliche Parallelität ist im Programm ausgezeichnet.
- **Abstraktionsebene 3**
  - Programm beschreibt explizite Dekomposition in Tasks .
- **Abstraktionsebene 2**
  - Programm beschreibt Zuordnung der Tasks zu den Prozessen.
- **Abstraktionsebene 1**
  - Programm enthält explizite Kommunikationsoperationen.
- **Abstraktionsebene 0**
  - Alle Aspekte der parallelen Ausführung werden im Programm festgelegt.

4. Parallele Programmiermodelle

1. Klassifikation paralleler Programmiermodelle
2. **Beispiele für parallele Programmiermodelle**
3. Message Passing Interface (MPI)
4. OpenMP

Logik basierte Programmierung

- **Beispielprogramm**

```
A :- B,C,D
A :- E,D
```

  - **Prozedurale Leseweise:** Um das Goal A zu beweisen, müssen entweder die Goals B,C und D oder die Goals E und D bewiesen werden.
- **OR Parallelität:**
  - Beide Klauseln für A werden parallel berechnet, bis eine bewiesen wurde oder beide fehlschlagen.
- **AND Parallelität:**
  - Parallele Berechnung der Goals B,C und D bzw. E und D bis alle Goals bewiesen sind oder eines fehlschlägt.

Future Konstrukt in Multilisp

- **`future EXPR`**
  - Liefert ein **Future** für den Wert des Ausdrucks EXPR zurück und
  - startet einen Task der den Wert von EXPR berechnet.
  - Berechnung von EXPR läuft parallel zum aufrufenden Task ab.
- **Auflösung des Futures:** Wenn das Ergebnis der Berechnung von EXPR vorliegt, wird das Future durch den berechneten Wert ersetzt.
- Jeder Task, der den Wert eines Futures benötigt, wird solange blockiert, bis das Future aufgelöst ist.
- Einige Operationen erfordern nicht die Auflösung eines Futures, wie z.B. Argumentübergabe.

High Performance Fortran

- **High Performance Fortran (HPF)** ist eine Programmiersprache, die speziell für die Daten parallele Programmierung entwickelt wurde.
- Zusätzlich zu Fortran90:
  - Programmierkonstrukte zur flexiblen **Auszeichnung von Parallelität** und
  - Programmierkonstrukte zur **Steuerung der Datenverteilung** auf die einzelnen Prozessoren.

Auszeichnung von Parallelität

- **Explizite Parallelität**

```
real A(10,20)
real B(10,20)
logical L(10,20)
A = A + 1.0  ! Parallele Berechnung und Zuweisung
L = A .EQ. B ! Parallele Berechnung und Zuweisung
```

- **Implizite Parallelität** (Schleifen)

```
do i = 1,m
  do j = 1,n
    A(i,j) = B(i,j) * C(i,j)
  enddo
enddo
```

Auszeichnung von Parallelität

- Die **`INDEPENDENT`** Direktive von HPF zeigt an, dass einzelne Schleifeniterationen unabhängig sind und somit parallel bearbeitet werden können.

```
!HPF$ INDEPENDENT
do i = 1,n
  A(Index(i)) = B(i)
enddo
```

Datenverteilung

- Ziel der Datenverteilung: möglichst hohe Lokalität der Daten:
  - Möglichst wenig Kommunikation zwischen den Prozessen
- HPF sieht 3-stufiges Datenverteilungsmodell vor:
  - Auf der ersten Stufe wird spezifiziert, welche Datenelemente auf den selben Prozessor abgebildet werden (**Kollokation**).
    - **`ALIGN`** Direktive
  - Auf der zweiten Stufe wird die Verteilung der Daten auf virtuelle Prozessoren durchgeführt.
    - **`PROCESSORS`** und **`DISTRIBUTE`** Direktiven
  - Auf der dritten Stufe erfolgt die Zuordnung von virtuellen Prozessoren zu den real vorhandenen Prozessoren
    - Abbildungsvorschrift ist implementierungsabhängig

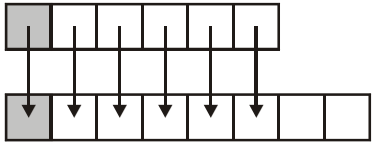
Datenverteilung: ALIGN

- Beispiel:

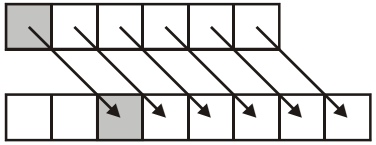
```
real A(10)
real B(10)
!HPF$ ALIGN A(:) WITH B(:)
```

Die jeweils korrespondierenden Elemente der Arrays A und B werden kolloziert, z.B. werden A(0) und B(0) dem selben Prozess zugeordnet.

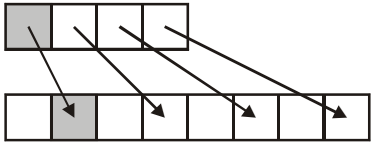
Datenverteilung: ALIGN



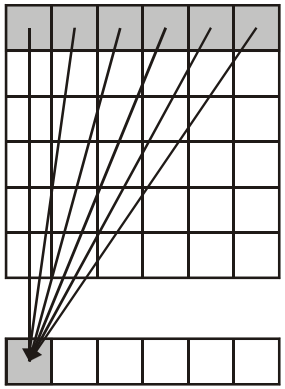
ALIGN A(I) WITH B(I)



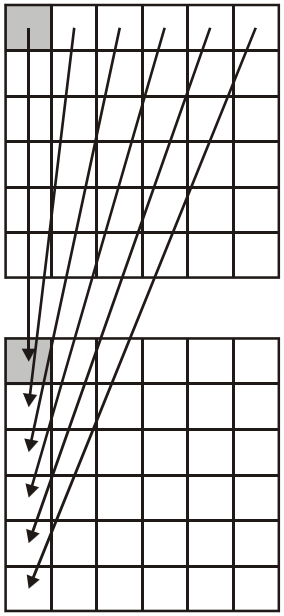
ALIGN A(I) WITH B(I+2)



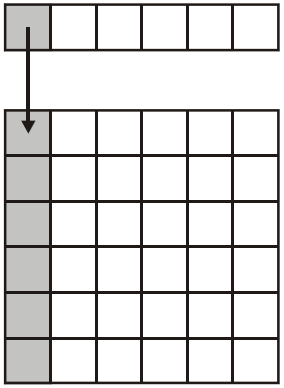
ALIGN A(I) WITH B(2*I)



ALIGN A(:,*) WITH B(:)



ALIGN A(I,J) WITH B(J,I)



ALIGN A(:) WITH B(*,:)

Datenverteilung

- Zunächst wird mit der **`PROCESSORS`** Direktive ein Array von virtuellen Prozessoren (Prozessen) spezifiziert:

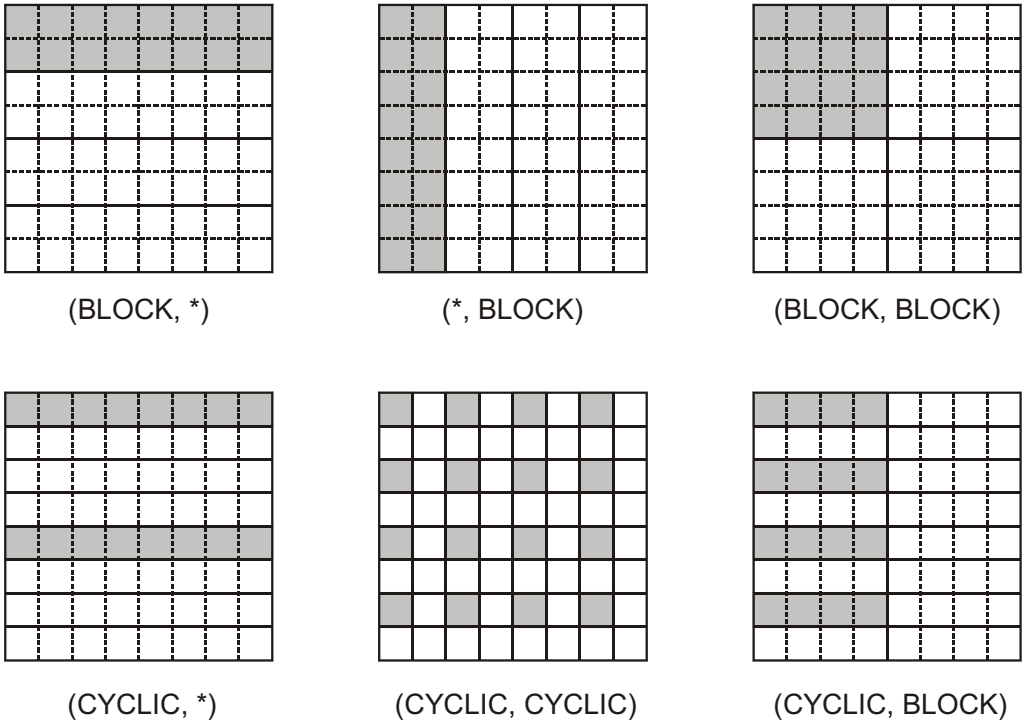
```
!HPF$ PROCESSORS P(64)
!HPF$ PROCESSORS Q(8,8)
```

- Mittels der **`DISTRIBUTE`** Direktive wird dann die Partitionierung und Verteilung der Datenstruktur vorgenommen:

```
real X(1024,1024)
!HPF$ DISTRIBUTE X(BLOCK,*) ONTO Q
```

  - **`BLOCK:`** Blockverteilung
  - **`CYCLIC:`** zyklische Verteilung
  - **`*:`** keine Verteilung in dieser Dimension

Datenverteilung: DISTRIBUTE



- Die DISTRIBUTE Direktive beeinflusst auch alle kollozierten Datenelemente.
  - darf nicht auf kollozierte Daten selbst angewendet werden

Linda

- Das Programmiermodell von Linda besteht aus einem **Tupelraum**
  - Speicher
  - Kommunikationskanal
- Daten können in Form von **Tupeln** in den Tupelraum eingefügt und Tupel können aus dem Tupelraum gelesen, bzw. entfernt werden.
- Neben den passiven Datentupeln sind in Linda aktive **Prozesstupel** vorgesehen.
  - Alle Prozesstupel werden parallel ausgeführt und kommunizieren über den Tupelraum durch Einfügen und Lesen von Datentupeln.
  - Wenn die Ausführung eines Prozesstupels beendet ist, wird dieses zu einem gewöhnlichen Datentupel.

Tupel in Linda

- Ein Tupel besteht aus einer Sequenz von typbehafteten Werten.
  - Beispiel: (‘‘sqrt’’, 16, 4).
- Ein **Muster** ist ein spezielles Tupel, bei dem ein oder mehrere Einträge so genannte formale Parameter sind.
  - Beispiel: (‘‘sqrt’’, 16, int ?X).
- Ein Muster passt zu einem Tupel,
  - wenn Tupel und Muster in den Werten punktweise übereinstimmen bzw.
  - der Typ aller formalen Parameter des Musters mit dem Typ der entsprechenden aktuellen Einträge des Tupels übereinstimmt.
  - In diesem Fall werden die entsprechenden Einträge des Tupels den formalen Parameter des Musters zugewiesen.

Linda Operationen

- out(t)
  - Einfügen des Tupels t in den Tupelraum
  - Der aufrufende Prozess kehrt unmittelbar nach dem Aufruf zurück und führt seine Berechnung fort.
- in(m)
  - Entfernen eines Tupels das zum Muster m passt.
  - Falls mehrere passende Tupel im Tupelraum vorhanden sind, wird zufällig eine Tupel ausgewählt.
  - Wenn kein passendes Tupel im Tupelraum verfügbar ist, blockiert der Prozess.
- rd(m)
  - wie in(m), nur dass das passende Tupel im Tupelraum verbleibt.

Linda Operationen

- eval(t)
  - Diese Operation unterscheidet sich von out(t) dadurch, dass das Tupel t erst nach dem Einfügen in den Tupelraum ausgewertet wird.
  - Die Auswertung erfolgt in einem eigenen Prozess.
  - Das Tupel t wird somit bis zum Ende des Auswertungs-vorgangs zum Prozesstupel.
  - Beispiel: eval(‘‘sqrt’’, 16, sqrt(16))
- Viele wichtige Datenstrukturen, Kommunikations- und Synchronisationskonstrukte lassen sich in dieser Tupel-Sprache codieren.

Beispiel: Matrix Multiplikation in Linda

```
out(„A“,0,< ... erste Zeile ... >)
out(„A“,1,< ... zweite Zeile ... >)
...
out(„B“,0,< ... erste Spalte ... >)
out(„B“,1,< ... zweite Spalte ... >)
...
out(„next“,0)

for (;;) {
  in(„next“, int ?index)
  if (index < dim*dim)
    out(„next“,index+1)
  i = (index-1) /(dim+1)
  j = (index-1) %(dim+1)
  rd(„A“,i,vector ?row)
  rd(„B“,j,vector ?column)
  eval(„res“,i,j,DotProduct(row,column))
}
```

4. Parallele Programmiermodelle

1. Klassifikation paralleler Programmiermodelle
2. Beispiele für parallele Programmiermodelle
3. **Message Passing Interface (MPI)**
4. OpenMP

Eigenschaften des Message Passing Programmiermodells

- Abstrakte Maschine ist charakterisiert durch:
  - Menge von Prozessen
  - **Getrennte Adressräume:** jeder Prozess hat eigenen Adressraum
  - Kommunikation mittels Nachrichtenaustausch (send/receive)
- Konsequenzen:
  - Daten müssen explizit den Prozessen zugeordnet werden.
  - Jede Art von Interaktion erfordert explizite Kooperation der beteiligten Prozesse (Besitzer und Verwender der Daten).

- **Explizite Parallelisierung**
  - Das Message Passing Programmiermodell befindet sich auf niedriger Abstraktionsebene (Abstraktionsebene 0).

Struktur von Message Passing Programmen

- Meistens werden Message Passing Programme im SPMD Modell konzipiert.
- **Asynchronous Style**
  - Tasks werden vollständig unabhängig voneinander ausgeführt.
- **Loosely Synchronous Style**
  - Tasks synchronisieren sich an bestimmten Punkten der Ausführung.

Grundlegende Message Passing Primitive: Send und Receive

- **`send(void* sendbuf, int nelems, int dest)`**
  - **`sendbuf`**: Zeiger auf Puffer mit den zu sendenden Daten
  - **`nelem`**: Anzahl der zu sendenden Datenworte
  - **`dest`**: ID des Empfänger-Prozess

- **`receive(void* recvbuf, int nelems, int source)`**
  - **`recvbuf`**: Zeiger auf Puffer für die zu empfangenden Daten
  - **`nelem`**: Anzahl der zu empfangenden Datenworte
  - **`source`**: ID des Sender-Prozess

Blockierende vs. nicht-blockierende Message Passing Operationen

Prozess P0

```
a = 100;
send(&a,1,1);
a = 0;
```

Prozess P1

```
receive(&a,1,0);
printf("%d\n",a);
```

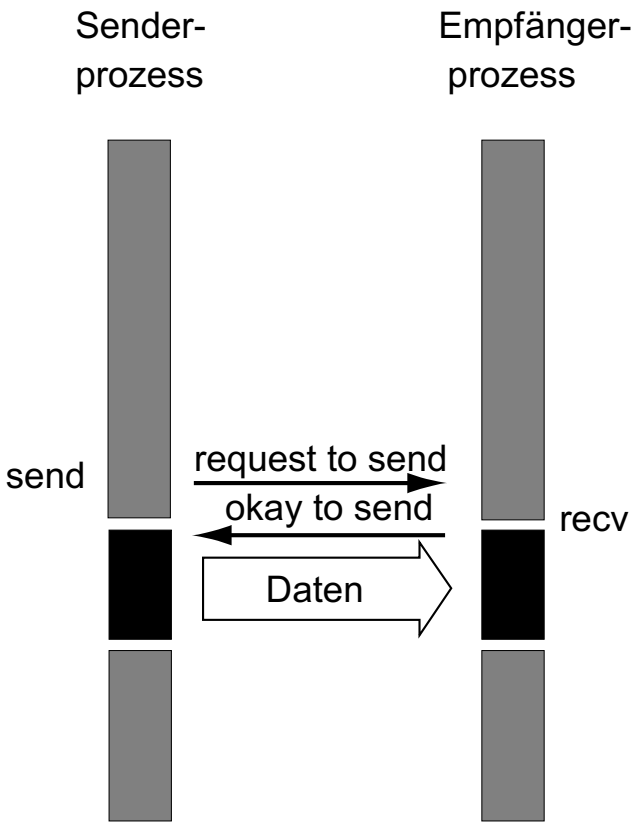
- Oft wird (mittels spezieller Hardware) die Nachrichtenübertragung parallel zum Programmablauf durchgeführt.
  - Übertragung von 0 statt 100 möglich!
- **Blockierende Message Passing Operationen**
  - Send Aufruf kehrt erst zurück, wenn der Sendepuffer modifiziert werden kann, ohne die Semantik zu verändern.
- **Nicht-blockierende Message Passing Operationen**
  - Send Aufruf kehrt sofort zurück; Korrektheit muss vom Programmierer sichergestellt werden.

Blockierende nicht-gepufferte Send/Receive Operationen

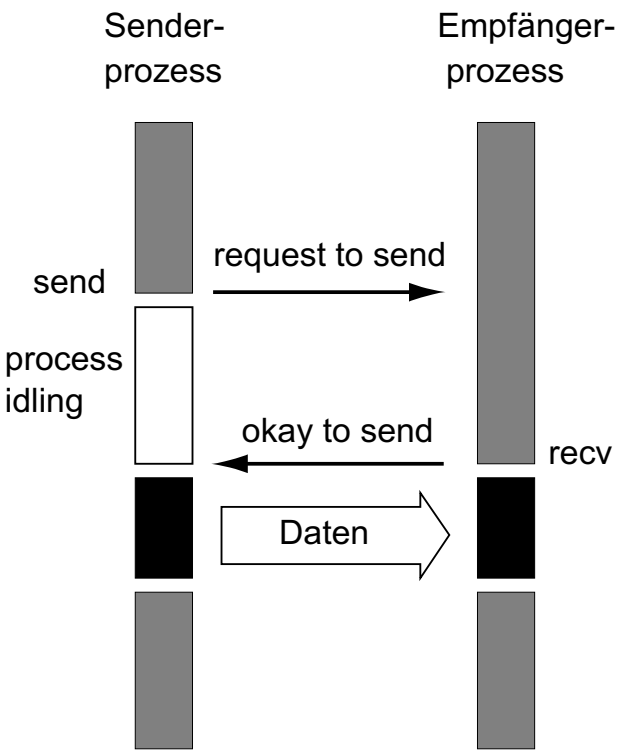
- Send-Aufruf kehrt erst zurück, wenn
  - der entsprechende Receive-Aufruf stattgefunden hat und
  - die Nachricht vollständig übertragen wurde.
- Implementierung mittels **Handshake-Protokoll**.
- Nachteile:
  - Falls keine enge Synchronisation möglich, wird der Sender- oder Empfängerprozess blockiert (→ **Process Idling**).
  - Deadlocks möglich:

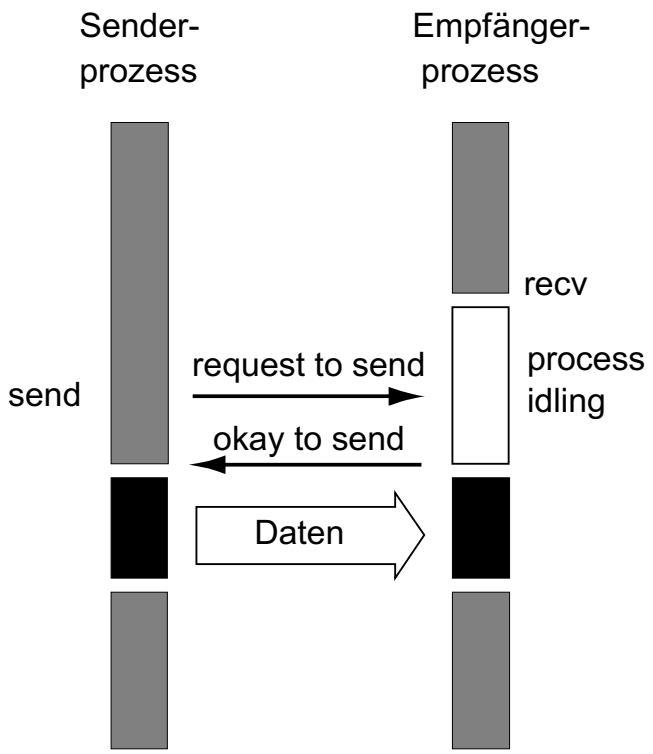
Prozess P0	**Prozess P1**
`send(&a,1,1);`	`send(&b,1,0);`
`receive(&b,1,1);`	`receive(&a,1,0);`

Blockierende nicht-gepufferte Send/Receive Operationen



Blockierende nicht-gepufferte Send/Receive Operationen





Blockierende gepufferte Send/Receive Operationen

- Sender und/oder Empfängerprozess verwenden interne Pufferspeicher für die Kommunikation.
  - Puffervariablen im Programm werden vom eigentlichen Nachrichtenaustausch entkoppelt.
- Nachteile:
  - Overhead für Puffermanagement (Kopieren der Daten, ...).
  - Bei Pufferüberlauf muss Senderprozess blockiert werden.
  - Deadlocks möglich (receive Operation kehrt erst zurück, wenn Daten im lokalen Puffer verfügbar sind):

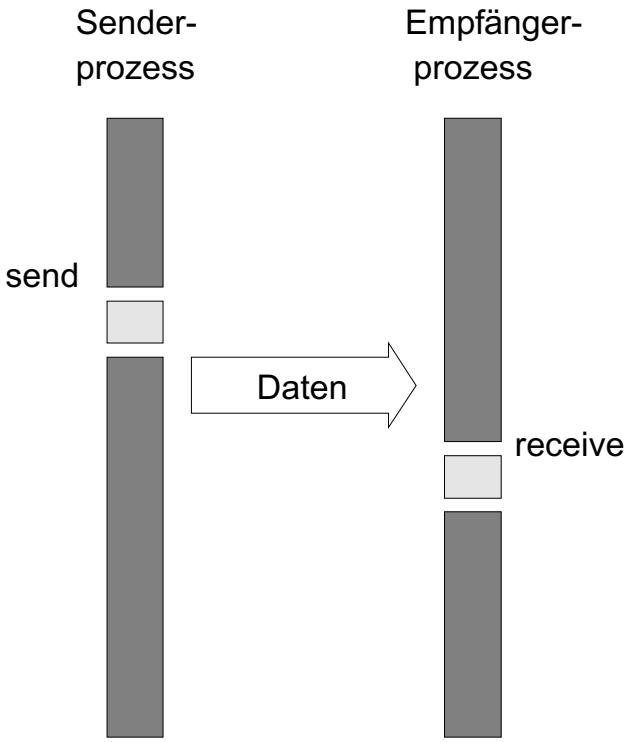
Prozess P0

```
receive(&a,1,1);
send(&b,1,1);
```

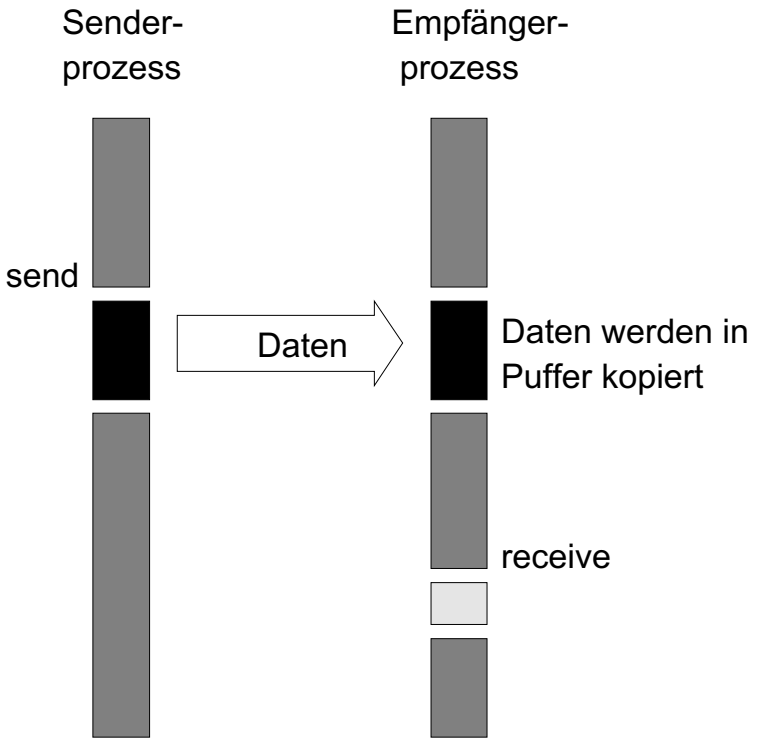
Prozess P1

```
receive(&b,1,0);
send(&a,1,0);
```

Blockierende gepufferte Send/Receive Operationen



Mit Kommunikationshardware
Puffer bei Sender- und
Empfängerprozess

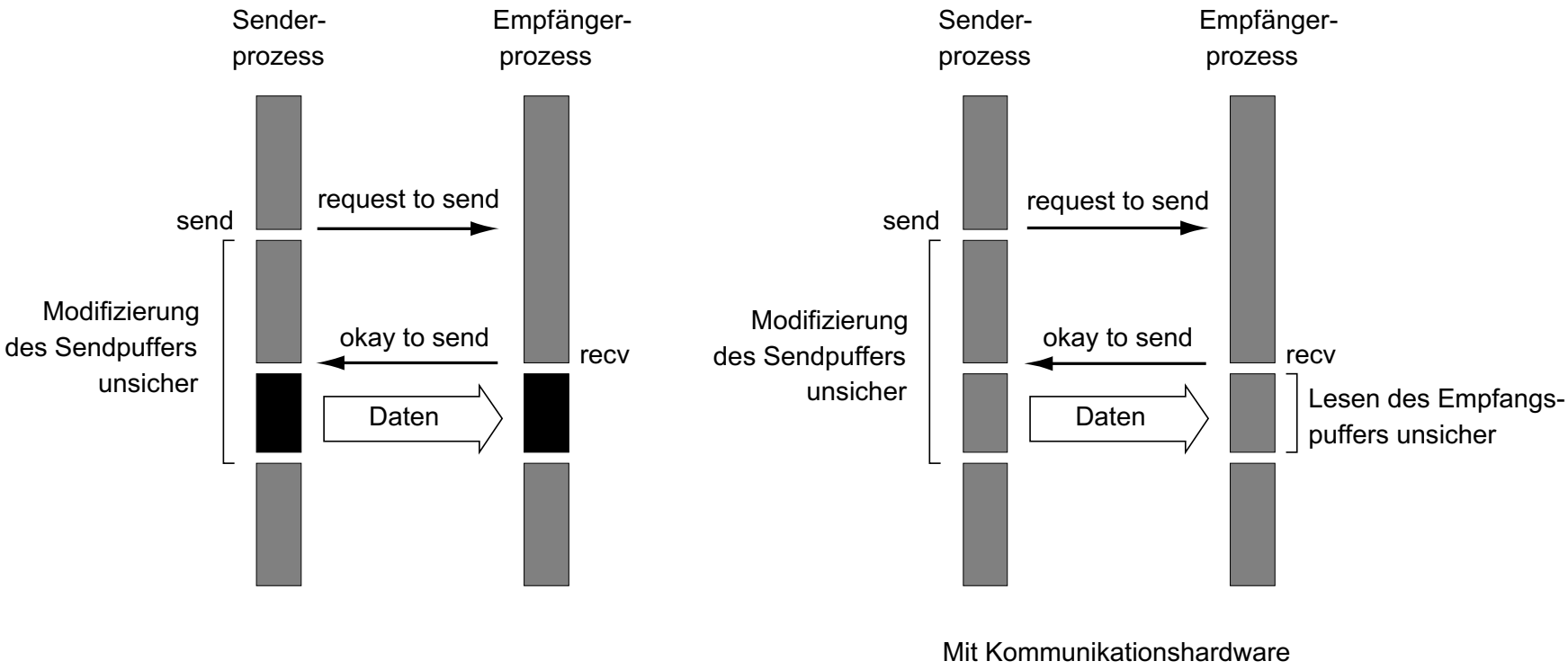


Puffer bei Empfängerprozess

Nicht-blockierende Send/Receive Operationen

- Nicht-blockierende Send/Receive Aufrufe kehren zurück, bevor Puffervariablen sicher geändert werden können.
- Kein Overhead in Form von Process Idling oder Puffermanagement wie bei den blockierenden Operationen
- Programmierer muss sicherstellen, dass Puffervariablen nicht vor Beendigung der Kommunikationsoperation verändert werden.
- **Check-Status** Primitiv gibt Auskunft, ob Puffervariablen sicher überschrieben werden können.

Nicht-blockierende Send/Receive Operationen



Mit Kommunikationshardware

Vergleich

	blockierend	Nicht-blockierend
gepuffert	Send-Aufruf kehrt zurück, nachdem die Daten in den Kommunikationspuffer kopiert wurden.	
nicht-gepuffert	Send-Aufruf kehrt zurück, wenn ein entsprechender Receive-Aufruf ausgeführt wurde.	Send(Receive)-Aufruf kehrt sofort zurück. Sichere Modifikation der Daten nicht sofort möglich.
	Korrektheit wird implizit sichergestellt	Programmierer muss Korrekt-heit explizit sicherstellen

Message Passing Interface (MPI)

- Ausgangssituation: „Message passing is the assembly language of parallel programming“
  - niedere Abstraktionsebene
  - Hardwarehersteller liefern eigene Implementierungen, die sich in Syntax und Semantik z.T. wesentlich unterscheiden.
- Message Passing Interface (MPI) Standard (1994)
  - Erstellt durch Message Passing Interface Forum
    - Konsortium mit Vertretern aus Industrie, Wirtschaft und Regierungsbehörden
  - Definiert Syntax und Semantik von ca. 120 Bibliotheks-funktionen.
- MPI Programme werden (üblicherweise) in C oder Fortran erstellt.

Grundlegende MPI Aufrufe und Konzepte

- 6 elementare MPI Bibliotheksfunktionen:
  - **`MPI_Init`**
  - **`MPI_Finalize`**
  - **`MPI_Comm_size`**
  - **`MPI_Comm_rank`**
  - **`MPI_Send`**
  - **`MPI_Recv`**
- Rückgabewerte (vom Typ **`int`**):
  - **`MPI_SUCCESS`** bei korrekter Ausführung, sonst
  - implementierungsabhängiger Fehlerwert.
- Header File: **`#include <mpi.h>`**

Initialisierung und Beendigung von MPI Programmen

- `int MPI_Init(int* argc, char*** argv)`
  - Argumente sind die Kommandozeilenargumente des Programms
  - Initialisierung der MPI Umgebung
  - Muss vor allen anderen MPI Aufrufen ausgeführt werden
- `int MPI_Finalize()`
  - Ausführung von clean-up Routinen
  - Es dürfen keine MPI Aufrufe nachfolgen
- Beide Bibliotheksfunktionen müssen von allen Prozessen der Berechnung aufgerufen werden.

MPI Kommunikationsdomänen

- Eine MPI **Kommunikationsdomäne** definiert eine (Teil-) Menge von kommunizierenden Prozessen.
  - Kommunikationsdomänen können sich (vollständig) überlappen.
- Im Programm werden MPI Kommunikationsdomänen durch **Kommunikatoren** repräsentiert.
  - Kommunikatoren haben den Typ **`MPI_Comm`**.
  - Alle MPI Aufrufe zum Nachrichtenaustausch erwarten als Argument einen Kommunikator.
  - Der Kommunikator **`MPI_Comm_World`** repräsentiert die Menge aller Prozesse der Berechnung.

Abfrage von Konfigurationsinformationen

- **`int MPI_Comm_size(MPI_Comm comm, int* size)`**
  - Liefert in **`size`** die Anzahl der Prozesse, die zur Kommunikationsdomäne des Kommunikators **`comm`** gehören.
- **`int MPI_Comm_rank(MPI_Comm comm, int* rank)`**
  - Liefert in **`rank`** den **Rang** (Integer ID) des aufrufenden Prozesses innerhalb der Kommunikationsdomäne des Kommunikators **`comm`**.
  - IDs werden mit 0 beginnend fortlaufend vergeben.
- Der aufrufende Prozess muss zur Kommunikations-domäne des angegebenen Kommunikators gehören.

„Hello World“ in MPI

```
#include <mpi.h>

main(int argc, char** argv)
{
  int nprocs, myrank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  printf("Hello World from process %d out of %d",
         myrank, nprocs);

  MPI_Finalize();
}
```

Ausführen von MPI Programmen

- Die genaue Prozedur zum Kompilieren und Starten eines MPI Programms ist implementierungsabhängig.
- Beispiel: MPICH Implementierung:
  - Kompilierung mittels Compiler Front-End **`mpicc`**:

    ```
    > mpicc –o hello_world hello_world.c
    ```

  - Kommandozeilen Programm **`mpirun`** startet parallele Berechnung mit der angegebenen Zahl an Prozessoren:

    ```
    > mpirun –np 8 hello_world
    ```

  - Konfigurationsfile (Machine File) legt die parallele Umgebung fest: >

    ```
    more /usr/mpich/share/machines.LINUX
    # Format: Knoten:Anzahl der Prozessoren:
    node1001:2
    node1002:2
    node1003:2
    node1004:2
    ...
    ```

MPI Datentypen

MPI Datentyp	Bedeutung/C Datentyp
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE`	8 Bit (keine Konversion)
`MPI_PACKED`	serialisierte Datenstruktur

Senden von Nachrichten

```
int MPI_Send(
    void *buf,                  /* Zeiger auf Sendepuffer */
    int count,                  /* Anzahl der Elemente */
    MPI_Datatype datatype,      /* Datentyp der Elemente */
    int dest,                   /* Empfänger */
    int tag,                    /* Message Tag */
    MPI_Comm comm)              /* Kommunikator */
```

- Der Sendepuffer **`buf`** enthält **`count`** konsekutive Elemente vom Typ **`datatype`**.
- Der Empfänger der Nachricht ist der Prozess mit Rang **`dest`** innerhalb der Kommunikationsdomäne **`comm`**.
- Mittels **`tag`** wird ein applikationsspezifischer Nachrichten-typ festgelegt (Wert von 0 bis **`MPI_TAG_UB`**).

Empfangen von Nachrichten

```
int MPI_Recv(
  void* buf,                /* Zeiger auf Empfangspuffer */
  int count,                /* Größe des Empfangspuffers */
  MPI_Datatype datatype,    /* Datentyp der Elemente */
  int source,               /* Sender */
  int tag,                  /* Message Tag */
  MPI_Comm comm             /* Kommunikator */
  MPI_Status* status)       /* Statusinformationen */
```

- Es werden Nachrichten vom Typ **`tag`** vom Prozess mit Rang **`source`** in der Kommunikationsdomäne **`comm`** empfangen.
  - Sind mehrere passende Nachrichten vorhanden, wird eine Nachricht zufällig ausgewählt.
  - Wildcards mittels **`MPI_ANY_SOURCE`** bzw. **`MPI_ANY_TAG`**
- **`count`** gibt die Größe (Anz. der Elem.) des Puffers **`buf`** an.
  - Fehler bei Pufferüberlauf: **`MPI_ERR_TRUNCATE`**

Empfangen von Nachrichten

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
}
```

- **MPI_SOURCE** und **MPI_TAG** geben genauere Auskunft bei Verwendung von **MPI_ANY_SOURCE** bzw. **MPI_ANY_TAG**.
- **MPI_ERROR** speichert den Fehlercode.
- Abfrage der Länge (Anz. der Elemente) der Nachricht:

```
int MPI_Get_count(
    MPI_Status* status,
    MPI_Datatype datatype,
    int* count)
```

  - **count** enthält Ergebnis

Semantik von MPI_Send und MPI_Recv

- **`MPI_Recv`** kehrt erst zurück, nachdem die entsprechende Nachricht empfangen und in den angegebenen Puffer kopiert wurde.
- MPI erlaubt für **`MPI_Send`** zwei unterschiedliche Implementierungsarten:
  - blockierend und gepuffert
  - blockierend und nicht-gepuffert
- MPI Programm sollten so entwickelt werden, dass sie für beide Implementierungsarten von **`MPI_Send`** korrekt sind.
  - Solche Programme werden als **safe** bezeichnet.

Beispiel

```
...
int a[10], b[10], myrank;
MPI_Status stat;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0) {
  MPI_Send(a,10,MPI_INT,1,1,MPI_COMM_WORLD);
  MPI_Send(b,10,MPI_INT,1,2,MPI_COMM_WORLD);
} else if (myrank == 1) {
  MPI_Recv(b,10,MPI_INT,0,2,MPI_COMM_WORLD,&stat);
  MPI_Recv(a,10,MPI_INT,0,1,MPI_COMM_WORLD,&stat);
}
```

Programm ist **nicht safe**.

Beispiel: Zirkuläre Kommunikation

```
...
int a[10], b[10], size, myrank;
MPI_Status stat;
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(a,10,MPI_INT,(myrank+1)%size,1,
          MPI_COMM_WORLD);
MPI_Recv(b,10,MPI_INT,(myrank-1+size)%size,1,
          MPI_COMM_WORLD,&stat);
```

Programm ist **nicht safe.**

Gleichzeitiges Senden und Empfangen mittels MPI_Sendrecv

```
int MPI_Sendrecv(
  void* sendbuf,           /* Zeiger auf Sendepuffer */
  int sendcount,           /* Größe der Nachricht */
  MPI_Datatype sendtype, /* Datentyp der Elemente */
  int dest,                /* Empfänger */
  int sendtag,             /* Message Tag */
  void* revbuf,            /* Zeiger auf Empfangspuffer */
  int recvcount,           /* Größe des Empfangspuffers */
  MPI_Datatype recvtype, /* Datentyp der Elemente */
  int source,              /* Sender */
  int recvtag,             /* Message Tag */
  MPI_Comm comm            /* Kommunikator */
  MPI_Status* status)      /* Statusinformationen */
```

- Argumente stellen Kombination der Argumente von **`MPI_Send`** und **`MPI_Recv`** dar.
- Empfangs- und Sendepuffer müssen verschieden sein.

Beispiel: Zirkuläre Kommunikation mit MPI_Sendrecv

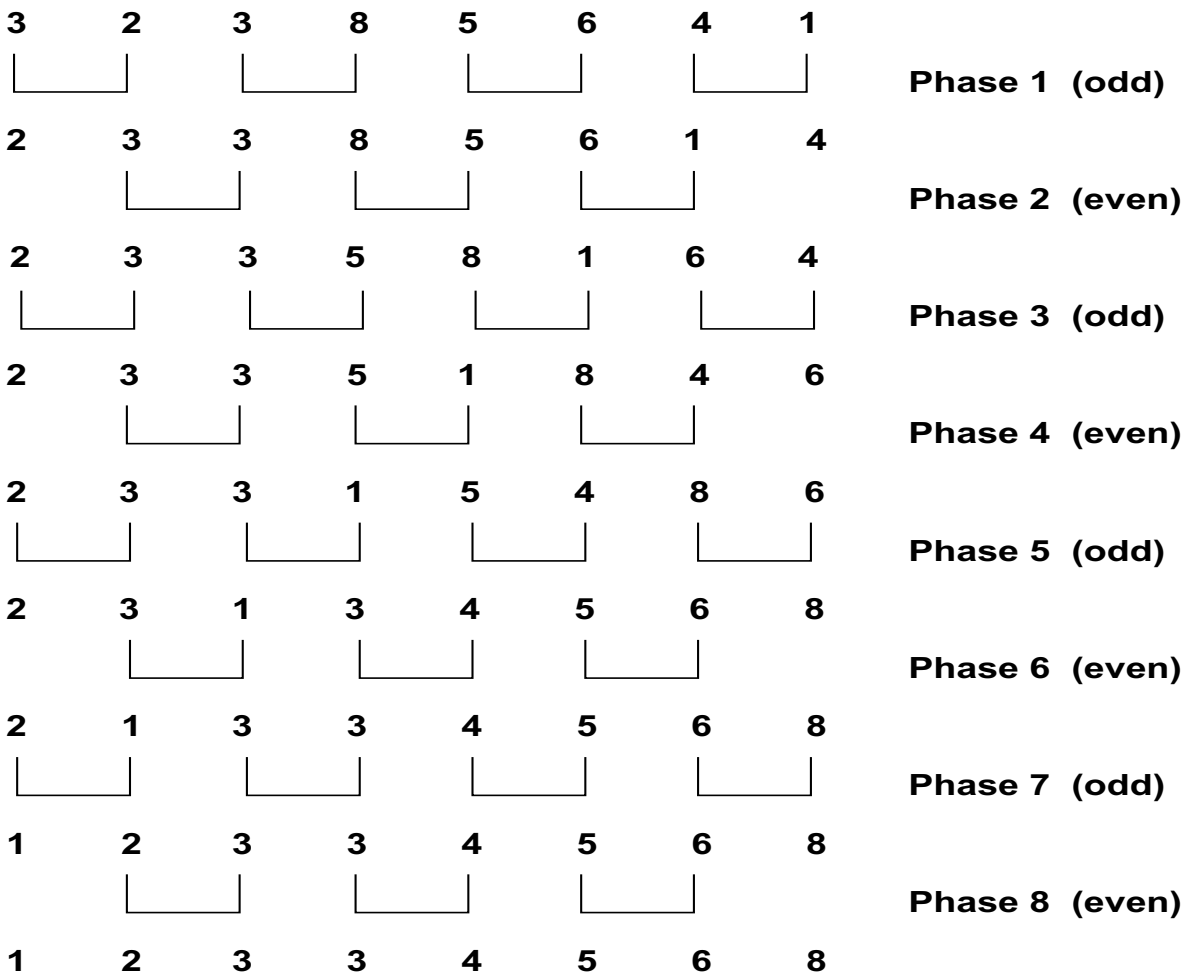
```
...
int a[10], b[10], size, myrank;
MPI_Status stat;
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Sendrecv(a,10,MPI_INT,(myrank+1)%size,1,
             b,10,MPI_INT,(myrank-1+size)%size,1,
             MPI_COMM_WORLD,&stat);
```

Odd-Even Sortier-Algorithmus

- Abgeleitet von Bubble-Sort

```
procedure ODD-EVEN (a_1, … , a_n)          //  NB: n gerade
   for i := 1 to n do                      // n Phasen
      if i is odd then                     // odd-Phase
         for j := 0 to n/2 − 1 do
            compare-exchange(a_2j+1, a_2j+2);
      if i is even then                    // even-Phase
         for j := 1 to n/2 − 1 do
            compare-exchange(a_2j , a_2j+1);
   end for
end ODD-EVEN
```

Beispiel


3 2 3 8 5 6 4 1
Phase 1 (odd)
2 3 3 8 5 6 1 4
Phase 2 (even)
2 3 3 5 8 1 6 4
Phase 3 (odd)
2 3 3 5 1 8 4 6
Phase 4 (even)
2 3 3 1 5 4 8 6
Phase 5 (odd)
2 3 1 3 4 5 6 8
Phase 6 (even)
2 1 3 3 4 5 6 8
Phase 7 (odd)
1 2 3 3 4 5 6 8
Phase 8 (even)
1 2 3 3 4 5 6 8

SPMD Odd-Even Sortier-Algorithmus

```
procedure ODD-EVEN-PAR (a_1, … , a_n)      // NB: n Prozesse, a_i ist bei P_i
   id = rank+1
   for i := 1 to n do                      // n Phasen
      if i is odd then                     // odd-Phase
         if id is odd then
            mp-compare-exchange(id+1);
         else
            mp-compare-exchange(id−1);
       if i is even then                   // even-Phase
         if id is even then
            mp-compare-exchange(id+1);
         else
            mp-compare-exchange(id−1);
         end for
end ODD-EVEN-PAR
```

MPI Implementierung des parallelen Odd-Even Sortier-Algorithmus

- siehe Programmlisting

Nicht-blockierende Kommunikation mit MPI_Isend und MPI_Irecv

```
int MPI_Isend(
  void *buf,                /* Zeiger auf Sendepuffer */
  int count,                /* Anzahl der Elemente */
  MPI_Datatype datatype,    /* Datentyp der Elemente */
  int dest,                 /* Empfänger */
  int tag,                  /* Message Tag */
  MPI_Comm comm,            /* Kommunikator */
  MPI_Request* request)     /* Ausführungsstatus */
```

- **`MPI_Isend`** kehrt i. Allg. zurück, bevor die Daten aus dem angegebenen Puffer **`buf`** kopiert wurden.
- Überlappung von Berechnung und Kommunikation

Nicht-blockierende Kommunikation mit MPI_Isend und MPI_Irecv

```
int MPI_Irecv(
  void *buf,                /* Zeiger auf Empfangspuffer */
  int count,                /* Größe des Empfangspuffers */
  MPI_Datatype datatype,    /* Datentyp der Elemente */
  int source,               /* Sender */
  int tag,                  /* Message Tag */
  MPI_Comm comm,            /* Kommunikator */
  MPI_Request* request)     /* Ausführungsstatus */
```

- **`MPI_Irecv`** kehrt i. Allg. zurück, bevor die Daten empfangen und in den angegebenen Puffer **`buf`** kopiert wurden.
- Blockierende und nicht-blockierende Kommunikations-operationen können gepaart werden.

Nicht-blockierende Kommunikation mit MPI_Isend und MPI_Irecv

```
int MPI_Test(
  MPI_Request* request,
  int* flag,
  MPI_Status* status)

int MPI_Wait(
  MPI_Request* request,
  MPI_Status* status)
```

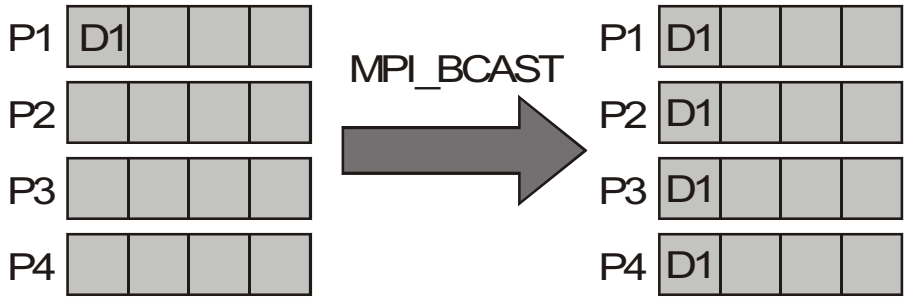
- **`MPI_Test`** prüft , ob die durch **`request`** spezifizierte nicht-blockierende Operation abgeschlossen ist.
  - Beendigung wird durch einen von 0 verschiedenen Wert für **`flag`** angezeigt.
  - Ist **`flag`** von 0 verschieden, so enthält **`status`** weitere Informationen (s.o.).
- **`MPI_Wait`** blockiert, bis die durch **`request`** spezifizierte nicht-blockierende Operation abgeschlossen ist.

MPI Gruppenkommunikation

- Gruppenkommunikation:
  - Mehr als 2 Prozesse kommunizieren gleichzeitig.
  - Kommunikation weist Muster auf.
- Menge der beteiligten Prozesse wird durch Kommunikator festgelegt.
  - Virtuelle Synchronisation: Alle Prozesse der Kommunikations-domäne des Kommunikators müssen die entsprechende Gruppenkommunikationsoperation aufrufen.
  - Keine Nachrichten-Tags möglich.
- Ausgezeichnete Prozesse (z.B. Sendeprozess) werden explizit gekennzeichnet.

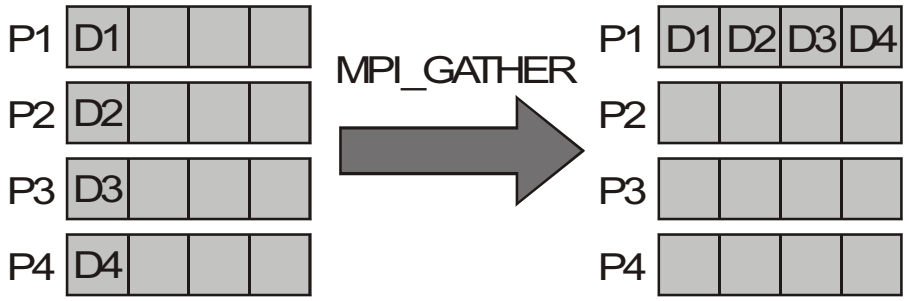
MPI Gruppenkommunikation: Broadcast

```
int MPI_Bcast(
  void *buf,                /* Sendepuffer bei root */
                            /* sonst Empfangspuffer */
  int count,                /* Anzahl der Elemente */
  MPI_Datatype datatype,    /* Datentyp der Elemente */
  int root,                 /* Sender-Prozess */
  MPI_Comm comm)            /* Gruppe */
```



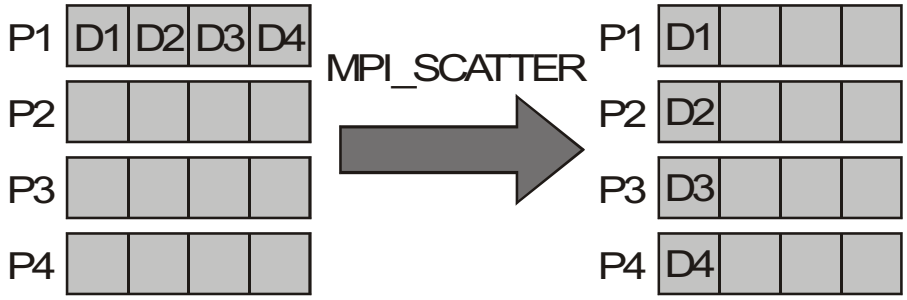
MPI Gruppenkommunikation: Gather

```
int MPI_Gather(
  void *sendbuf,            /* Sendepuffer */
  int sendcount,            /* Anzahl der Elemente */
  MPI_Datatype sendtype,    /* Datentyp der Elemente */
  void *recvbuf,            /* Empfangspuffer bei target */
  int recvcount,            /* Anz. Elem. (pro Prozess) */
  MPI_Datatype recvtype,    /* Datentyp der Elemente */
  int target,               /* Empfänger-Prozess */
  MPI_Comm comm)            /* Gruppe */
```



MPI Gruppenkommunikation: Scatter

```
int MPI_Scatter(
  void *sendbuf,           /* Sendepuffer */
  int sendcount,           /* Anz. Elem. (pro Prozess)*/
  MPI_Datatype sendtype, /* Datentyp */
  void *recvbuf,           /* Empfangspuffer */
  int recvcount,           /* Anzahl der Elemente */
  MPI_Datatype recvtype, /* Datentyp */
  int root,                /* Sender-Prozess */
  MPI_Comm comm)           /* Gruppe */
```



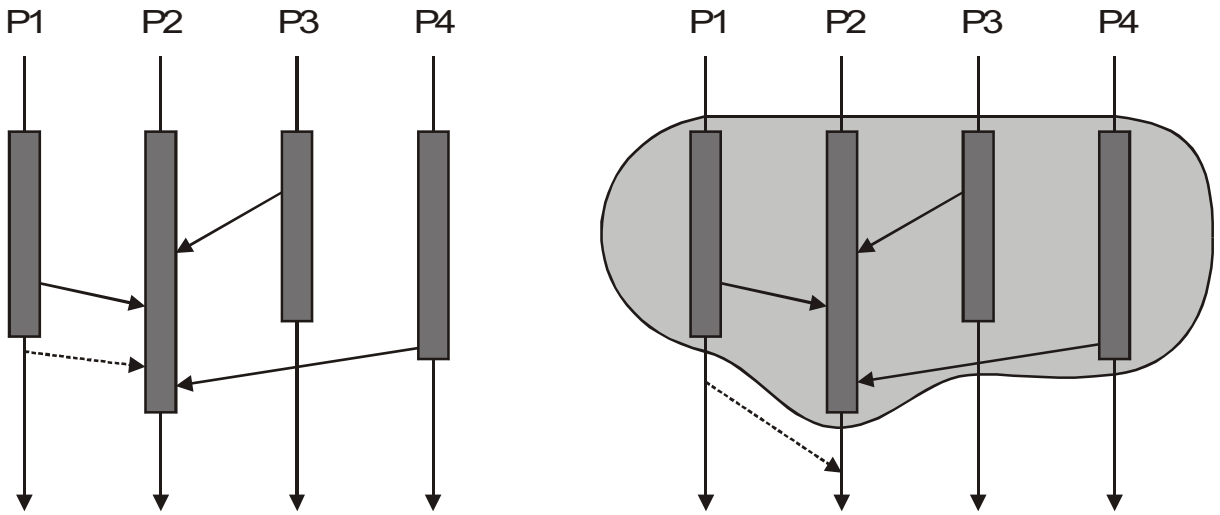
Erzeugung neuer Kommunikationsdomänen

```
int MPI_Comm_split(MPI_Comm comm, int color,
  MPI_Comm* new_comm)
```

- **`MPI_Comm_split`** wird von allen Prozessen der Kommunikationsdomäne **`comm`** aufgerufen.
- Die Kommunikationsdomäne comm wird in disjunkte Teilgruppen partitioniert.
  - Eine Teilgruppe enthält diejenigen Prozesse, die den selben Wert für **`color`** übergeben haben.

Modulare Programmierung in MPI

```
MPI_Comm comm, newcom;
/* neuer Kontext: neue Kommunikationsdomäne mit allen
  Prozessen der alten Kommunikationsdomäne */
MPI_Comm_dup(comm, &newcomm);
function(newcomm, ...);
MPI_Comm_Free(newcomm);
```



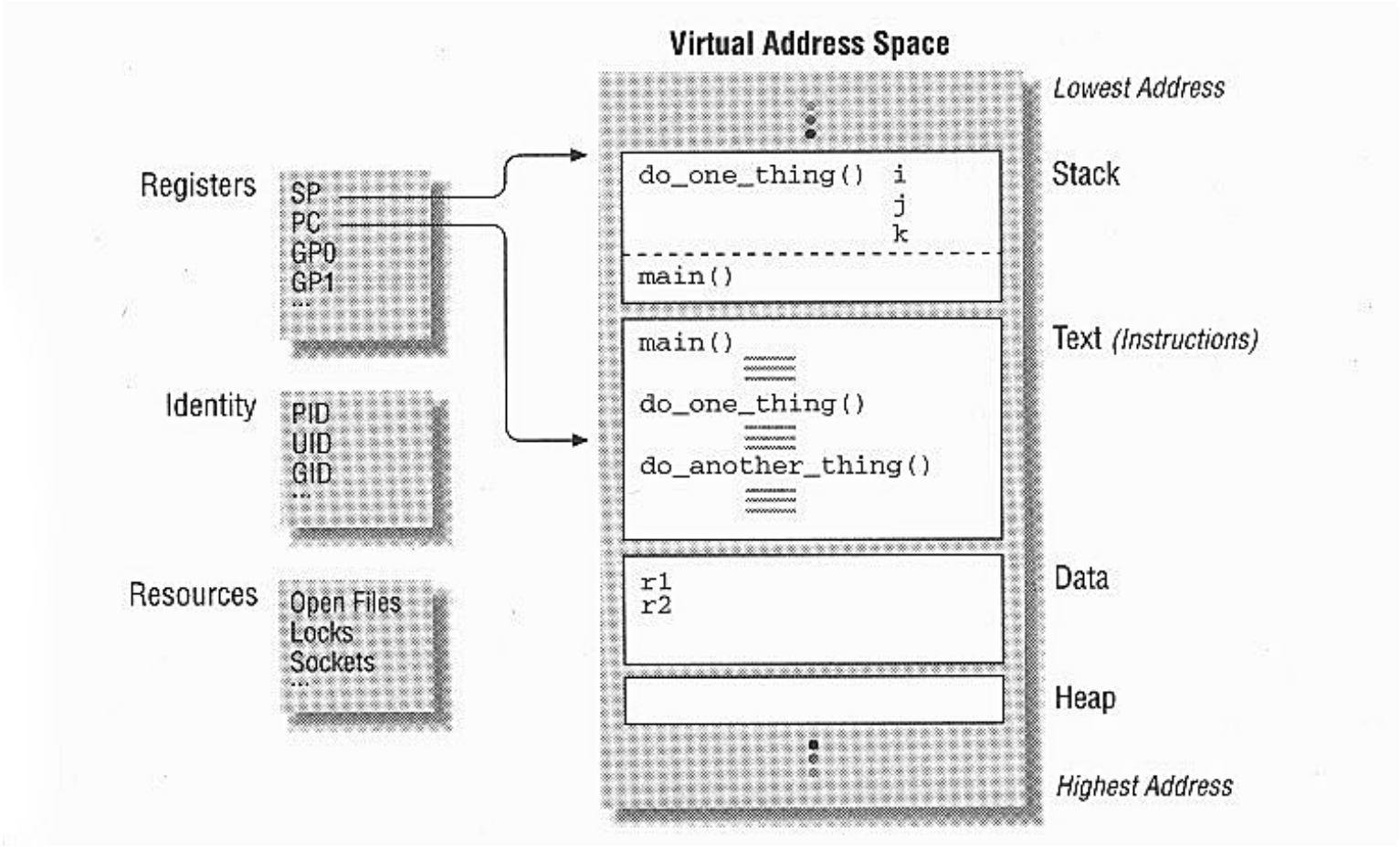
4. Parallele Programmiermodelle

1. Klassifikation paralleler Programmiermodelle
2. Beispiele für parallele Programmiermodelle
3. Message Passing Interface (MPI)
4. **OpenMP**

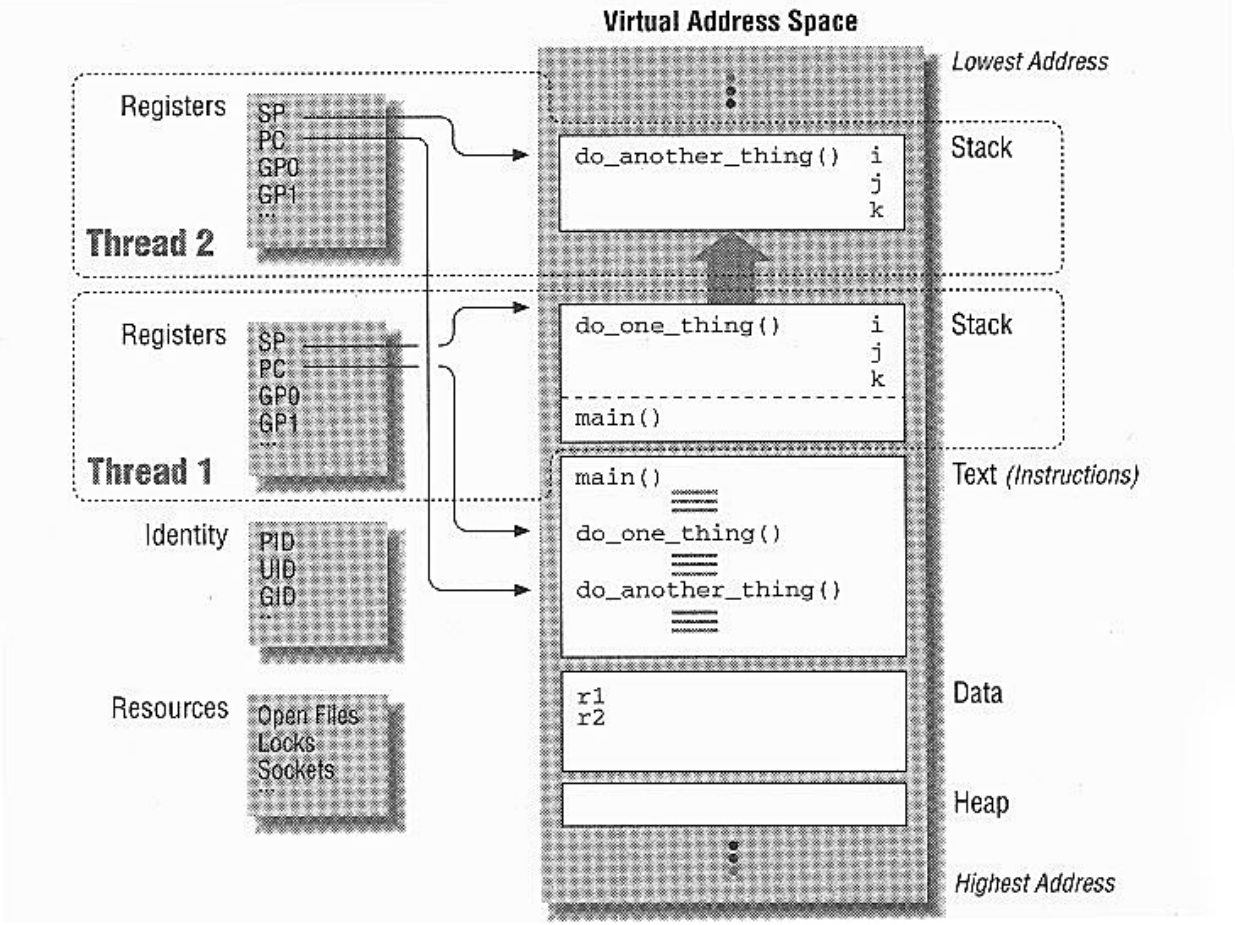
Einführung: Threads

- Moderne Betriebssysteme unterstützen mehrere Kontrollflüsse (**Threads of Control)** pro Prozess.
  - Alle Threads teilen sich den virtuellen Adressraum des Prozesses.
  - Jeder Thread besitzt eigenen Kontext:
    - Registersatz incl. Stackpointer (SP) und Programcounter (PC)
    - Stack
- Bei Parallelrechnern mit gemeinsamem Adressraum können die Threads eines Prozesses auf verschiedenen Prozessoren ausgeführt werden.
- Es existieren umfangreiche APIs zur Erstellung von Programmen mit mehreren Threads :
  - POSIX Thread für Unix Betriebssysteme
  - Win32 Threads für MS Windows

Traditionelles Prozessmodell



Prozessmodell mit mehreren Threads



Beispielprogramm (POSIX Threads)

```
#include <pthread.h>

void do_one_thing(int* x) {int i,j,k; ... };
void do_another_thing(int* y) {int i,j,k; ... };

main() {
  int a = 12;
  int b = 17;
  pthread_t thread1, thread2;

  pthread_create(&thread1, NULL,
                 (void*)do_one_thing, (void*)&a);
  pthread_create(&thread2, NULL,
                 (void*)do_another_thing, (void*)&b);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
}
```

Überblick OpenMP

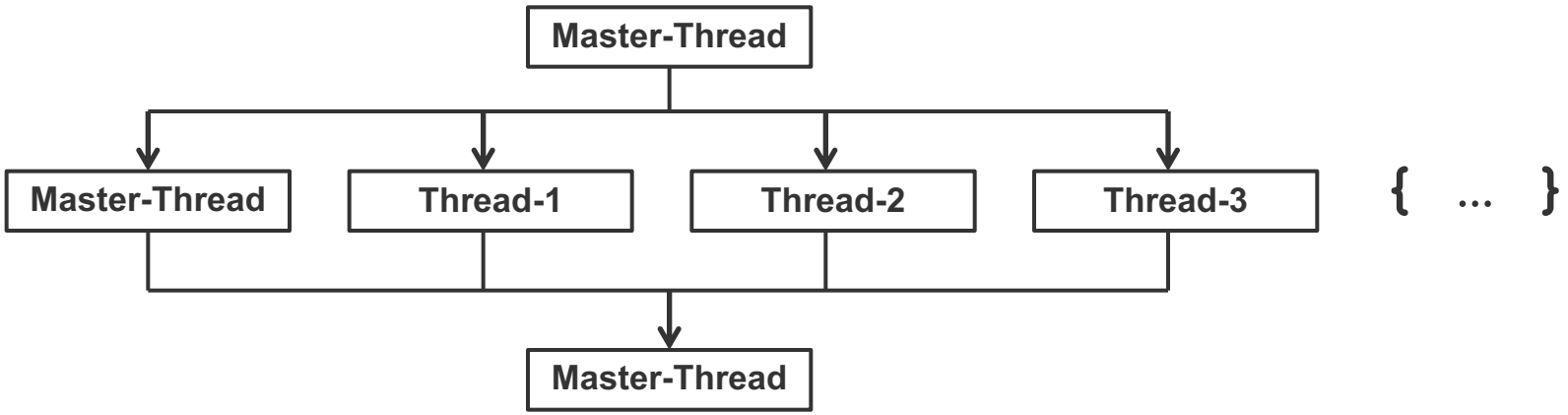
- Standardisiertes API zur Programmierung von Parallel-rechnern mit gemeinsamem Adressraum in C/C++ und Fortran.
- OpenMP befindet sich auf höherer Abstraktionsebene als die Thread APIs modernen Betriebssysteme.
  - POSIX Threads: Systemprogrammierer
  - OpenMP: Anwendungsprogrammierer
- OpenMP basiert auf:
  - Compiler Direktiven

    `#pragma omp directive [clause list]`
    - **`directive:`** Name der Direktive
    - **`clause list:`** Liste von Klauseln
  - Bibliotheksaufrufen
  - Umgebungsvariablen

Die parallel Direktive

```
#pragma omp parallel [clause list]
  { ... /* structured block */ ... }
```

- OpenMP Programme werden ab dem Auftreten einer **`parallel`** Direktive parallel ausgeführt.
  - Es wird eine Gruppe von Threads erzeugt welche jeweils den nachfolgenden Block ausführen.
  - Der aufrufende Thread wird zum Master-Thread der Gruppe.



Die parallel Direktive

- Festlegung der Anzahl der erzeugten Threads:
  - Statisch mittels der Klausel **`num_threads(integer expr)`**
  - Dynamisch:
    - Systemaufruf: **`omp_set_num_threads(int num_threads)`**
    - Umgebungsvariable: **`OMP_NUM_THREADS`**
  - Priorisierung erfolgt gemäß der obigen Reihenfolge
  - Falls keine Angaben gemacht werden, wird auf einen von der Implementierung festgelegten Wert zurückgegriffen.
    - In der Regel die Anzahl der vorhandenen Prozessoren

Die parallel Direktive: Verwendungsart der Variablen des Master-Threads

- **`shared(variable list)`** Klausel
  - Alle Threads der Gruppe arbeiten auf den angegebenen Variablen des Master-Threads.
- **`private(variable list)`** Klausel
  - Jeder Thread arbeitet jeweils auf einer (privaten) Kopie der angegeben Variablen.
- **`firstprivate(variable list)`** Klausel
  - Wie **`private`**, zusätzlich werden die Kopien der Variablen mit den Werten des Master-Threads initialisiert.
- **`reduction(operator: variable list)`** Klausel
  - Alle Kopien der aufgelisteten (privaten) Variablen werden mittels des angegebenen skalaren Operators verknüpft.
  - Das Ergebnis wird den entsprechenden Variablen des Master-Threads zugewiesen.

Die parallel Direktive: Verwendungsart der Variablen des Master-Threads

- **`default(none|shared|private)`** Klausel
  - **`none:`** Die Verwendungsart muss für jede vorkommende Variable spezifiziert werden (Schutz vor Fehlern).
  - **`shared:`** Falls nicht anders angeben, ist die Verwendungsart einer Variablen „shared“.
  - **`private:`** Falls nicht anders angeben, ist die Verwendungsart einer Variablen „private“.

Übersetzung: OpenMP nach POSIX Threads

```
int a, b;
main() {
  // serial segment

  {
  }
}
```

OpenMP Programm

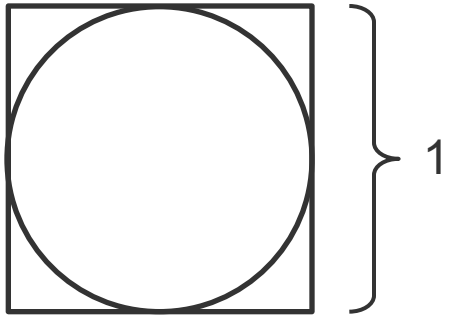
```
int a, b;
main() {
  // serial segment

}

  int a;
  // parallel segment
}
```

Erzeugter Pthreads Code

Beispiel:
Näherungsverfahren zur Berechnung von PI

- Es wird zufällig eine große Anzahl von Punkten in einem Quadrat mit Kantenlänge 1 markiert in das ein Kreis mit Radius 0,5 einbeschrieben ist.

  - Die Fläche des Quadrats beträgt 1.
  - Die Fläche des Kreises beträgt PI/4.



- Das Verhältnis der Anzahl der Punkte, die im Kreis liegen zur Gesamtzahl der Punkte stellt ein Näherungswert für PI/4 dar.

Beispiel: Näherungsverfahren zur Berechnung von PI

```
#pragma omp parallel default(private) shared(npoints)\
            reduction(+: sum) num_threads(8)
{
  num_threads = omp_get_num_threads();
  points_per_thread = npoints / num_threads;
  sum = 0;
  for (i=0; i<points_per_thread; i++) {
    rand_no_x = (double)rand()/(double)RAND_MAX;
    rand_no_y = (double)rand()/(double)RAND_MAX;
    if (((rand_no_x-0.5)*(rand_no_x-0.5) +
        (rand_no_y-0.5)*(rand_no_y-0.5)) < 0.25)
     sum++;
  }
}
```

Auszeichnung von Parallelität

- Die **`parallel`** Direktive kann mit den folgenden Direktiven kombiniert werden, um Parallelität flexibler auszuzeichnen:
  - **`sections`** Direktive: Spezifikation verschiedener unabhängiger Tasks.
  - **`for`** Direktive: Parallele Ausführung von Schleifeniterationen.
- Diese Direktiven werden ignoriert, falls keine **`parallel`** Direktive vorangestellt ist, d.h. die Ausführung erfolgt dann weiterhin sequentiell.

Die sections Direktive

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            procedureA();
        }
        #pragma omp section
        {
            procedureB();
        }
    }
}
```

Die for Direktive

```
#pragma omp for [clause list]
   /* for loop */
```

- Mittels der **`for`** Direktive können Schleifeniterationen einer **`for`** Schleife auf einzelne Threads zur parallelen Ausführung verteilt werden.
- Anforderungen an **`for`** Schleife:
  - Schleife darf keine **`break`** Anweisung enthalten
  - Schleifenvariable muss vom Typ **`int`** sein
  - ...
- Mittels der **`schedule`** Klausel wird die Zuordnung der Schleifeniterationen zu den Threads gesteuert.

Beispielprogramm: Berechnung von Pi

```
#pragma omp parallel default(private) shared(npoints)\
          reduction(+: sum) num_threads(8)
{
  sum = 0;
  #pragma omp for schedule(static)
  for (i=0; i<npoints; i++) {
    rand_no_x = (double)rand()/(double)RAND_MAX;
    rand_no_y = (double)rand()/(double)RAND_MAX;
      if (((rand_no_x-0.5)*(rand_no_x-0.5) +
         (rand_no_y-0.5)*(rand_no_y-0.5)) < 0.25)
     sum++;
  }
}
```

Die static Scheduling Klasse

```
schedule(static [, chunk_size])
```

- Die Schleifeniterationen werden in Chunks der Größe **`chunk_size`** aufgeteilt.
- Die Chunks werden in einem round robin Verfahren den Threads (statisch) zugewiesen.
- Ist kein Wert für **`chunk_size`** angegeben, so wird für jeden Thread ein Chunk gebildet.

Die dynamic Scheduling Klasse

```
schedule(dynamic [, chunk_size])
```

- Die Schleifeniterationen werden in Chunks der Größe **`chunk_size`** aufgeteilt .
- Die Chunks werden dynamisch freien Threads zugewiesen.
- Ist kein Wert für **`chunk_size`** angegeben, so wird für jede Iteration ein Chunk gebildet.
- Verwendung z.B. bei
  - Parallelrechnern mit unterschiedlich leistungsfähigen Prozessoren und/oder
  - unterschiedlicher Berechnungskomplexität einzelner Iterationen.

Die guided Scheduling Klasse

`schedule(guided [, chunk_size])`

- Die Größe der Chunks wird bei jeder Zuweisung exponentiell reduziert.
- Die kleinste Größe eines Chunks ist **`chunk_size`**.
- Die Chunks werden dynamisch freien Threads zugewiesen.
- Der Defaultwert für **`chunk_size`** ist 1.

Die runtime Scheduling Klasse

`schedule(runtime)`

- Die Scheduling Klasse und Größe der Chunks wird mittels der Umgebungsvariablen **`OMP_SCHEDULE`** festgelegt.
- Beispiele:
  - **`setenv OMP_SCHEDULE "static,4"`**
  - **`setenv OMP_SCHEDULE "guided"`**

Synchronisationskonstrukte in OpenMP

- **`barrier`** Direktive
  - Barrier Synchronisationskonstrukt
- **`single`** Direktive / **`master`** Direktive
  - Sequentialisierung
- **`critical`** Direktive
  - Kritischer Abschnitt, wechselseitiger Ausschluss

Die barrier Direktive

```
#pragma omp barrier
```

- Alle Threads in einer Gruppe warten, bis alle zur Stelle der barrier Direktive vorangeschritten sind.

Die single/master Direktiven

```
#pragma omp single
{  ... /* structured block */ ... }
```

- Der angegebene Block wird nur von einem Thread der Gruppe ausgeführt.
  - Der ausführende Thread wird zufällig bestimmt.
  - Am Ende des Blocks wird implizit eine Barrier Synchronisation ausgeführt.

```
#pragma omp master
{  ... /* structured block */ ... }
```

- Der angegebene Block wird nur vom Master Thread der Gruppe ausgeführt.
  - Es erfolgt keine implizite Barrier Synchronisation.

Die critical Direktive

```
#pragma omp critical [(name)]
  {  ... /* critical region */ ... }
```

- Definition kritischer Abschnitte
  - Ein kritischer Abschnitt besteht aus allen ausgezeichneten (nicht notw. konsekutiven) Programmteilen.
  - Die Klausel **`name`** ermöglicht Auszeichnung unterschiedlicher kritischer Abschnitte.
- In einem kritischen Abschnitt befindet sich zu jedem Zeitpunkt der Ausführung höchstens ein Thread.
  - Falls sich ein Thread im kritischen Abschnitt befindet, werden andere Threads die zur **`critical`** Direktive gelangen blockiert, bis der Thread den kritischen Abschnitt verlassen hat.

Die atomic Direktive

```
#pragma omp atomic
  expression-statement
```

- Spezialfall der **`critical`** Primitive: Kritischer Abschnitt besteht aus Aktualisierung einer einzelnen Speicherstelle der Form:
  - **`x++, x--`**
  - **`++x, --x`**
  - **`x <binary operator> = <expr>`**
- Effiziente Implementierung durch spezielle Maschinen-instruktionen möglich.