7. MPI – Message Passing Interface

- de facto Standard für nachrichten-basierte Programmierung
- für FORTRAN, C und C++

=> www.mpi-forum.de

MPI Programmiermodell

- SPMD- Programmstruktur
- 6 Basisfunktionen:
  - `MPI_Init` Initialisierung
  - `MPI_Finalize` Terminierung
  - `MPI_Comm_rank` Prozessid abfragen
  - `MPI_Comm_size` Anzahl Prozesse bestimmen
  - `MPI_Send` Senden
  - `MPI_Recv` Empfangen
- Namenskonventionen:
  - MPI_ als Präfix für alle Routinen und Konstanten
  - Konstanten in Großbuchstaben
  - FORTRAN nur Großbuchstaben, C Groß- und Kleinbuchstaben

Struktur eines MPI Programms in C

```
#include “mpi.h“

main (int argc, char **argv)
{
   int my_rank, nprocs

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   ...
   ...
   MPI_Finalize();
}
```

Initialisierung und Terminierung

- `MPI_Init (int *argc, char ***argv)`

  Adressen der Argumente von main

  - initialisiert die MPI-Umgebung
  - definiert den sogenannten Kommunikator MPI_COMM_WORLD

- Ein Kommunikator definiert
  - eine Gruppe von Prozessen und
  - einen Kommunikationskontext.

  MPI_COMM_WORLD umfasst alle Prozesse.

- `MPI_Finalize()`
  - beendet MPI.
  - Es sind keine weiteren MPI-Aufrufe möglich.

Abfrageroutinen

- **MPI_Comm_size (MPI_Comm comm, int *size)**

  Eingabe Ausgabe

  ermittelt die Anzahl der Prozesse in einem Kommunikator.

- **MPI_Comm_rank (MPI_Comm comm, int *size)**

  Eingabe Ausgabe

  bestimmt den Rang oder die Identifikationsnummer eines Prozesses innerhalb eines Kommunikators.

  Die Prozesse sind innerhalb eines Kommunikators von 0 bis #Prozesse - 1 durchnummeriert.

Senden

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

  zu übertragende Daten
  Rang des Empfängers
  Kennung
  Kommunikator

  ist die Standard-Senderoutine.

„count“ Datenobjekte vom Typ „datatype“ ab Adresse „buf“
(-> Sendepuffer) werden
mit der Kennung „tag“ an den Prozess mit Rang „dest“
innerhalb des Kommunikators „comm“ gesendet.

Empfangen

- **MPI_Recv (void *buf,**
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm comm,
  MPI_Status status)

zu empfangende Daten

Rang des Senders

Kennung

Kommunikator

source, tag, count
tatsächlich empfangener Daten

Nachrichten sind Felder eines speziellen MPI-Datentyps umgeben von einem Nachrichtenumschlag mit
Sender – Empfänger – Tag – Kommunikator.

MPI-Datentypen

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

Beispielprogramm

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (size < 2) {  printf("Need at least 2 processes.\n");
                 MPI_Finalize();
                 return(1);
}
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (0 == rank) {
    MPI_Send(buf, BUFSIZE, MPI_INT, 1, 11, MPI_COMM_WORLD);
    printf("rank %d sent message\n", rank);
}
else if (1 == rank) {
    MPI_Recv(buf, BUFSIZE, MPI_INT, 0, 11, MPI_COMM_WORLD,
           &status);
    printf("rank %d received message\n", rank);
}
MPI_Finalize();
```

MPI_ANY_TAG und MPI_ANY_SOURCE

- **Beim Nachrichtenempfang können mittels**

  MPI_ANY_TAG und MPI_ANY_SOURCE

  Nachrichten mit beliebigem Tag bzw. beliebigem Sender empfangen werden. Vorsicht Nichtdeterminismus!

- **In diesem Fall können das empfangene Tag und der tatsächliche Sender aus dem Status-Argument ermittelt werden.**
  In C ist status ein Record (Verbund) mit den Feldern MPI_SOURCE und MPI_TAG. => Zugriff: status.MPI_SOURCE bzw. status.MPI_TAG

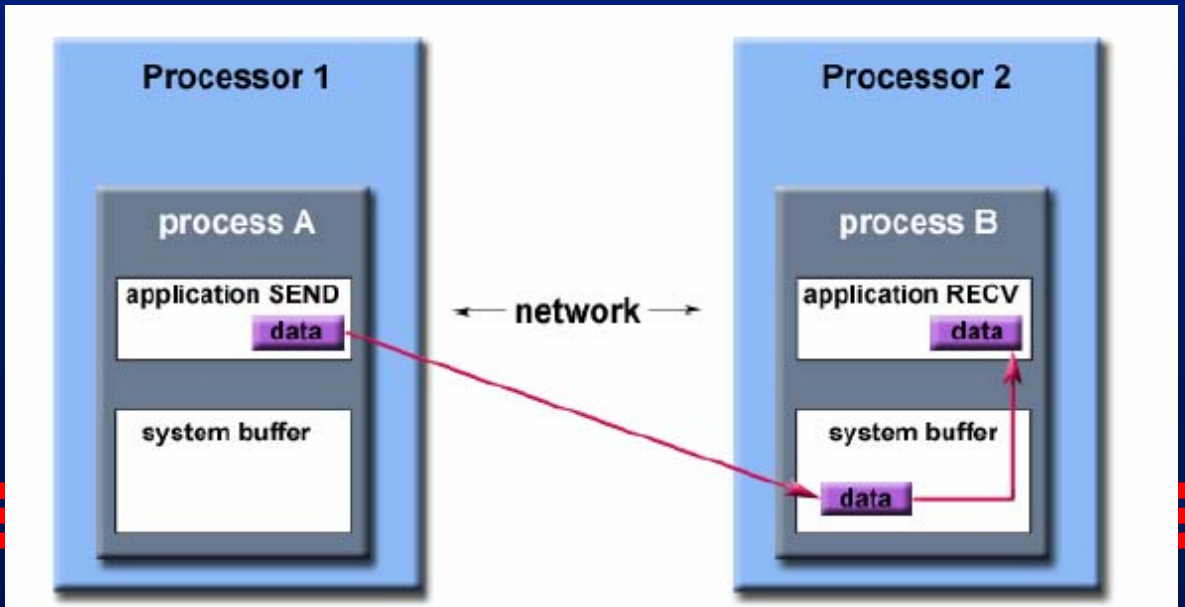
- **MPI_Get_count (** **MPI_STATUS status,** } Eingabe
  MPI_DATATYPE datatype, }
  int *count) } Ausgabe

  liefert die Anzahl tatsächlich empfangener Datenelemente.

Punkt-zu-Punkt-Kommunikation

- MPI unterstützt
  - Punkt-zu-Punkt-Kommunikation (1:1)
  - kollektive Kommunikation

- Punkt-zu-Punkt
  - genau zwei Prozesse, Sender und Empfänger, identifiziert durch ihren Rang innerhalb eines Kommunikators



Arten von Sendefunktionen

- Standard MPI_Send
  - blockierende Basis-Sende-Operation
  - Standard lässt konkrete Implementierung offen
- gepuffert MPI_Bsend
  - Nachrichten werden gepuffert, damit das Senden unabhängig vom Empfang abgeschlossen werden kann
  - lokale Operation
  - Bereitstellung von Pufferbereich durch Programmierer erforderlich
- synchron MPI_Ssend
  - Senden endet erst, wenn Empfang der Daten begonnen hat
  - nicht-lokale Operation
- ready MPI_Rsend
  - sofortiges Senden (ohne Zwischenpufferung)
  - Empfängerprozess muss zum Empfang bereit sein
  - lokale Operation

Blockierend vs nicht-blockierend

- **Blockierend bedeutet:**

  Sendefunktion wird erst beendet, wenn die Nachrichteninitiierung so weit fortgeschritten ist, dass der (Anwendungs-) Sendepuffer nicht durch eine nachfolgende Anweisung überschrieben werden kann.

- **Nicht-blockierend bedeutet:**

  Der Benutzer ist selbst dafür verantwortlich, dass die abgehende Nachricht nicht korrumpiert wird. Das MPI System wird nur mit der Nachrichtensendung beauftragt.
  => Überlappung von Kommunikation und Berechnung.

  MPI_Isend - MPI_Ibsend - MPI_Irsend - MPI_Issend

Empfangsfunktionen / Pufferverwaltung

- Es gibt nur eine Empfangsoperation, unabhängig vom Sendemodus, die
  - blockierend (Standard) oder
  - nicht-blockierend (MPI_Irecv)

  sein kann.

- Pufferverwaltung für MPI_Bsend / MPI_Ibsend
  - MPI_Buffer-attach (void *buffer, int size)

    Bereitstellung von Pufferplatz, size in Bytes

  - MPI_Buffer_detach (void *buffer, int size)

    Freigabe von Pufferplatz

Abfrageargument bei nicht-blockierendem Senden und Empfangen

- Bei den nicht-blockierenden Sende- und Empfangsanweisungen kann über ein zusätzliches Argument

  MPI_Request *request,

  das zurückgeliefert wird, später getestet werden, ob die Kommunikationsanweisung abgeschlossen ist.
- MPI_Wait(&request, &status)

  wartet, bis die zugehörige Kommunikationsanweisung fertig ist.
- MPI_Test(&request, &flag, &status)

  zeigt mit flag, ob die Kommunikationsanweisung fertig ist oder nicht.

typische Konstellation:

```
MPI_Isend/Irecv (...., &request)
... /* Berechnungen */
MPI_Wait (&request, &status)
```

Semantik der Punkt-zu-Punkt-Komm.

- Reihenfolge von Nachrichten mit identischen Umschlägen bleibt erhalten
- Werden passende Sende-/Empfangsanweisungen initiiert, so wird mindestens eine der Anweisungen abgeschlossen.

Beispiel:

```
MPI_Comm-rank(comm,rank);
if (rank == 0)
    { MPI_Bsend (buf1, count, MPI_REAL, 1, tag1, comm);
      MPI_Ssend (buf2, count, MPI_REAL, 1, tag2, comm); }
else
    { MPI_Recv (buf1, count, MPI_REAL, 0, tag2, comm, &status);
      MPI_Recv (buf2, count, MPI_REAL, 0, tag1, comm, &status); }
```

- keine Garantie von Fairness

Kollektive Kommunikation

- Typen
  - globale Barriere
  - globale Datenbewegungen
  - Reduktionsoperationen
- MPI_Barrier (MPI_Comm comm)

  bewirkt globale Synchronisation aller Prozesse im Kommunikator comm

- MPI_Bcast(void *inbuf, int incnt, MPI_Datatype intype,
  int root, MPI_Comm comm)

  bewirkt Broadcast von dem Prozess mit Rang root zu allen anderen Prozessen im Kommunikator comm.

  Die ersten drei Argumente sind analog zu send/receive.

Aufsammeln und Verteilen von Daten

- **MPI_Gather / MPI_Scatter**

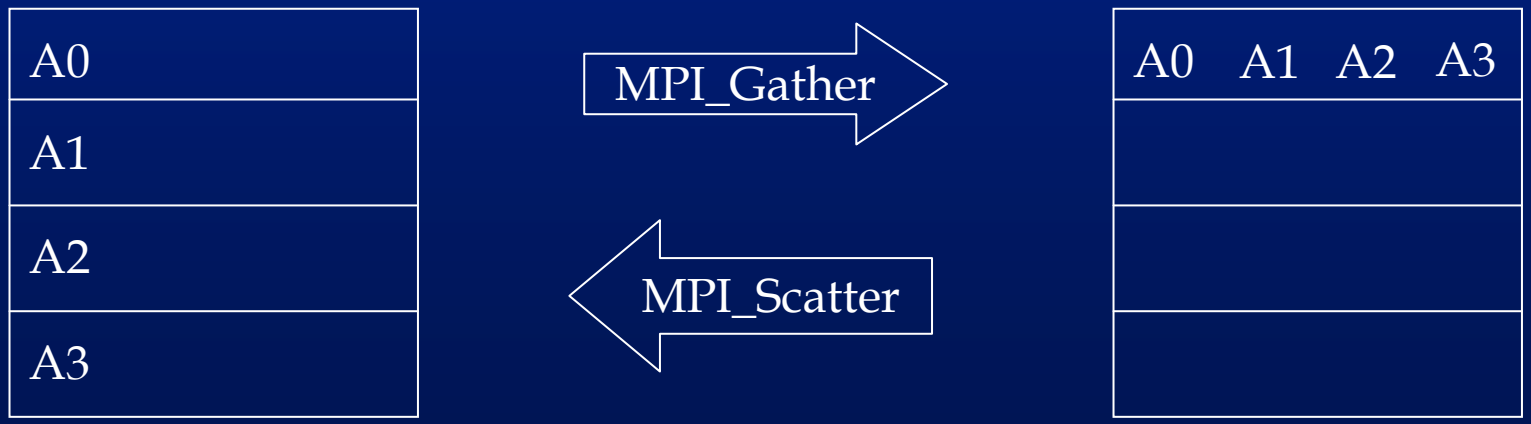
  (void *inbuf, int incnt, MPI_Datatype intype,

  Eingabe

  void *outbuf, int outcnt, MPI_Datatype outtype,

  Ausgabe

  int root, MPI_Comm comm)



Reduktionsoperationen

MPI_Reduce / MPI_Allreduce (void *inbuf, void *outbuf, int count, MPI_Datatype type, MPI_Op op, int root / _ , MPI_Comm comm)

MPI Reduction Operation		C Data Types
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical AND	integer
MPI_BAND	bit-wise AND	integer, MPI_BYTE
MPI_LOR	logical OR	integer
MPI_BOR	bit-wise OR	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double
MPI_MINLOC	min value and location	float, double and long double

Zeitmessungen in MPI

- **MPI_Wtime ()** wall clock time

  liefert als Gleitkommazahl mit doppelter Präzision eine Sekundenzahl bzgl. eines festen, d.h. während der Laufzeit eines Prozesses unveränderten Zeitpunkts in der Vergangenheit

  Zeitmessungen erfordern mehrere Aufrufe dieser Routine:

```
{ double starttime, endtime;
  starttime = double MPI_Wtime();
    ...
  endtime = double MPI_Wtime();
  printf(„That took %f seconds\n“, endtime-starttime); }
```

- **MPI_WTick()**

  liefert eine Gleitkommazahl, die die Zeit zwischen aufeinanderfolgenden Uhrticks angibt.

  => Genauigkeit von Zeitmessungen

MPI_Sendrecv

Gekoppeltes Senden und Empfangen:

- Senden und gleichzeitiges Empfangen einer Nachricht
- blockierend bis Sendepuffer frei ist und Empfangspuffer empfangene Daten enthält

```
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)
```

Kommunikatoren und Prozessgruppen

- Nachrichtenkennungen (tags)
  - Unterscheidung und Selektion von Nachrichten
- Prozessgruppen
  - Einschränkung des Empfangs von Nachrichten und der Anwendung einer kollektiven Kommunikation

werden in fast allen Kommunikationsbibliotheken unterstützt.

neu in MPI: Kommunikatoren

- bessere Kapselung von Bibliotheksfunktionen
- Erleichterung der Arbeit mit Prozessgruppen und virtuellen Topologien

Kommunikator-Definition

Ein Kommunikator

- bestimmt eine Gruppe von Prozessen, d.h. eine geordnete Menge von Prozessen mit lokalem Rang $\in$ {0, ..., #Prozesse – 1}
  => Umgebung für Punkt-zu-Punkt- und kollektive Kommunikationen
- definiert einen Kontext für Kommunikationen (implizit)
  => Einführung separater, sicherer, d.h. sich nicht beeinflussender Universen zum Nachrichtenaustausch, wichtig für Unterprogrammbibliotheken

vordefinierte Kommunikatoren:

- MPI_COMM_WORLD umfasst die Gruppe aller Prozesse
- MPI_COMM_SELF enthält nur den Prozess selbst
- MPI_COMM_NULL ungültiger Kommunikator

Prozessgruppen: Konstruktion, Analyse und Manipulation

- **MPI_Comm_group** (MPI_Comm comm, MPI_Group *group)
  liefert die Prozessgruppe des Kommunikators comm.

- **MPI_Group_union/intersection/difference**
  (MPI_Group group1, MPI_Group group2, MPI_Group group3)
  bestimmt die Vereinigung, den Schnitt und die Differenz von Gruppen.
  Vereinigung und Schnitt sind assoziativ, aber wegen der Prozessnummerierung nicht kommutativ.

- **MPI_GROUP_EMPTY** ist die leere Prozessgruppe.

- **MPI_Group_free** (MPI_Group group) deallokiert eine Prozessgruppe. group wird an **MPI_GROUP_NULL** gebunden.

Prozessgruppen Fortsetzung

- **MPI_Group-size** (MPI_Group group, int *size)
  liefert in size die Anzahl der Prozesse einer Gruppe.

- **MPI_Group_rank** (MPI_Group group, int *rank)
  liefert den Rang eines Prozesses in einer Gruppe und,
  falls der Prozess nicht in der Gruppe ist, MPI_UNDEFINED.

- **MPI_Group_compare**
  (MPI_Group group1, MPI_Group group2, int *result)
  liefert in result:
  - MPI_IDENT bei identischen Gruppen, d.h. dieselben Prozesse mit derselben Nummerierung
  - MPI_SIMILAR bei Gruppen mit denselben Prozessen, aber unterschiedlicher Nummerierung
  - MPI_UNEQUAL sonst.

Prozessgruppen Fortsetzung 2

- MPI_Group_incl /excl

  (MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

  erzeugt eine neue Gruppe mit n Prozessen,

  die in group die Ränge ranks[0] ... ranks[n-1] haben und

  in newgroup die Ränge 0 ... n-1.

  Prozess ranks[i] in group ist Prozess i in newgroup.

  Die in ranks angegebenen Ränge müssen in group vorkommen und paarweise verschieden sein.

  => Umordnung von Elementen einer Gruppe

  Die excl-Variante streicht die durch ranks angegebenen Prozesse aus der Gruppe.

Erzeugung/Freigabe von Kommunikatoren

Neue Kommunikatoren können aus bestehenden Kommunikatoren oder aus Prozessgruppen gebildet werden.

Die folgenden Funktionen sind kollektive Funktionen, d.h. sie müssen von allen Prozessen eines Kommunikators aufgerufen werden.

- MPI_Comm_dup (MPI_Comm comm, MPI_Comm *newcomm)
  erzeugt einen neuen Kommunikator mit derselben Prozessgruppe, denselben assoziierten Informationen, aber einem neuen Kommunikationskontext.
- MPI_Comm_free (MPI_Comm *comm)
  Deallokation von comm und Rückgabe von MPI_COMM_NULL

Erzeugung von Kommunikatoren aus Prozessgruppen

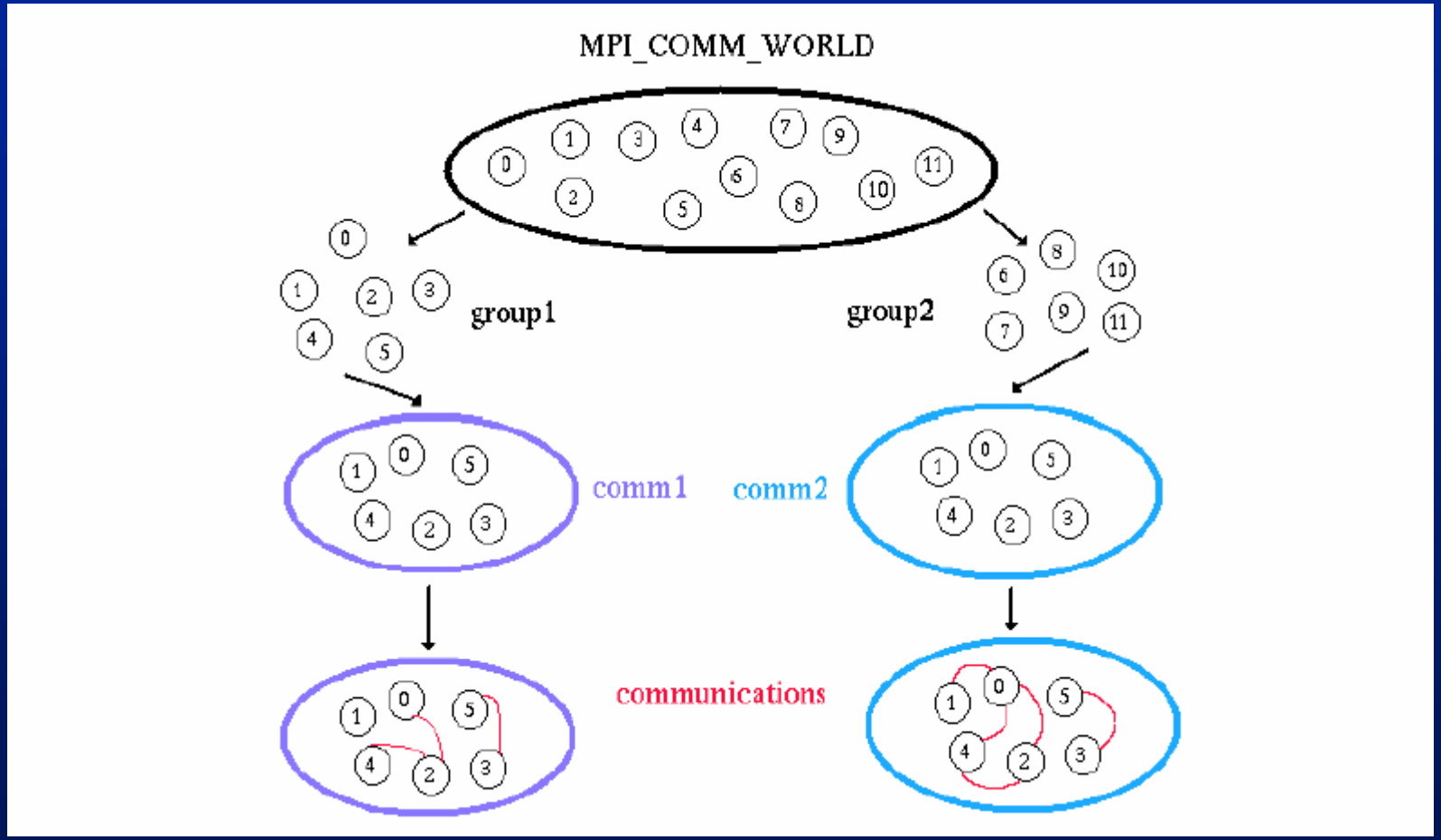
- **MPI_Comm_create**

  (MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

  erzeugt einen neuen Kommunikator mit Prozessgruppe group, die Teilmenge der Prozessgruppe von comm sein muss.

  - **kollektiver Aufruf**
  - **identische group Argumente erforderlich**
  - **Prozesse in comm, aber nicht in group, erhalten MPI_COMM_NULL zurück.**
  - **Synchronisation der Aufrufe nicht erforderlich, aber „Nachrichten aus der Zukunft“ möglich**

Typischer Aufbau neuer Kommunikatoren



Aufteilen von Kommunikatoren

- MPI_Comm_Split

  (MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

  Aufteilung der Prozessgruppe von comm in disjunkte Teilgruppen, je eine pro Wert von color

  => Prozesse mit derselben Farbe sind in derselben Teilgruppe

  Nummerierung innerhalb der Gruppen gemäß key, bei identischen Schlüsseln gemäß Nummerierung in comm

  Aufrufe mit der Farbe MPI_UNDEFINED liefern den Kommunikator MPI_COMM_NULL zurück.

Bsp: MPI_Comm_split (comm, 0, 0, newcomm)

entspricht

MPI_Comm_dup (comm, newcomm)

Virtuelle Topologien

- Unterstützung fester Kommunikationsstrukturen
  - effizientere Abbildung von Prozessstrukturen auf physikalischer Zielmaschine
  - einfachere, adäquatere Benennung von Prozessen
  - Verbesserung der Lesbarkeit von Programmen
- zwei Arten:
  - kartesische Topologien: Gitter, Würfel, Torus, Hypercube
  - Graphtopologien
- feste Assoziation mit Kommunikator
- Erzeugung virtueller Topologie erzeugt neuen Kommunikator

Erzeugung kartesischer Topologie

- **MPI_Cart_create**

  (MPI_Comm comm_old, int ndims, [int] dims, [bool] periods, bool reorder, MPI_Comm *comm_cart)

assoziiert mit comm_old eine kartesische Topologie mit ndims Dimensionen und der Ausdehnung dim[i] in Dimension i.

periods[i] = true bedeutet eine zyklische Verbindung in Dim. i.

reorder = false => Prozessränge bleiben unverändert
reorder = true => Umordnung von Prozessen erlaubt, um Abbildung auf physikalische Topologie zu verbessern.

comm_cart kann weniger Prozesse enthalten als comm_old, einige Prozesse erhalten MPI_COMM_NULL als Resultat.

Nummerierung beginnt immer bei Null, auch in Fortran.

Zugriffsroutinen

- **MPI_Cart_rank** (MPI_Comm comm, int *coords, int *rank)

  liefert in Kommunikator mit kartesischer Topologie den Rang eines Prozesses zu dessen Koordinaten und MPI_PROC_NULL bei ungültigen Koordinaten (nicht bei periodischen Strukturen)

- **MPI_Cart_coords**

  (MPI_Comm comm, int rank, int maxdims, int *coords)

  liefert Koordinaten zu Prozessrang.

  maxdims gibt Dimension des Arrays coords an.

- **MPI_Cart_shift**

  (MPI_Comm comm, int direction, int disp,
  int *rank_source, int *rank_dest)

  liefert Ränge der Partnerprozesse für Shift-Operationen.

  directions bestimmt die Dimension für Shift, disp die Schrittweite.

Bestimmung balancierter Gitterstrukturen

- **MPI_Dims_create (int nnodes, int ndims, int *dims)**

  bestimmt zu nnodes Prozessen eine balancierte kartesische Topologie mit ndims Dimensionen

  Im Feld dims können Felder vorbesetzt werden.

  nnodes muss Vielfaches von dem Produkt aller Werte dims[i] mit dims[i] <> 0 sein.

Interkommunikatoren

- Interkommunikatoren dienen der Kommunikation zwischen disjunkten Prozessgruppen. Sie erlauben nur Punkt-zu-Punkt-Kommunikationen.

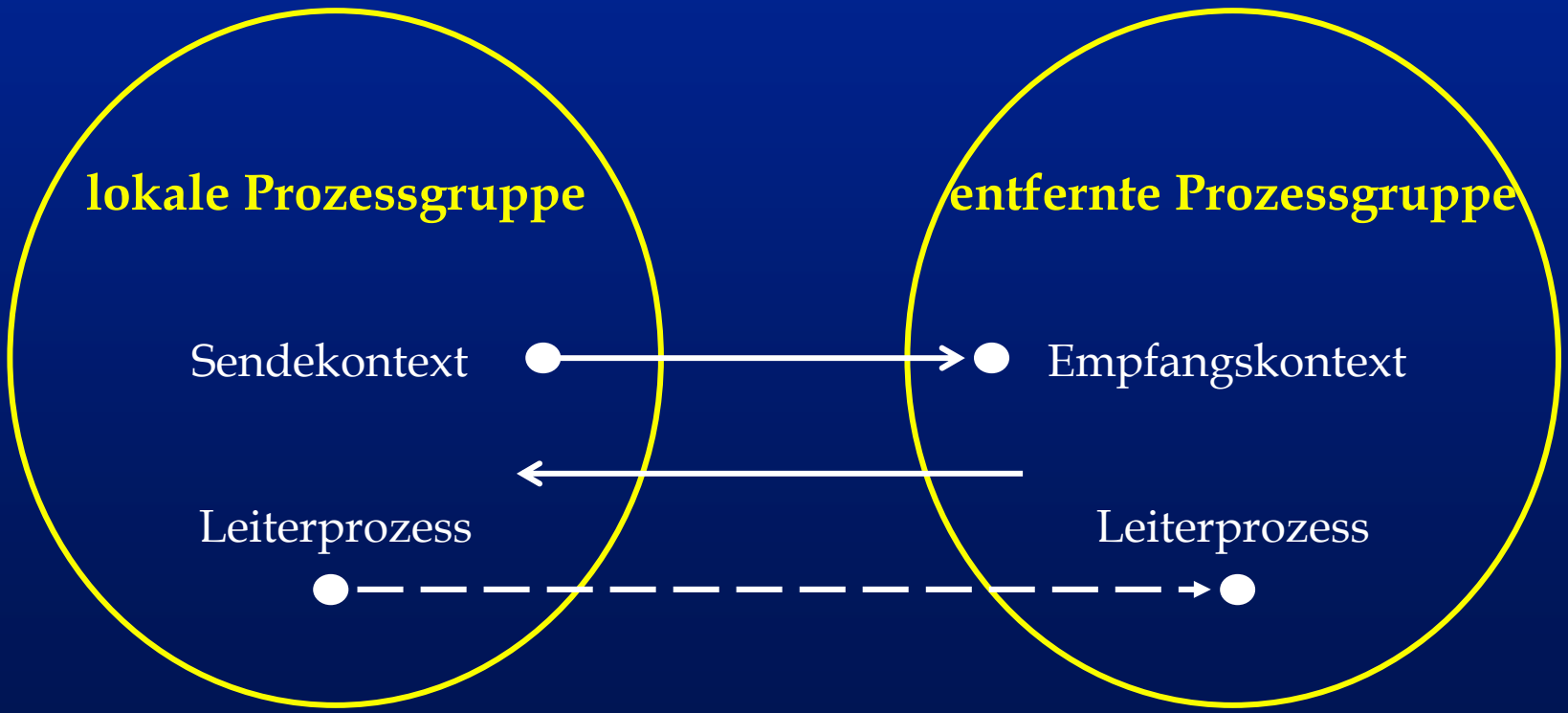
- MPI_Intercomm_create (local_comm, local_leader,
  peer_comm, remote_leader,
  tag, newintercomm)

  muss kollektiv in beiden Gruppen aufgerufen werden.
  Alle Prozesse müssen sich jeweils auf einen Leiter einigen.

  peer-comm ist ein übergeordneter Kommunikator, der beide Gruppen enthalten muss, meist MPI_COMM_WORLD oder ein Duplikat.

Interkommunikation

- Der Aufruf von MPI_Intercomm_create bewirkt
  - kollektive Kommunikationen in den einzelnen Gruppen
  - Punkt-zu-Punkt-Kommunikation zwischen Leiterprozessen in peer_comm mit „tag“



Abgeleitete Datentypen

- Erzeugung in zwei Stufen:
  - A - Spezifikation des Datentyps
  - B - Bereitstellung des Datentyps

- ad A: Eine Typabbildung (type map) ist eine Folge von Paaren der Form

  < (type_i, disp_i | 0 <= i < n >

  wobei type_i Basistypen und

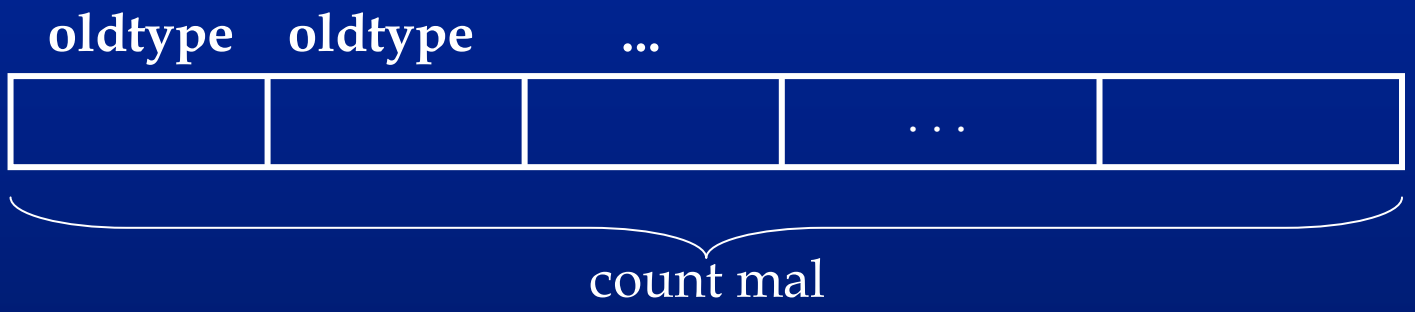
  disp_i ganze Zahlen /relative Adressen (displacements)

  sind.
- Die Folge <type_i | 0 <= i < n> heißt Typsignatur der Typabbildung.
- Zusammen mit einer Basisadresse buf spezifiziert eine Typabbildung einen Kommunikationspuffer mit n Einträgen. Der i-te Eintrag beginnt

  an der Adresse buf+disp_i und hat Typ type_i.

Routinen zur Definition abgeleiteter Typen

- **MPI_Type_contiguous (count, oldtype, newtype)**
  bewirkt count-fache (count > 0) Replikation von oldtype zu newtype.



- **Die Typüberprüfung beim Empfang von Daten berücksichtigt nur die Typsignatur, d.h. die Folge der Basistypen, und nicht die zugrundeliegenden Typdefinitionen.**

Routinen zur Definition abgeleiteter Typen

- **MPI_Type_vector (count, blocklength,stride, oldtype, newtype)**

  blocklength spezifiziert die Blocklänge, stride die Schrittweite.

Bsp: **count = 2, blocklength = 3, stride 5**

oldtype 

newtype

Bereitstellung von Datentypen

- **MPI_Type_commit (datatype)**

  aktiviert neu erzeugten Datentyp, muss vor Verwendung in Kommunikationsroutinen aufgerufen werden.

- **MPI_Type_free (datatype)**

  Freigabe von Datentyp