MPI-Kurs

Inhaltsverzeichnis

1 Einleitung

1.1 Ziel und Sinn von MPI

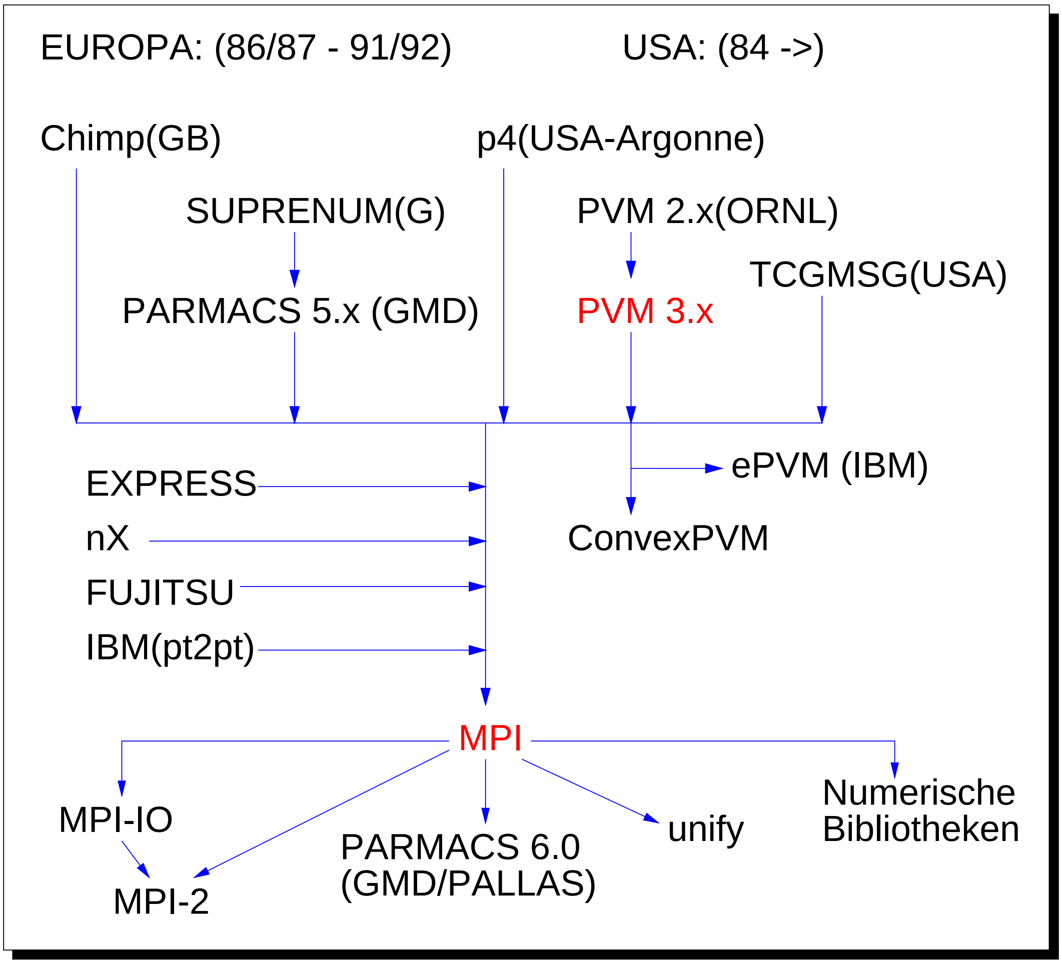
MPI (Message-Passing Interface) ist ein Versuch, das Message-Passing Modell für das parallele Rechnen so zu standardisieren, daß der/die ProgrammiererIn mit einer einheitlichen Schnittstelle arbeitet.

Das erste Treffen fand am 29. April 1992 in Williamsburg, Virginia in den U.S.A. statt. In den ersten Proposals der damals kleinen Gruppe wurde der Schwerpunkt auf die Punkt-zu-Punkt Kommunikation gelegt, erst später wurden weitere „Gesichtspunkte" des Message-Passing Modells eingeschlossen. Im November 1992 wurde beschlossen einen Standard heraus zu bringen. Dieser sollte auf ähnliche Weise entstehen, wie der für High Performance Fortran (HPF): es wurden Unterkomitees für die einzelnen Schwerpunkte gegründet und eine E-Mail-Liste für die allgemeine Diskussion aufgebaut. Das Komitee traf sich alle sechs Wochen für jeweils zwei Tage und „jeder" konnte Mitglied dieses Komitees werden, sofern er nur zweimal hintereinander bei den regelmäßigen Treffen erschien. Am 5. Mai 1994 kam der offizielle Standard von MPI heraus.

Die Ziele von MPI sind folgende:

- Erstellen einer Schnittstelle für die Programmierung von Applikationen,
- Effiziente Kommunikation: Vermeidung von memory-to-memory Kopien, Überlapp von Kommunikation und CPU-Rechnen und Benutzung von Kommunikationsprozessoren,
- Implementation soll in heterogener Umgebung lauffähig sein,
- Bequeme Einbindung von C und FORTRAN 77 als Schnittstelle,
- Sichere Kommunikation, d.h. der/die BenutzerIn muß sich nicht darum kümmern, daß die Kommunikation fehlerfrei verläuft,
- Kein großes Abweichen von bisherigen Message-Passing Systemen, wie PVM, p4, NX, Express, CHIMP, etc., aber mit größerer Flexibilität,
- Die Schnittstelle zu herstellerspezifischen Message-Passing Systemen auf der Ebene der Hardware soll leicht aufzubauen sein,
- Semantik der Schnittstelle soll unabhängig einer Programmiersprache sein,
- MPI soll „thread-sicher" sein.
- MPI soll als Basis für die Entwicklung von Bibliotheken (numerische Bibliotheken, Kommunikationsbibliotheken, u.ä.) dienen.

1.2 Vor- und Nachfahren



Stammbaum von MPI

Auf der Tagung „Supercomputing '94" in den U.S.A., fand ein Treffen statt, auf dem entschieden werden sollte, ob ein MPI-2 entwickelt werden soll.

Mögliche Verbesserung bzw. Erweiterungen sollen sein: Parallel-I/O, Active messages, Process startup, Dynamic process control, Remote store/access, FORTRAN 90 und C++ language bindings, Graphics, Real-time support und „Verschiedenes".

1.3 Implementationen

Mittlerweile existieren vier öffentliche MPI-Implementationen, drei aus den U.S.A. und eine aus Großbrittanien:

MSU UNIFY implementation ist eine Vermischung von PVM- und MPI-Aufrufen, mit dem Ziel beide Arten von Befehlen zur Verfügung zu stellen. Die URL lautet:

ftp://ftp.erc.msstate.edu/unify.

EPCC CHIMP implementation setzt auf das britische Message-Passing System CHIMP auf. Die URL lautet:

ftp://epcc.ed.ac.uk/pub/chimp.

Ohio Supercomputer Center LAM implementation LAM (Local Area Multicomputer) ist eine Programmierumgebung und Entwicklungssytem für parallele Maschinen im Rahmen eines Message-Passing Modells und heterogenen UNIX-Rechnern. LAM beinhaltet eine volle MPI-Implementation und ist kompatibel zu PVM. LAM selber ist Teil eines größeren Projektes: Trollious. Die URL lautet

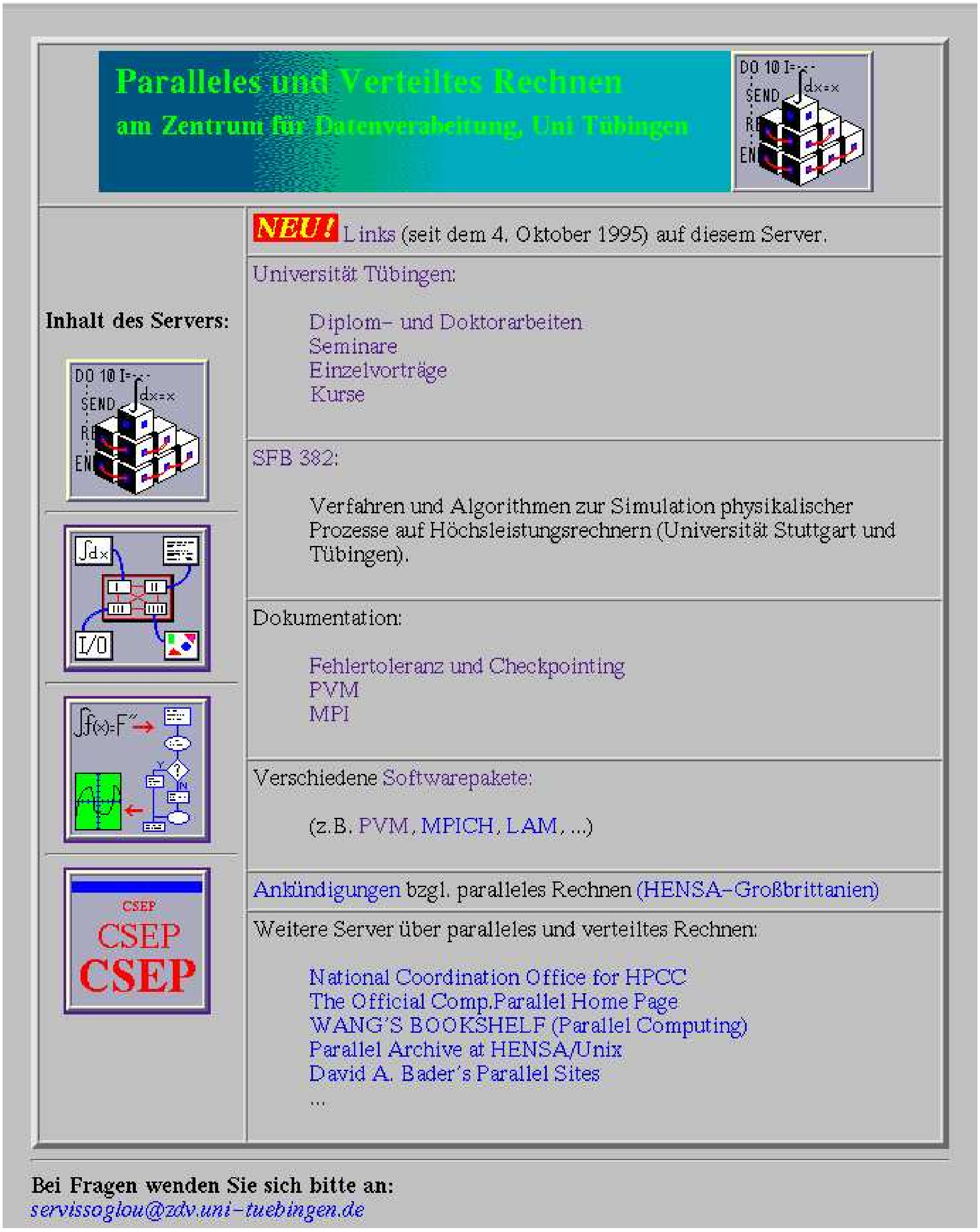
http://www.osc.edu/lam.html oder
ftp://tbag.osc.edu/pub/lam

The ANL portable implementation basiert auf das ADI (Abstract Device Interface). Für Workstations ist p4 oder PVM das ADI, die Schnittstelle zwischen der Hardware und MPI. Die URL lautet

http://www.mcs.anl.gov/mpi/index.html oder
ftp://info.mcs.anl.gov/pub/mpi

„Spiegel“ der obigen Implementationen liegen in Tübingen:

ftp://ftp.uni-tuebingen.de/pub/parallel/ oder
http://alphamor.zdv.uni-tuebingen.de/PARALLEL/



WWW-Seite über paralleles Rechnen (s.o.)

1.4 MPICH am ZDV

Auf dem HP-Pool ist immer die aktuellste MPICH-Implementation installiert. Die ADI-Implementation auf dem HP-Pool basiert auf p4, d.h. Socketverbindungen und ohne Dämonen.

C-Programme, die unter MPI laufen sollen, müssen mit folgenden Flags übersetzt werden:

```
-Aa -D_POSIX_SOURCE -D_HPUX_SOURCE -DFORTRANNOUNDERSCORE
```

MPI-Bibliothek wird wie folgt an den Object-Code „gelinkt“

```
-L/soft/lib/mpich/hpux/ch_p4 -lmpi -lV3
```

Auch FORTRAN 77-Programme lassen sich unter MPI entwickeln, hierzu muß folgende Option für den FORTRAN-Compiler *f77* gesetzt werden:

```
-Wl
```

Die MPI-Bibliothek für die FORTRAN-Programme wird unter den gleichen Bedingungen angebunden wie für C-Programme:

```
-L/soft/lib/mpich/hpux/ch_p4 -lmpi -lV3 -lU77
```

Die zusätzliche Bibliothek (`-lU77`), die mit eingebunden wird, ist die BSD 3F Bibliothek. Diese bietet eine Schnittstelle zu einigen Systemroutinen der *libc*-Bibliothek.

Programme unter MPI werden mit Hilfe des Programms

mpirun `[Optionen] <Progname> [-p4pg hostfile]`

gestartet. Hier einige Optionen:

- **-h(elp)**
  Auflistung aller Optionen,
- **-arch <architecture>**
  Angabe der lokalen Archtektur, sollte für WS-Pools immer angegeben werden (HP-UX),
- **-dbx**
  Der erste Prozeß wird unter dem Debugger *dbx* gestartet,
- **-machine <machine name>**
  Startup-Prozedur für spezielle Architektur:
  `chameleon, p4, cray_t3d, ksr, sgi`,
- **-np <np>**
  Anzahl von Prozessoren, auf dem das Programm laufen soll,
- **-p4pg <hostfile>**
  Liste mit Maschinen, auf denen die Applikation laufen soll.

Soll die Applikation unter MPI auf mehreren Workstations laufen, so gibt es zwei Möglichkeiten:

1. Im Verzeichnis *../mpich/util/machines* ein File anlegen, das folgenden Namen hat: *machines.*`xxxx` und die Rechner der Architektur `xxxx` eintragen. Diese Liste wird von oben nach unten abgearbeitet (ohne spezielles Load-Balancing).

2. Erstellung einer individuellen Liste von Maschinen, auf denen die Applikation laufen soll (der Inhalt einer Zeile):

   ```
   <hostname> <#procs> <progname> [<login>]
   ```

2 Vorbemerkungen-Erklärungen

Fünf Begriffe aus der Semantik von MPI sollen hier kurz erläutert werden:

- *non-blocking*, der Aufruf kehrt vor der Vollendung zurück, erlaubt aber dem/der BenutzerIn nicht die benutzten Ressourcen der Prozedur zu benutzen.
- *blocking*, Aufruf kehrt zurück und der/die BenutzerIn kann die entsprechenden Ressourcen verwenden.
- *local*, die Beendigung des Prozesses hängt nur vom lokalen Prozeß ab, es ist keine Kommunikation mit anderen Benutzerprozessen nötig.
- *non-local*, Beendigungsbedingung verlangt Ausführung eines MPI-Befehles in einem anderen Prozeß. Dieser Aufruf kann Kommunikation mit einem anderen Benutzerprozeß benötigen.
- *collective*, wenn alle Prozesse in einer Gruppe diesen Aufruf tätigen müssen.

Zusätzlich zu den „bekannten" Datentypen oder Objekten hat MPI einen weiteren eingeführt: *opaque object* (ähnlich dem POSIX-Fortran Binding Standard).

```
+--------------------+                                   +------------+
|   SYSTEM MEMORY    |                                   |USER SPACE  |
+====================+                                   +============+
|Nachrichten puffern| <--- opaque object <-----+  handles   |
+--------------------+                                   +------------+
|Speicherung der in-|                                    |MPI-Aufrufe|
|ternen Dastellung  |                                    |            |
+--------------------+                                   |Zuweisungen|
|Gruppen, Kommunika-|<---- User hat keine ---->|            |
|toren, Datentypen  |      Verantwortung                 |Vergleiche  |
+--------------------+                                   +------------+
```

Bemerkung: *opaque*-Objekte sind nur im eigenen Prozeß gültig und lassen sich nicht transferieren!

Bei der Allokation eines *opaque*-Objektes ist dieses vom Typ OUT, bei der Deallokation vom Typ INOUT, desweiteren findet eine Deallokation erst dann statt, wenn alle darauf zugreifenden Operationen beendet sind. Vorsicht sollte bei einer Deallokierung gelten: ist diese einmal aufgerufen, läßt sich nicht mehr auf das dazu passende Objekt zugreifen. Beim Kopieren von „handles" sollte auf eine entsprechende Deallokierung geachtet werden!

MPI macht keine Bemerkungen zur intersprachlichen Programmierung, d.h. der Standard bezieht sich nicht auf die Schnittstellen zwischen zweier Programmiersprachen.

FORTRAN 77:
MPI_<Großbuchstaben>(..., ...,...,<return-code>)
Die Namensgebung benutzt den inoffiziellen Standard von 30 Buchstaben in denen auch das Zeichen „underscore“ auftauchen kann. Die Felder beginnen mit 1 und die „handles“ sind vom Typ `INTEGER`.

ANSI C:
MPI_<Großbuchstabe><Buchstaben>(..., ..., ..., ...)
Die Felder beginnen mit 0 und die „handles“ sind je nach Aufruf vom Typ `int` oder Zeiger.

MPI wünscht, daß Betriebssysteme, die Threads benutzen, diese parallel MPI-Aufrufe tätigen können. Ruft ein Thread eine blockierende MPI-Funktion auf, darf diese nicht die dazu parallelen Threads beeinflussen oder gar blockieren.

3 Punkt-zu-Punkt Kommunikation

3.1 Allgemeine Eigenschaften des Datentransfers

MPI kennt vier Modelle des Versendens, die sich auf den blockierenden als auch nicht blockierenden Modus beziehen:

standard: MPI entscheidet, ob Daten zum Senden gepuffert werden sollen oder nicht (nicht lokal).

- Datenpufferung: Sende-Befehle sind vollendet, wenn Pufferung vollzogen ist.
- Keine Datenpufferung: Sende-Befehle sind fertig, wenn passender Empfangs-Befehl entstanden ist.

buffered: MPI sorgt für Pufferung der zu versendeten Daten (lokal).

synchronous: Die Sende-Befehle sind erst dann abgeschlossen, wenn der entsprechende Empfang der Daten begonnen hat. D.h. der Sendepuffer kann danach gleich wiederverwendet werden (ähnlich wie oben), mit der „Gewißheit", daß die Daten anfangen anzukommen. Findet auf beiden Seiten eine Blockierung statt, so erreicht man eine Synchronisation.

ready: Die Sende-Befehle können nur dann gestartet werden, wenn vorher ein dazu passender Empfangs-Befehl gestartet wurde.

Alle vier Modi des Versendens einer Nachricht können mit einem Empfangs-Befehl befriedigt werden.

MPI verlangt von jeder Implementation folgende Eigenschaften (d.h. diese müssen vom Hersteller gewährleistet werden):

Reihenfolge: Aufeinanderfolgende Versendungen (Nachrichtenkennung und Empfänger sind gleich) werden in der gleichen Reihenfolge empfangen. Dies wird bei Betriebssystemen mit Threads nicht gefordert.

Progress: Bei zusammenpassendem Paar (Versand/Empfang) ist gewährleistet, daß mindestens eine Operation beendet wird (Vermeidung von Deadlocks).

Fairness: Diese ist nicht gewährleistet, konkurrierende Prozesse können andere Kommunikationsprozesse überlagern.

Ressourcen: Systempuffer wird im Puffer-Modus benötigt, evtl. auch beim Standard:

- Wiederholung des Puffer-Modus kann zum Überlaufen des Puffers führen, hierauf sollte mit einer Fehlermeldung reagiert werden.
- Beim Standard-Modus kann kein Puffer überlaufen, aber ein Blockieren stattfinden, bis genug Puffer bereitgestellt wird oder der entsprechende Empfänger bereit ist.

Diese Eigenschaften müssen sowohl die blockierenden, als auch die nichtblockierenden Befehle einer MPI-Implementation erfüllen.

3.2 Blockierende Kommunikation

```
MPI_SEND(
        buf,                                  IN
        count,                                IN
        dtype,                                IN
        dest,                                 IN
        tag,                                  IN
        comm                                  IN
        )
```

```
MPI_RECV(
        buf,                                  OUT
        count,                                IN
        dtype,                                IN
        source,                               IN
        tag,                                  IN
        comm,                                 IN
        status                                OUT
        )
```

Das Argument comm gibt das *Universum* an, in dem der Transfer der Nachricht stattfinden soll. Ein Universum heißt MPI_COMM_WORLD, dies ist das Gebiet in dem alle Prozesse sich nach dem Start unter MPI befinden. Das ist das größte gemeinsame Gebiet. In einem Universum hat jeder Prozeß eine Rangnummer, von 0 bis n-1.

Das Argument status enthält Informationen über die Quelle, der empfangenen Nachricht. In ANSI C ist es eine Struktur, in FORTRAN 77 ein Integer-Feld der Länge MPI_STATUS_SIZE.

```
MPI_GET_COUNT(
        status,                     IN
        datatype,                   IN
        count                       OUT
        )
```

Der obige Aufruf gibt die Länge der empfangenen Nachricht zurück.

Vorsicht: Das Argument `count` gibt dabei die Anzahl der Elemente und nicht der Bytes an!

Bemerkung: Ein Typematching findet unter drei Bedingungen statt:

- Die Kommunikation beider Partner findet mit dem gleichen Datentyp statt,
- die Kommunikation mindestens einer der Partner findet mit dem Datentyp MPI_BYTE statt oder
- mit dem Datentyp MPI_PACKED.

MPI führt keine Typkonvertierung aus, bei heterogenen Architekturen aber eine Darstellungskonversion (bezieht sich auch auf die message envelope). Die Darstellungskonversion entfällt beim Datentyp MPI_BYTE.

```
MPI_BSEND( buf, count, datatype, dest, tag, comm )
MPI_SSEND( buf, count, datatype, dest, tag, comm )
MPI_RSEND( buf, count, datatype, dest, tag, comm )
```

Die obigen Befehle dienen den drei weiteren Versendungsmodi (mit Beispiel):

```
#include "mpi.h"
main( int argc, char **argv )
{
    char       message[20];
    int        myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if ( myrank == 0 ){
        strcpy( message, "Hallo, wohin geht es?" );
        MPI_Send( message, strlen( message ), MPI_CHAR, 1, 99,
                  MPI_COMM_WORLD );
    }
    else{
        MPI_Recv( message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD,
                  &status );
        printf( "Angekommen :%s:\n", message );
    }
    MPI_Finalize();
}
```

```
MPI_BUFFER_ATTACH(
        buffer,                                  IN
        size                                     IN
        )
```

```
MPI_BUFFER_DETACH(
        buffer_addr,                             OUT
        size                                     OUT
        )
```

Bemerkung: Zu dem Puffer-Modus des Sendens hat der Benutzer die Möglichkeit sich selber einen Puffer zu allokieren. Ausschließlich dieser Puffer wird nach der Allokation für das obige Senden benutzt, d.h. der Benutzer hat darauf zu achten, daß er groß genug ist!

Der erste Aufruf allokiert einen `size` Bytes großen Puffer im Speicher, während der zweite Aufruf diesen Speicher deallokiert. Dieser wird erst dann dem System zurückgegeben, wenn alle anstehenden Transferoperationen, die sich auf diesen Puffer beziehen, abgeschlossen sind. Aufbau des Puffers:

```
          +---------------------------+
          |                           |
+---+---+------------------------+-V----------
| h | p |                        | h |
| a | o |                        | a |
| n | i |                        | n |
| d | n |   data ...........     | :
| l | t |                        | :
| e | e |                        |
|   | r |                        |
+---+---+------------------------+------------
```

Vorsicht: Jeder Prozeß kann nur einen Puffer allokieren!

3.3 Nicht blockierende Kommunikation

Diese Art der Kommunikation ermöglicht der Applikation das parallele Ausführen des Datentransfer und ein Weiterrechnen im Programm. Dies hat zur Folge, daß die Applikation mindestens zwei Befehlsaufrufe benötigt um den Transfer ordnungsgemäß zu beenden:

Mit einem *start call* wird die Kommunikation angestoßen. Dieser Aufruf ist lokal und gibt die Kontrolle sofort wieder der Applikation zurück.

Der *complete call* versucht die Kommunikation zu beenden. Dieser Aufruf wartet solange (blockiert) bis ein Fehler auftaucht oder die Kommunikation erfolgreich war. Dieser Aufruf gehört zur Gruppe der Verifikation, d.h. er schaut nach, ob die Kommunikation geregelt verläuft.

Das nicht blockierende Versenden unterstützt alle vier Versendungsmodi (Standard, Puffer, Synchron und Ready). Ein unterschiedliches Verhalten ist aber gegeben:

- *Standard*: *send complete call* kehrt nur dann zurück, wenn Puffer vorhanden ist oder der entsprechende Empfang stattgefunden hat.
- *Synchron*: *send complete call* ist nicht, wie bei den anderen Versendungsmodi lokal, sondern nicht lokal, denn eine Rückkehr findet nur bei entsprechendem Empfang statt.
- *ready*: *send start call* ist nur dann möglich, wenn ein passender Empfang schon vorhanden ist.

Bemerkung: nicht blockierende Versendungen können mit blockierenden Empfängen kombiniert werden und umgekehrt.

Die Routinen benutzen *opaque request*-Objekte über die mit einem *handle* zugegriffen werden kann. Diese Objekte dienen zur Identifikation der jeweiligen Kommunikation: Sendemodus, Kommunikationspuffer, Inhalt, *message envelope* und Kommunikationsstatus.

```
MPI_ISEND/_IBSEND/_ISSEND/_IRSEND(
        buf,                                   IN
        count,                                 IN
        datatype,                              IN
        dest,                                  IN
        tag,                                   IN
        comm,                                  IN
        request                                OUT
        )
```

```
MPI_IRECV(
        buf,                                   OUT
        count,                                 IN
        datatype,                              IN
        source,                                IN
        tag,                                   IN
        comm,                                  IN
        request                                OUT
        )
```

Die obigen fünf Befehle stossen das Senden für alle vier Modi (I=immediate), bzw. den Empfang an.

```
MPI_WAIT(
         request,                              INOUT
         status                                OUT
         )
```

```
MPI_TEST(
         request,                              INOUT
         flag,                                 OUT
         status                                OUT
         )
```

Diese Aufrufe beenden korrekt die Kommunikation. Der erste Aufruf blockiert, da eine Rückkehr nur dann erfolgt, wenn der Transferanstoß erfolgreich war. Nach der Rückkehr hat `request` den Wert MPI_REQUEST_NULL und das zugehörige Objekt ist deallokiert.

`MPI_TEST` ist lokal, da dieser Aufruf testet, ob der Kommunikationsanstoß vollendet ist. Ist der Anstoß fertig, so hat `flag` den Wert true (Deallokation des Objektes (s.o.) und das Objekt `status` hat eine genauere Beschreibung der Nachricht), sonst den Wert false.

```
MPI_WAITANY(
        count,                    IN
        array_of_reqs,         INOUT
        index,                   OUT
        status                   OUT
        )
```

```
MPI_TESTANY( count, array_of_reqs, index, flag, status )
MPI_WAITALL( count, array_of_reqs, array_of_statuses )
MPI_TESTALL( count, array_of_reqs, flag, array_of_status )
```

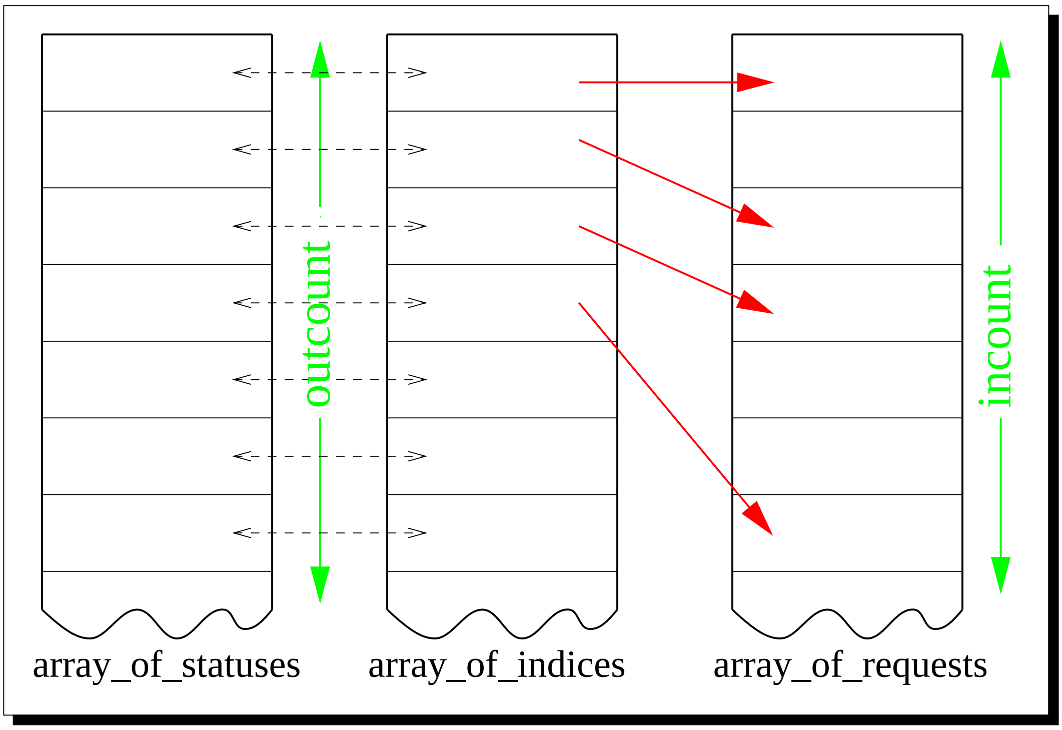
MPI bietet unter der Überschrift der gemeinschaftlichen Komplettierungen die obigen vier Aufrufe an. Die `...ANY`-Aufrufe warten bzw. testen bis mindestens einer der nicht blockierenden Transfers in der Liste `array_of_reqs` fertig ist. `index` enthält die Listennummer des Transfers, der fertig ist. Werden mehrere gleichzeitig beendet, so entscheidet der „Zufall" welcher `index` gespeichert wird.

Die zwei `...ALL`-Aufrufe warten bzw. testen bis alle in der Liste stehenden nicht blockierenden Transfers fertig sind. In der Liste `array_of_status` stehen die einzelnen Status der Transfers.

Für die Bedeutung des Arguments `flag` siehe weiter vorn.

```
MPI_WAITSOME/_TESTSOME(
          incount,                          IN
          array_of_request,                 INOUT
          outcount,                         OUT
          array_of_indices,                 OUT
          array_of_statuses                 OUT
          )
```

Diese zwei Funktionen arbeiten ähnlich den ...`ANY`-Funktionen, mit dem Unterschied, daß die ...`SOME`-Funktionen alle beendeten Transferfunktionen, die in der Liste `array_of_requests` stehen, notieren und deren Status in `array_of_statuses` speichern:



Darstellung der Listen in `MPI_...SOME`

Client-Server-Konzept (mit und ohne Gefahr der „Austrocknung“):

```
      CALL MPI_COMM_SIZE( comm, size, ierr )
      CALL MPI_COMM_RANK( comm, rank, ierr )
      IF ( rank > 0 ) THEN     ! client Code
        DO WHILE ( .TRUE. )
          CALL MPI_ISEND( a, n, MPI_REAL, 0, tag, comm, request, ierr )
          CALL MPI_WAIT( request, status, ierr )
        END DO
      ELSE                     ! Rang=0 -- Server Code
        DO i = 1, size-1
          CALL MPI_IRECV( a( 1, i ), n, MPI_REAL, i, tag, comm,
     &                    request_list( i ), ierr )
        END DO
        DO WHILE ( .TRUE. )
          CALL MPI_WAITANY( size-1, request_list, index, status, ierr )
          CALL DO_SERVICE( a( 1, index ) )  ! handle one message
          CALL MPI_IRECV( a( 1, index ), n, MPI_REAL, index, tag, comm,
     &                    request_list( index ), ierr )
        END DO
      END IF
```

==

```
      CALL MPI_COMM_SIZE( comm, size, ierr )
      CALL MPI_COMM_RANK( comm, rang, ierr )
      IF ( rank > 0 ) THEN     ! client code
        DO WHILE ( .TRUE. )
          CALL MPI_ISEND( a, n, MPI_REAL, 0, tag, comm, request, ierr )
          CALL MPI_WAIT( request, status, ierr )
        END DO
      ELSE                     ! Rang=0 -- Server Code
        DO i = 1, size-1
          CALL MPI_IRECV( a( 1, i ), n, MPI_REAL, i, tag, comm,
     &                    request_list( i ), ierr )
        END DO
        DO WHILE ( .TRUE. )
          CALL MPI_WAITSOME( size-1, request_list, numdone, index_list,
     &                       status_list, ierr )
          DO i = 1, numdone
            CALL DO_SERVICE( a( 1, index_list( i ) ) )
            CALL MPI_IRECV( a( 1, index_list( i ) ), n, MPI_REAL,
     &                      index_list( i ), tag, comm,
     &                      request_list( i ), ierr )
          END DO
        END DO
      END IF
```

```
MPI_SEND_INIT/_BSEND_INIT/_SSEND_INIT/_RSEND_INIT(
          buf,                                      IN
          count,                                    IN
          dtype,                                    IN
          dest,                                     IN
          tag,                                      IN
          comm,                                     IN
          request                                   OUT
          )
```

```
MPI_RECV_INIT(
          buf,                                      OUT
          count,                                    IN
          dtype,                                    IN
          source,                                   IN
          tag,                                      IN
          comm,                                     IN
          request                                   OUT
          )
```

Möchte der/die AnwenderIn in seinem/ihrem Programm die gleichen Daten an den gleichen Empfänger verschicken, kann ein dauerhaftes request-Objekt erzeugt werden, um den Overhead zwischen Applikation und Kommunikationskontrolle (und nur diesen!) zu verringern. Es wird etwas ähnliches, wie ein halber Kanal bereitgestellt. Der Gegenpart der Kommunikation muß kein dauerhaftes request-Objekt bereitstellen.

```
MPI_START(
        request                                         INOUT
        )
```

```
MPI_START_ALL(
        count,                                          IN
        array_of_requests                               INOUT
        )
```

Bei der Erstellung des dauerhaften `requests` bleibt dieser solange inaktiv, bis der obige Aufruf getätigt wird.

Beide Befehle initiieren den/die `request(s)` in der Liste `array_of_requests` und aktivieren ihn/sie (Aufruf ist lokal). Ähnlich wie die nicht blockierenden Aufrufe, brauchen auch diese eine Komplettierung.

```
MPI_REQUEST_FREE(
        request                                   INOUT
        )
```

Der obige Aufruf deallokiert ein persistentes request-Objekt. Hiermit fordert man MPI auf, das request-Objekt nach Beendigung der Kommunikation zu deallokieren.

Bemerkung: `MPI_REQUEST_FREE` kann auch in anderen Zusammenhängen aufgerufen werden (z.B. nach einem nicht blockierenden Sende-Befehl).

Vorsicht: Ist einmal ein request-Objekt deallokiert, ist dieses nicht mehr wiederzufinden und somit auch nicht mehr nachzuprüfen, ob die damit verbundene Kommunikation erfolgreich war!

```
MPI_SENDRECV(
         sendbuf,                                  IN
         sendcount,                                IN
         sendtype,                                 IN
         dest,                                     IN
         sendtag,                                  IN
         recvbuf,                                  OUT
         recvcount,                                IN
         recvtype,                                 IN
         recvtype,                                 IN
         source,                                   IN
         recvtag,                                  IN
         comm,                                     IN
         status                                    OUT
         )
```

```
MPI_SENDRECV_REPLACE(
         buf,                                      INOUT
         count,                                    IN
         dtype,                                    IN
         dest,                                     IN
         sendtag,                                  IN
         source,                                   IN
         recvtag,                                  IN
         comm,                                     IN
         status                                    OUT
         )
```

Die beiden Aufrufe ermöglichen das Senden und Empfangen (in dieser Reihenfolge) mit einem Befehl. Beide Befehle blockieren. Der erste braucht für das Senden und Empfangen verschiedene Puffer, der zweite Befehl speichert die empfangenen Daten in dem Sendepuffer, dessen Daten vorher verschickt worden sind. Diese Funktionen eignen sich um Daten weiterzureichen. Der untere Aufruf kann nur mit einem Datentyp arbeiten.

MPI bietet in diesem Zusammenhang (gilt auch für beliebig andere Kommunikationsbefehle) den Dummy-Rang MPI_PROC_NULL an, dieser kann für Grenzwerte oder an den „Rändern“ nichtzirkularer Verschiebungen verwendet werden.

```
MPI_IPROBE(
        source,                IN
        tag,                   IN
        comm,                  IN
        flag,                  OUT
        status                 OUT
        )
```

```
MPI_PROBE(
        source,                IN
        tag,                   IN
        comm,                  IN
        status                 OUT
        )
```

Diese Befehle erlauben dem Programm einen möglichen Empfang zu überprüfen. Der erste Befehl blockiert nicht, während der zweite so lange blockiert, bis eine entsprechende Nachricht angekommen ist. Sind Daten angekommen, müssen sie explizit empfangen werden (z.B. mit `MPI_RECV`). Diese Befehle eignen sich für die Fälle, in denen der Empfänger nicht sicher ist, welche Art von Daten ankommen.

Bemerkung: In diesen, wie auch in anderen Aufrufen, können die Argumente `tag` und `source` Wildcard-Werte annehmen (MPI_ANY_TAG und MPI_ANY_SOURCE).

```
MPI_CANCEL(
        request                                   IN
        )
```

```
MPI_TEST_CANCELLED(
        status,                                   IN
        flag                                      OUT
        )
```

Die obigen Aufrufe sperren einen Transferbefehl und sind lokal. Der erste Aufruf markiert eine Übertragung (Senden/Empfangen) als gesperrt. Der *request-handle* muß anschließend mit einer Komplettierung deallokiert werden, damit er für eine spätere Benutzung zur Verfügung steht. Dieser Befehl garantiert nicht die Sperrung: entweder Sperrung oder Ausführung des Transfers, aber nicht beides.

Mit dem letzten Aufruf erfährt der/die ProgrammiererIn, ob die Kommunikation, assoziert mit `status`, gesperrt wurde.

3.4 Abgeleitete Datentypen

Bisher wurden Daten transferiert, die kontinuierlich und von einem Typ waren. Manchmal ist es erwünscht Daten verschiedenen Typs oder nichtkontinuierliche Daten (z.B. Unterblöcke von Matrizen) zu senden. MPI versucht hier entgegenzukommen, d.h. der/die BenutzerIn muß nicht nichtkontinuierliche in kontinuierliche Daten verwandeln und dann verschicken, stattdessen wird diese „undankbare“ Aufgabe von MPI übernommen. Der Implementation ist es überlassen, ob sie den obigen Weg geht.

Abgeleitete Datentypen bestehen aus Basisdatentypen und können rekursiv definiert sein. Der/Die ProgrammiererIn definiert mit Hilfe von MPI-Funktionen die Darstellung des Datentyps.

Die Informationen über einen allgemeinen Datentypen werden in einem *opaque*-Objekt spezifiziert, das aus zwei Teilen besteht:

```
Sequenz von Basistypen       Sequenz von Ganzzahl(Byte)
    |          |                  |     Verschiebungen
    |          +------------------+
    |    type map = {(type_0,disp_0), ..., (type_n-1,disp_n-1)}
    |
type signature des allgemeinen Datentyps
        typesig = {type_0, type_1, ..., type_n-1}
```

type map und Adresse des Puffers bilden den Kommunikationspuffer. Dieser besteht aus n Einträgen: der i-te Eintrag ist an der Stelle `buf+disp_i` und ist vom Typ `type_i`, dabei hat die zu versendete Nachricht n Werte der Datentypen, die in *typesig* definiert sind.

Bemerkung: Die Verschiebungen müssen nicht positiv, verschieden oder in aufsteigender Reihenfolge sein, d.h. die Reihenfolge der Eintragungen kann von der der Speicherung verschieden sein.

Allgemeine Datentypen können in jedem Sende- und Empfangs-Befehl verwendet werden: die Adresse `buf` und `datatype` bilden ein *handle* für den allgemeinen Datentypen.

Die meisten Rechner speichern Daten des verschiedenen Typs in feste Einheiten von Blöcken ab, z.B. Fließkommazahlen der doppelten Genauigkeit in acht Bytes, in diesem Zusammenhang hat MPI die Größe *extend* definiert:

$$
\begin{aligned}
\text{lb(}\texttt{typemap}\text{)} &= \text{min}_j\ \texttt{disp}_j \\
\text{ub(}\texttt{typemap}\text{)} &= \text{max}_j\ (\ \texttt{disp}_j\ \texttt{+ sizeof(type}_j\texttt{)}\ ) \\
\text{extend(}\texttt{typemap}\text{)} &= \text{ub(}\texttt{typemap}\text{) - lb(}\texttt{typemap}\text{)} + \epsilon
\end{aligned}
$$

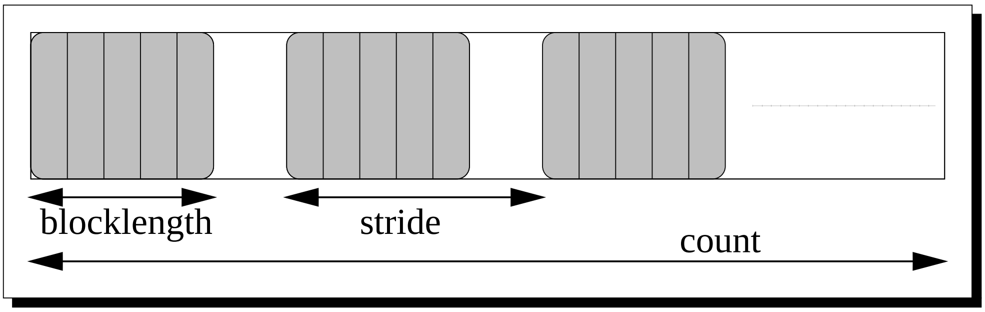
Dieses ϵ dient zur Addierung einer positiven Anzahl von Bytes zur mehrfachen Darstellung einer Größe (z.B. 64 Bit).

```
MPI_TYPE_CONTIGUOUS(
        count,                          IN
        oldtype,                        IN
        newtype                         OUT
        )
```

Der neue Datentyp ist eine kontinuierliche (aneinandergereihte) Kopie von `count` Daten vom Datentyp `oldtype`.

```
MPI_TYPE_VECTOR/_HVECTOR(
          count,                        IN
          blocklen,                     IN
          stride,                       IN
          oldtype,                      IN
          newtype                       OUT
          )
```

Das ist ein etwas allgemeinerer Konstrukteur, der `count` Blöcke der Länge `blocklen` (Anzahl der Elemente vom Typ `oldtype`), in aufeinanderfolgender Schrittweite `stride` konstruiert. Im ersten Befehl bezieht sich `stride` auf die Anzahl der Elemente des Typs `oldtype`, im zweiten Befehl auf die Anzahl der Bytes.



Funktionsweise der Befehle `...VECTOR`

Folgende Konfiguration geht ineinander über:

```
MPI_TYPE_CONTIGUOUS( count, old, new )
             = MPI_TYPE_VECTOR( count, 1, 1, old, new )
             = MPI_TYPE_VECTOR( 1, count, n, old, new )
```

mit n beliebig.

```
MPI_TYPE_INDEXED/_HINDEXED(
          count,                                IN
          array_of_blocklengths,                IN
          array_of_displacements,               IN
          oldtype,                              IN
          newtype                              OUT
          )
```

Diese Befehle unterscheiden sich von den vorherigen zwei Befehlen dadurch, daß die `count` Blöcke variable Blocklänge besitzen und die Blöcke verschiedene Abstände voneinander haben können. Wieder bezieht sich im ersten Aufruf die Verschiebung auf die Anzahl der Elemente des Typen `oldtype`, während der zweite Aufruf sich auf die Anzahl der Bytes bezieht.

```
MPI_TYPE_STRUCT(
          count,                                    IN
          array_of_blocklenghts,                    IN
          array_of_displacements,                   IN
          array_of_types,                           IN
          newtype                                   OUT
          )
```

Der obige Befehl ist der allgemeinste Konstrukeur von Datentypen. Dieser erlaubt nicht nur die Variation der Blocklänge und -abstände, sondern auch die der Datentypen in den jeweiligen Blöcken:

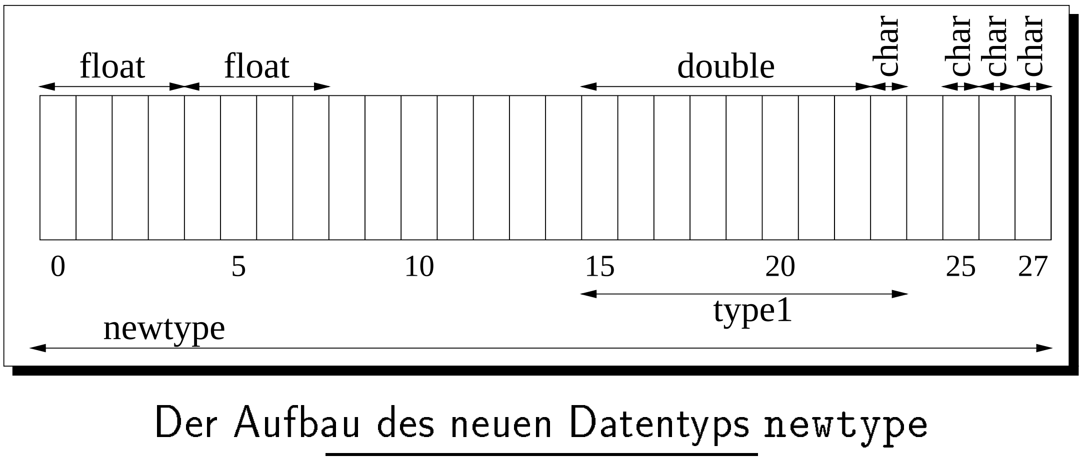
`type1={(double, 0), (char, 8)} und T=(MPI_FLOAT, type1, MPI_CHAR)`

Dann lautet die *type map* des neuen Typs `newtype` nach dem folgenden Aufruf

`MPI_TYPE_STRUCT( 3, (2, 1, 3), (0, 16, 26), T, newtype )`

wie folgt:

`{(float,0),(float,4),(double,16),(char,24),(char,26),(char,27),(char,28)}`



Der Aufbau des neuen Datentyps `newtype`

Voraussetzung: MPI_CONST hat vier Bytes und MPI_CHAR hat ein Byte.

```
MPI_ADDRESS(
        location,                        IN
        address                          OUT
        )
```

```
MPI_TYPE_EXTEND/_SIZE/_COUNT(
        datatype,                        IN
        extend/size/count                OUT
        )
```

Alle Adressen der allgemeinen Datentypen sind relativ zur Adresse des Arguments `buf`, können aber mit Hilfe von MPI auch absolut gesetzt werden. Bezugspunkt ist der Anfang des Adreßraumes: MPI_BOTTOM. Diese Vorgehensweise kann Kopieroperationen vor dem Versenden und nach dem Empfangen ersparen und somit die Performance steigern.

Um diese Art des Datentransfers zu ermöglichen, bietet MPI die obigen vier Informationsfunktionen an. Der erste Aufruf ermittelt die absolute Adresse (`address`) der Variable im laufenden Programm, die sich am Ort `location` befindet. Im Gegensatz zu ANSI C liefert diese Funktion immer die absolute Adresse, in ANSI C wird dies nicht verlangt.

Die zweite Funktion liefert die maximale Ausdehnung `extend` des Datentyps `datatype`. Die dritte, die „tatsächliche" Größe `size` des Datentyps (in Bytes) und die letzte Funktion gibt die Anzahl (`count`) der Basisdatentypen zurück.

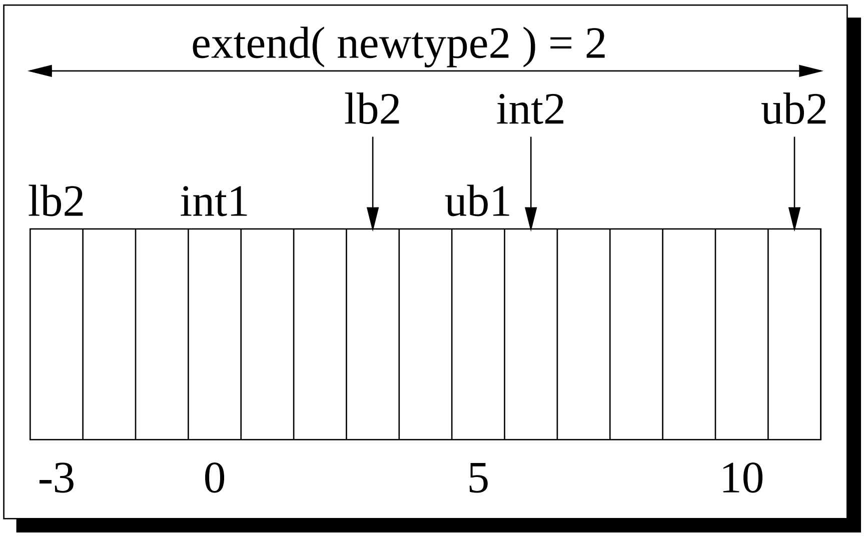
Vorsicht: MPI erkennt keine fehlerhaften Verschiebungen (z.B. außerhalb des gültigen Adreßraums)!

MPI bietet zu den Basisdatentypen zwei Pseudo-Datentypen an: MPI_LB und MPI_UB, beide mit *extend*=0. Diese Pseudo-Datentypen dienen zur Konstruktion von Datentypen mit neuen Grenzen bzw. Ausdehnungen (der Inhalt und die Reihenfolge der Daten wird dabei nicht beeinflußt). Folgendes soll als Beispiel dienen:

```
D=(-3, 0, 6)
T=(MPI_LB, MPI_INT, MPI_UB)
B=(1, 1, 1)
----> MPI_TYPE_STRUCT( 3, B, D, T, newtype1 )
----> extend( newtype1 ) = 9
[von -3 bis einschl. 5], denn type map = {(lb,-3),(int,0),(ub,6)}
```

Anwendung dieser Definition in dem folgenden Befehl liefert

```
MPI_TYPE_CONTIGUOUS( 2, newtype1, newtype2 )
type map (newtype2) = {(lb,-3),(int,0),(int,9),(ub,15)}
```



Schematische Darstellung der Größe `extend`

```
MPI_TYPE_LB/_UB(
        datatype,                          IN
        displacement                       OUT
        )
```

Passend zu den zwei Pseudo-Datentypen bietet MPI zwei Befehle an. Beide Befehle geben die Verschiebung in Bytes relativ zum „Original" an.

```
MPI_TYPE_COMMIT(
          datatype                                      IN
          )
```

MPI setzt eine Hürde vor der ersten Benutzung des neuen Datentyps in einem Kommunikationsbefehl. Zwar kann dieser für eine Definition eines weiteren neuen Datentyps genutzt werden, aber nicht für die Kommunikation. Hierfür ist es erforderlich den neuen Datentypen anzumelden bzw. von MPI anerkennen zu lassen. Dies geschieht mit dem obigen Befehl. Dabei ist `datatype` der neu anzuerkennende Datentyp. Basisdatentypen müssen nicht anerkannt werden, MPI hat dies in der Voreinstellung geschehen lassen.

Die so „akzeptierten" neuen Datentypen, lassen sich wie die Basisdatentypen, in jedem Kommunikationsbefehl anwenden.

Bemerkung: Das Typematching findet alleine nach der Reihenfolge der Basisdatentypen statt und ist unabhängig von den Verschiebungen und den intermediären Datentypen!

```
MPI_TYPE_FREE(
        datatype                                    INOUT
        )
```

Hiermit werden Datentypen gelöscht (=MPI_DATATYPE_NULL). Dies hat keine Wirkungen auf anstehende Kommunikationen oder davon abgeleitete Datentypen.

```
MPI_GET_ELEMENTS(
        status,                    IN
        datatype,                  IN
        count                      OUT
        )
```

Ist eine Nachricht angekommen, so läßt sich mit der obigen Funktion die Anzahl der Basisdatentypen ermitteln. Ist das Ergebnis kein Vielfaches von n (Anzahl der Basisdatentypen in `datatype`, so ist der Rückgabewert der Funktion `MPI_GET_COUNT` gleich MPI_UNDEFINED. Die letzten beiden Funktionen liefern bei Basisdatentypen den gleichen Wert!

3.5 Packen und Entpacken

```
MPI_PACK(
        inbuf,          IN
        incount,        IN
        datatype,       IN
        outbuf,         OUT
        outsize,        IN
        pos,            INOUT
        comm            IN
        )
```

Zusätzlich zu den bisherigen Routinen bietet MPI, ähnlich anderer Message-Passing Systeme (z.B. PVM), Pack- und Entpackroutinen an. Teils aus Kompatibilität, teils um die Funktionalität von MPI zu erweitern (geeignet für nichtkontinuierliche Daten).

Zum Packen wird der obige Aufruf benutzt. Die Daten an der Adresse `inbuf`, der Länge `incount` und vom Datentyp `datatype` werden in den Puffer `outbuf` mit der Länge `outsize` ab der Anfangsadresse `pos` gepackt, `pos` erhält nach dem Aufruf, die erste „freie" Adresse nach den gepackten Daten. Die Werte der Argumente `outsize` und `pos` beziehen sich auf Bytes.

```
MPI_UNPACK(
          inbuf,                                    IN
          insize,                                   IN
          pos,                                   INOUT
          outbuf,                                  OUT
          outcount,                                 IN
          datatype,                                 IN
          comm                                      IN
          )
```

Umgekehrt funktioniert dies mit der Entpackfunktion. Hier werden `outcount` Daten vom Datentyp `datatype` im Puffer `inbuf`, der `insize` Bytes lang ist, ab der Position `pos` in den Puffer `outbuf` gespeichert. Auch hier bezieht sich der Wert von `pos` auf Bytes.

Nachrichten gepackt und verschickt mit dem Datentyp MPI_PACK, können mit jedem Datentyp empfangen werden. Umgekehrt: jedes Senden mit beliebigem Datentyp kann mit MPI_PACK als Datentyp empfangen und entpackt werden.

<u>Bemerkung:</u> Eine Packeinheit kann nicht in mehrere Entpackeinheiten eingeteilt werden und mehrere Packeinheiten ergeben keine einzelne Packeinheit.

```
MPI_PACK_SIZE(
        incount,                        IN
        datatype,                       IN
        comm,                           IN
        size                            OUT
        )
```

Dieser Aufruf ermittelt die obere Grenze (in Bytes) für den Platz, den die Funktion `MPI_PACK` brauchen würde, wenn sie `incount` Daten vom Typ `datatype` packen würde.

Es folgen zwei Beispiele:

```

  int pos, i, j, a[2];
  char buff[1000];

  ....

  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  if ( myrank == 0 )
  {
     / * SENDER CODE */

    pos = 0;
    MPI_Pack( &i, 1, MPI_INT, buff, 1000, &pos, MPI_COMM_WORLD );
    MPI_Pack( &j, 1, MPI_INT, buff, 1000, &pos, MPI_COMM_WORLD );
    MPI_Send( buff, pos, MPI_PACKED, 1, 0, MPI_COMM_WORLD );
  }
  else  /* RECEIVER CODE */
    MPI_Recv( a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD )

  }
```

```
int pos, i;
float a[1000];
char buff[1000]
....
MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
if ( myrank == 0 ) /* Sender code */
{
  int len[2];
  MPI_Aint disp[2];
  MPI_Datatype type[2], newtype;

  /* neuer Datentyp fuer i gefolgt von a[0]..a[i-1] */
  len[0] = 1;
  len[1] = i;
  MPI_Address( &i, disp );
  MPI_Address( a, disp+1 );
  type[0] = MPI_INT;
  type[1] = MPI_FLOAT;
  MPI_Type_struct( 2, len, disp, type, &newtype );
  MPI_Type_commit( &newtype );

  /* i gefolgt von  a[0]..a[i-1] einpacken */
  pos = 0;
  MPI_Pack( MPI_BOTTOM, 1, newtype, buff, 1000, &pos,
            MPI_COMM_WORLD );

  MPI_Send( buff, pos, MPI_PACKED, 1, 0, MPI_COMM_WORLD )

  /* Eine Alternative zu den letzten drei Zeilen:
     MPI_Send( MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD ); */
}
else /* myrank == 1 -- Empfaenger code */
{
  MPI_Status status;

  MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, &status );

  /* i entpacken */
  pos = 0;
  MPI_Unpack( buff, 1000, &pos, &i, 1, MPI_INT, MPI_COMM_WORLD );

  /* a[0]..a[i-1] entpacken */
  MPI_Unpack( buff, 1000, &pos, a, i, MPI_FLOAT, MPI_COMM_WORLD );
}
```

4 Kollektive Kommunikation

4.1 Einführung und Überblick

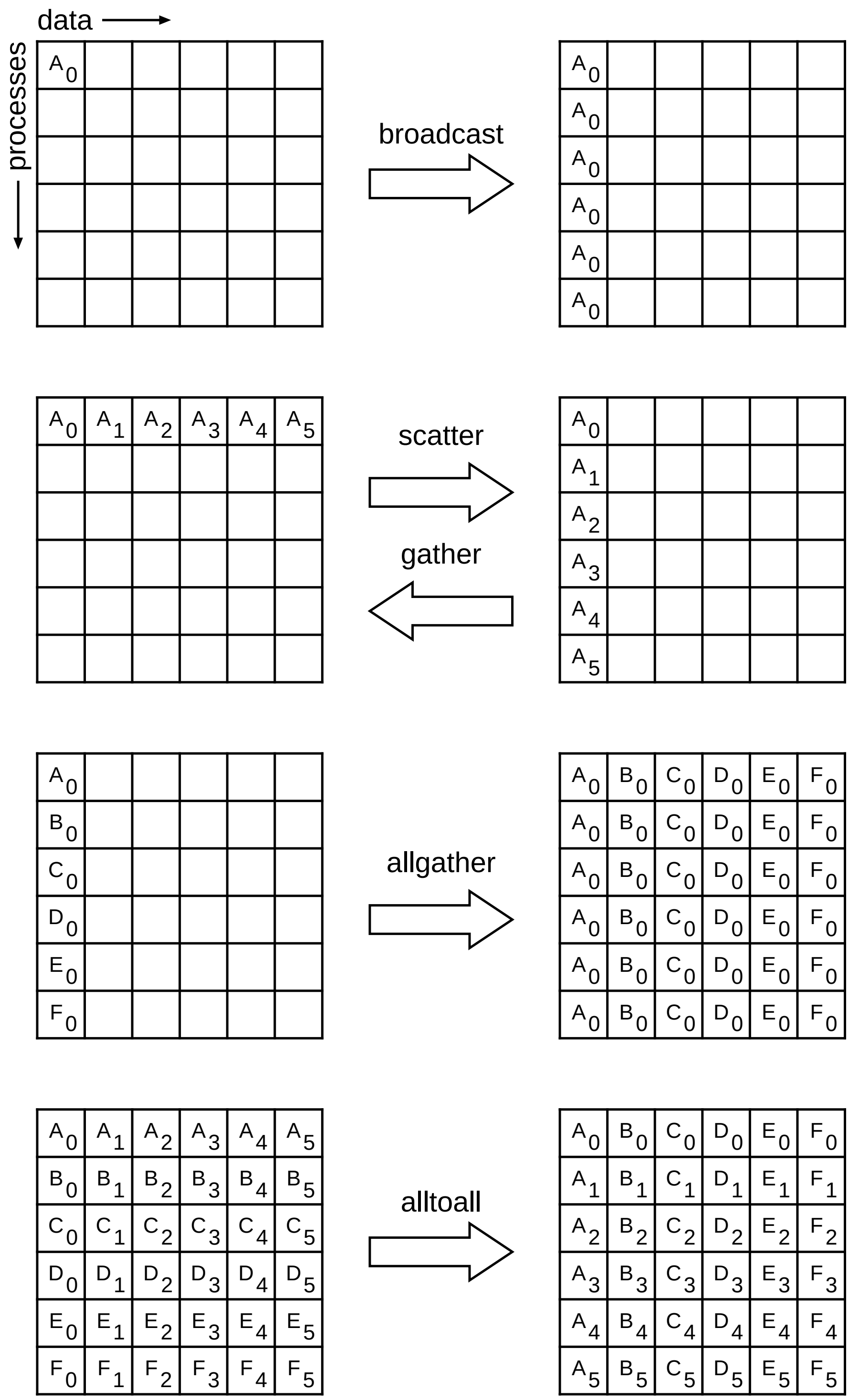
Unter kollektiver Kommunikation versteht MPI die Einbeziehung aller Prozesse in einem Kommunikator:

- Barrierensynchronisation für alle Mitglieder,
- *broadcast* von einem Mitglied zu allen anderen Mitgliedern,
- Sammeln von Daten von allen Mitgliedern zu einem Mitglied (*gather*),
- Verteilen (*scatter*) von Daten von einem Mitglied an alle Mitglieder,
- Variation von *gather*: Alle Mitglieder erhalten die Daten (*allgather*),
- *scatter* und *gather* von allen Mitgliedern zu allen Mitgliedern (*alltoall*),
- Reduktionsoperationen: Empfänger der Ergebnisse entweder alle Mitglieder oder eines,
- *reduction* und *scatter*,
- „*scannen*“ aller Mitglieder.

Eine kollektive Kommunikation wird erst dann ausgeführt, wenn alle Mitglieder die Kommunikationsroutine aufgerufen haben. Die Synchronisation und das Erkennen, ob alle Mitglieder den Aufruf getätigt haben, liegt in der Verantwortung des/der ProgrammiererIn. Die Syntax ist ähnlich der Punkt-zu-Punkt Kommunikation, d.h. auch abgeleitete Datentypen können daran teilnehmen. Bei einigen der kollektiven Operationen gibt es *einen* Sender bzw. Empfänger, dieser wird als *root* bezeichnet.

Vorsicht: Anzahl der versendeten Daten müssen übereinstimmen mit den empfangenen Daten und umgekehrt! Unterschiedliche *type maps* sind erlaubt. Bei „Rückkehr“ der kollektiven Kommunikationsroutine hat der/die BenutzerIn das Recht den Puffer wieder zu nutzen, d.h. aber nicht, daß die anderen Mitglieder fertig sind oder gar begonnen haben den Kommunikationsprozeß auszuführen. Kollektive Kommunikation muß nicht synchronisieren.

Bemerkung: MPI unterscheidet trotz gleichem `comm` (Kommunikator) die Punkt-zu-Punkt Kommunikation von der kollektiven Kommunikation. Außerdem hat MPI die kollektiven Kommunikationsbefehle so definiert, daß sie sich auf den Punkt-zu-Punkt Befehlen stützen können.



Schematische Darstellung der verschiedenen kollektiven Kommunikationen

4.2 Die einzelnen Befehle

```
MPI_BARRIER(
          comm                                      IN
          )
```

Dieser Befehl blockiert so lange den aufrufenden Prozeß, bis alle Mitglieder des Gebietes `comm` diesen Befehl getätigt haben.

```
MPI_BCAST(
        buffer,                      INOUT
        count,                          IN
        datatype,                       IN
        root,                           IN
        comm                            IN
        )
```

Der obige Aufruf (muß jeder Prozeß im Gebiet `comm` tätigen) schickt `count` Daten vom Typ `datatype`, die im Puffer (`buffer`) des Prozesses `root` liegen, an alle Mitglieder des Gebietes, einschließlich an **root** selber.

```
MPI_GATHER(
          sendbuf,                              IN
          scount,                               IN
          sendtype,                             IN
          recvbuf,                              OUT
          rcount,                               IN
          recvtype,                             IN
          root,                                 IN
          comm                                  IN
          )
```

```
MPI_GATHERV(
          sendbuf,                              IN
          scount,                               IN
          sendtype,                             IN
          recvbuf,                              OUT
          rcounts,                              IN
          displs,                               IN
          recvtype,                             IN
          root,                                 IN
          comm                                  IN
          )
```

Die obigen zwei Befehle sammeln Daten aller Mitglieder ein und speichern sie bei einem Mitglied (`root`) ab.

`root` sammelt `rcount` Daten vom Typ `recvtype` in den Puffer `recvbuf` von jedem Prozeß im Gebiet comm. Diese haben ihren Sendepuffer an der Adresse `sendbuf`. Auch `root` muß sich selber Daten schicken. Die Daten werden in der Reihenfolge des Prozeßranges gespeichert.

Bemerkung: (`rcount`, `scount`) und (`recvtype`,`sendtype`) müssen paarweise übereinstimmen, d.h. `rcount` ist die Anzahl der empfangenen Daten des jeweiligen Prozesses und nicht die Summe aller empfangenen Daten.

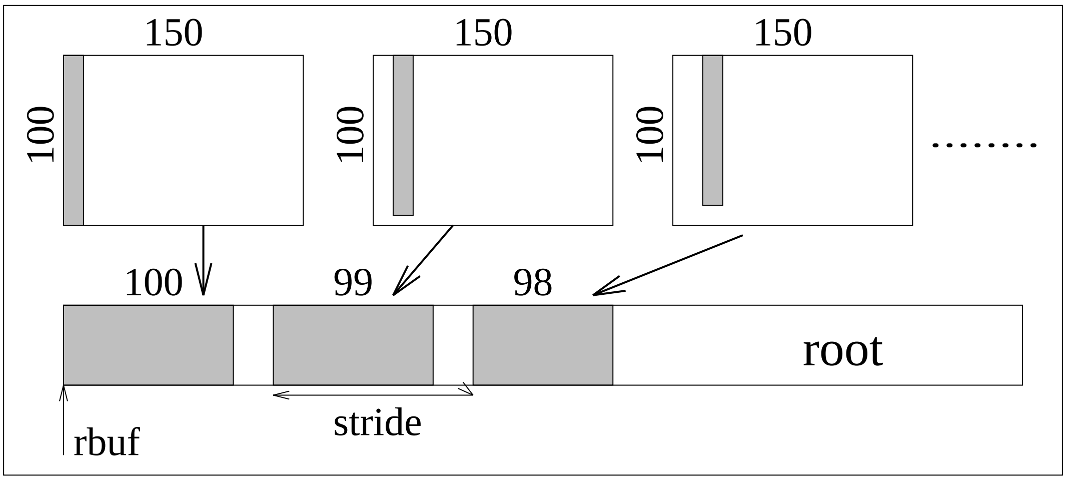
Der zweite Befehl ist eine Verallgemeinerung des ersten: er erlaubt, Dank der Felder `rcounts` und `displs`, die Anzahl der empfangenen Daten individuell für jeden Prozeß einzustellen, auch die Plazierung (in Bytes) der Daten relativ zur Adresse `recvbuf` läßt sich damit, je nach Rangordnung erreichen. Die Reihenfolge der Feldeinträge entspricht der Rangordnung der Prozesse.

Ein Beispiel für die Funktion `MPI_Gatherv`:

```
MPI_Comm     comm;
int          gsize, sendarray[100][150], *sptr;
int          root, *rbuf, stride, myrank;
MPI_Datatype stype;
int          *displs, i, *rcounts;
...
MPI_Comm_size( comm, &gsize );
MPI_Comm_rank( comm, &myrank );
rbuf = (int *)malloc( gsize * stride * sizeof( int ) );
displs = (int *)malloc( gsize * sizeof( int ) );
rcounts = (int *)malloc( gsize * sizeof( int ) );
for ( i = 0; i < gsize; ++i ) {
  displs[i] = i * stride;
  rcounts[i] = 100 - i;
}

/* Neuer Datentyp fuer die zu versendete Spalte */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype );
MPI_Type_commit( &stype );

/* sptr ist die Adresse, von "myrank" Spalte */
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
             root, comm );
```



Sammeln der oberen Hälfte eines Blockes

```
MPI_SCATTER(
          sendbuf,                                  IN
          scount,                                   IN
          sendtype,                                 IN
          recvbuf,                                 OUT
          rcount,                                   IN
          recvtype,                                 IN
          root,                                     IN
          comm                                      IN
          )
```

```
MPI_SCATTERV(
          sendbuf,                                  IN
          scounts,                                  IN
          displs,                                   IN
          sendtype,                                 IN
          recvbuf,                                 OUT
          rcount,                                   IN
          recvtype,                                 IN
          root,                                     IN
          comm                                      IN
          )
```

Das sind die inversen Operationen zu MPI_GATHER. root versendet seine Daten, die im sendbuf liegen und vom Typ sendtype sind, an alle Mitglieder im Gebiet comm.

Der zweite Aufruf ist eine allgemeinere Version des ersteren. Zusätzlich zu der ersten Funktion, bietet dieser die Möglichkeit, jedem Mitglied (entsprechend seiner Rangfolge) eine individuelle Anzahl von Daten zu schicken und diese ab einer beliebigen Adresse (in Bytes) aus sendbuf heraus zu senden.

```
MPI_ALLGATHER(
          sendbuf,                                          IN
          scount,                                           IN
          sendtype,                                         IN
          recvbuf,                                          OUT
          rcount,                                           IN
          recvtype,                                         IN
          comm                                              IN
          )
```

```
MPI_ALLGATHERV(
          sendbuf,                                          IN
          scount,                                           IN
          sendtype,                                         IN
          recvbuf,                                          OUT
          rcounts,                                          IN
          displs,                                           IN
          recvtype,                                         IN
          comm                                              IN
          )
```

Mit dem ersten Befehl bekommt nicht nur ein Prozeß alle Daten im Speicher `sendbuf` des Senders, sondern alle Mitglieder des Bereiches `comm`. Auch hier werden die Daten in der Reihenfolge des Ranges des jeweiligen Mitgliedes gespeichert. Beachte hierzu die Darstellung von *allgather* (die Reihenfolge der Datenadressen des Senders kann sich ändern).

Der zweite Aufruf ist eine allgemeinere Form des vorhergehenden Befehls, da er eine individuelle Anzahl von Daten empfangen kann und diese verschiedene Startadressen (in Bytes) annehmen können. In den beiden letzten Befehlen ist `recvcount` die Anzahl der empfangenen Daten jedes einzelnen Prozesses und <u>nicht deren Summe</u>.

```
MPI_ALLTOALL(
        sendbuf,                              IN
        scount,                               IN
        sendtype,                             IN
        recvbuf,                              OUT
        rcount,                               IN
        recvtype,                             IN
        comm                                  IN
        )
```

```
MPI_ALLTOALLV(
        sendbuf,                              IN
        scount,                               IN
        sdispls,                              IN
        sendtype,                             IN
        recvbuf,                              OUT
        rcounts,                              IN
        rdispls,                              IN
        recvtype,                             IN
        comm                                  IN
        )
```

Der erste Befehl erlaubt, daß jedes Mitglied verschiedene Daten an verschiedene Empfänger verschickt. Physikalisch gesehen ist `sendbuf` `scount*n` (mit n als Anzahl der Prozesse in der Gruppe) groß: ein Block für jeden Empfänger.

Der zweite Befehl erweitert die Flexibilität des vorhergehenden Befehls dadurch, daß er beliebige Startadressen auch innerhalb des Sendepuffers `sendbuf` zuläßt. Auch die Adressen innerhalb des Empfangspuffers `recvbuf` sind variabel: *Der j. Block von Prozeß i gesendet, wird vom j. Prozeß empfangen und an Blockanfang i gespeichert.*

4.3 Globale Operationen

Die jetzt zu erwähnenden Reduktionsoperationen beziehen sich auf Daten aller Prozesse in einer Gruppe. Hierbei werden verschiedene Formen unterstützt:

- Das Ergebnis der globalen Operation erhält nur ein Prozeß
- oder alle Prozesse (*allreduce*),
- ein Scannen von Daten (*parallel prefix*),
- und eine *reduce-scatter*-Operation (Kombination von *reduce* und *scatter*).

Als Rechenoperation dienen vordefinierte, als auch benutzereigene Operationen.

```
MPI_REDUCE(
          sendbuf,          IN
          recvbuf,          OUT
          count,            IN
          dtype,            IN
          op,               IN
          root,             IN
          comm              IN
          )
```

Die globale Operation op wirkt paarweise auf alle Daten im Puffer `sendbuf` aller Mitglieder des Gebietes `comm` und speichert das Ergebnis im Puffer `recvbuf` des Prozesses `root` ab. Die Operation op wird bei mehr als einem Element im Puffer, elementweise ausgeführt. Die von MPI vordefinierten Operationen sind assoziativ und kommutativ. Zumindest assoziativ sollten die benutzereigenen Operationen sein. Die Reihenfolge der ausgeführten Operationen sollte der Rangfolge der Prozesse in der Gruppe entsprechen, aber dies wird nicht zwingend verlangt (nur wenn die globale Operation nichtkommutativ ist).

Folgende `reduce`-Operationen werden von MPI angeboten:

`MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`:
: die ersten vier Operationen suchen elementweise die größte bzw. kleinste Zahl oder bilden die Summe oder das Produkt der Zahlen. Als Datentypen akzeptieren sie Ganz-, Fließkomma- und komplexe Zahlen.

`MPI_LAND`, `MPI_LOR`, `MPI_LXOR`:
: diese Operationen führen logische *und, oder* oder *xor* aus. Als Datentypen akzeptieren sie Ganzzahlen (nur von ANSI C) und logische Werte (s. FORTRAN 77).

`MPI_BAND`, `MPI_BOR`, `MPI_BXOR`:
: diese Operationen führen die ebengenannten Operationen bitweise durch. Als Datentypen akzeptieren sie Ganzzahlen und Bytes.

`MPI_MAXLOC`, `MPI_MINLOC`:
: diese Operationen suchen nach der größten bzw. kleinsten Zahl und geben gleichzeitig den dazugehörigen Index an. Der Index muß vom/von der BenutzerIn explizit gesetzt werden. Dazu bietet MPI für beide Sprachen neue Datentypen an, die gleichzeitig zwei Zahlen beinhalten: die zu vergleichende Zahl und den Index. In FORTRAN 77 sind diese Zahlen vom gleichen Datentyp, während in ANSI C der Index eine Ganzzahl ist und die andere Zahl entsprechend dem Datentyp.

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix} \qquad w = \begin{cases} max(u,v) \\ min(u,v) \end{cases}$$

$$k = \begin{cases} i & \text{wenn} & u > v \\ \text{min}(i,j) & \text{wenn} & u = v \\ j & \text{wenn} & u < v \end{cases} \qquad \text{bzw.} \qquad k = \begin{cases} i & \text{wenn} & u < v \\ \text{min}(i,j) & \text{wenn} & u = v \\ j & \text{wenn} & u > v \end{cases}$$

Hier ein Beispiel mit von MPI vordefinierte globale Operationen:

```
...
/* jeder Prozess hat ein Feld von 30 double: ain[30] */
double ain[30], aout[30];
int    ind[30];
int    i, myrank, root;

struct {
  double val;
  int   rank;
} in[30], out[30];

...

MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
for ( i = 0; i < 30; ++i ){
    in[i].val = ain[i];
    in[i].rank = myrank;
}

MPI_Reduce( in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC,
            root, comm );

/* An dieser Stelle liegt die Antwort beim Prozess "root" */
if ( myrank == root ){
  for ( i = 0; i < 30; ++i ) {
    aout[i] = out[i].val;
    ind[i] = out[i].rank;
  }
}
```

...und das gleiche in FORTRAN 77:

```
      ...
      ! Jeder Prozess hat ein Feld von 30 double: ain(30)

      DOUBLE PRECISION ain(30), aout(30)
      INTEGER          ind(30)
      DOUBLE PRECISION in(2,30), out(2,30)
      INTEGER          i, myrank, root, ierr

      MPI_COMM_RANK( MPI_COMM_WORLD, myrank )
      DO I=1, 30
        in(1,I) = ain(I)
        in(2,I) = myrank ! myrank wird in DOUBLE "gezwungen"
      END DO

      MPI_REDUCE( in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC,
     &            root, comm, ierr )
      ! Das Ergebnis erhaelt "root"

      IF ( myrank .EQ. root ) THEN
        DO I= 1, 30
          aout(I) = out(1,I)
          ind(I) = out(2,I) ! Rang wird in INTEGER "gezwungen"
        END DO
      END IF
```

```
MPI_OP_CREATE(
        function,                                IN
        commute,                                 IN
        op                                       OUT
        )
```

Mit diesem Aufruf wird eine benutzerdefinierte globale Operation an den „handle“ op gebunden, es wird angenommen, daß die Operation assoziativ ist. Ist `commute=true` dann ist die Operation zusätzlich kommutativ, sonst ist sie es nicht (`commute=false`). Ist sie nicht kommutativ, müssen die Operationen mit den Daten in aufsteigender Rangfolge „gefüttert“ werden.

Der Prototyp der Operation für ANSI C lautet wie folgt

```
typedef void MPI_User_function( void *invec, void *inoutvec,
                                int *len, MPI_Datatype *dtype )
```

und unter FORTRAN 77

```
FUNCTION MPI_USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE )
<type>   INVEC(LEN), INOUTVEC(LEN)
INTEGER  LEN, TYPE
```

Die Elemente in den Feldern `invec/INVEC` und `inoutvec/INOUTVEC` werden elementweise miteinander berechnet und das Ergebnis wird im Feld `inoutvec/INOUTVEC` abgelegt.

<u>Vorsicht:</u> Innerhalb dieser benutzerdefinierten Operationen dürfen keine MPI Transferroutinen aufgerufen werden!

```
MPI_OP_FREE(
          op                                          INOUT
          )
```

Dieser Aufruf deallokiert die benutzerdefinierte globale Operation op, op besitzt danach den Wert MPI_OP_NULL.

```
typedef struct {
  double real,imag;
} Complex;

/* die benutzerdefinierte Funktion */
void myProd( Complex *in, Complex *inout, int *len,
             MPI_Datatype *dptr )

{
  int i;
  Complex c;

  for ( i = 0; i < *len; ++i ){
    c.real = inout->real * in->real - inout->imag * in->imag;
    c.imag = inout->real * in->imag + inout->imag * in->real;
    *inout = c;
    in++; inout++;
  }
}

  /* der Aufruf findet jetzt statt */

  ...

  /* jeder Prozess hat ein Feld von 100 komplexen Zahlen */
  Complex a[100], answer[100];
  MPI_Op       myOp;
  MPI_Datatype ctype;

  /* Definition des neuen Types COMPLEX */
  MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
  MPI_Type_commit( &ctype );

  /* die benutzerdefinierte Funktion als globale Operation
     anmelden */
  MPI_Op_create( myProd, True, &myOp );

  MPI_Reduce( a, answer, 100, ctype, myOp, root, comm );

  /* An dieser Stelle liegt das Ergebnis beim Prozess "root" */
```

```
MPI_ALLREDUCE(
        sendbuf,                    IN
        recvbuf,                    OUT
        count,                      IN
        datatype,                   IN
        op,                         IN
        comm                        IN
        )
```

Eine Modifikation des zuletzt vorgestellten MPI-Aufrufes beinhaltet der obige Aufruf. Das Ergebnis der globalen Operation op wird nicht an einen Prozeß, sondern an alle Prozesse des „Universums“ comm geschickt.

```
MPI_REDUCE_SCATTER(
          sendbuf,                                   IN
          recvbuf,                                   OUT
          recvcounts,                                IN
          datatype,                                  IN
          op,                                        IN
          comm                                       IN
          )
```

Mit diesem Aufruf ist es möglich jedem Prozeß individuell eine bestimmte Anzahl von Ergebnisdaten (`recvcounts`) zu schicken. `sendbuf` ist dabei die Summe aller Elemente des Feldes `recvcounts`.

```
MPI_SCAN(
        sendbuf,                  IN
        recvbuf,                  OUT
        count,                    IN
        datatype,                 IN
        op,                       IN
        comm                      IN
        )
```

Im `recvbuf` dieses Aufrufes steht das Ergebnis der globalen Operation. Dieser Befehl legt im `recvbuf` des iten Prozesses die Ergebnisse der Werte aus `sendbuf` aller Prozesse vom Rang 0 bis einschließlich i.

5 Gruppen, Kontexte und Kommunikatoren

Das folgende Kapitel stellt MPI-Aufrufe vor, die die Arbeit von Bibliotheksentwickler erleichtern soll. Diese Bibliotheken sollen den/die BenutzerIn davor „bewahren“, sich mit den, für parallele Implementationen innewohnenden Komplikationen aufzuhalten.

Folgende Eigenschaften werden benötigt um Bibliotheken zu unterstützen:

- Ein sicherer Kommunikationsraum, d.h. die Kommunikation, die durch den Aufruf von Bibliotheksfunktionen entsteht, soll nicht mit bibliothksfremder Kommunikation wechselwirken bzw. gestört werden.
- Die Sichtweise der kollektiven Operationen sollte sich an Gruppen orientieren, um mögliche Synchronisationsprobleme zu vermeiden.
- Die Namensgebung sollte sich nach der zu erfüllenden Aufgabe richten.
- Die Möglichkeit, die Kommunikationsprozesse mit benutzerdefinierten Optionen und Wünschen zu erweitern.

MPI bietet im folgende Konzepte für das Unterstützen stabiler Bibliotheken:

- Kommunikations*kontexte* / *Contexts* of communication,
- Prozeß*gruppen* / *Groups* of processes,
- *Virtuelle Topologien* / *Virtual topologies*,
- *Attribute caching*,
- *Kommunikatoren* / *Communicators*

Kommunikatoren Kommunikatoren sind in der Hierarchie des Bibliothekbaus an oberster Stelle, sie umhüllen alle vorhergehenden Ideen. Kommunikatoren lassen sich sich in zwei Arten einteilen:

1. Intra-Kommunikatoren ist das meißt verwendetete Mittel zur Kommunikation. Diese beinhalten eine Gruppe und einen Kontext für Punkt-zu-Punkt und kollektive Kommunikation:

   Kontexte bieten sichere „Universen" und lassen sich mit einer zusätzlichen Kennung vergleichen. Diese „Universen" ermöglichen innerhalb eines Kontexts einen störungsfreien Datentransfer. (Anhängende Kommunikation wird nicht beeinflußt.)

   Gruppen definieren ein sortierte Sammlung von Prozessen in der jeder eine Rangnummer besitzt. Gruppen definieren die Low-Level-Namen für die IPC. Gruppen können unabhängig von Kommunikatoren manipuliert werden, aber nur die Kommunikatoren können in der Kommunikation referiert werden.

   Virtuelle Topologien definieren ein spezielles Mappen der Prozeßränge in einer Gruppe auf eine Topologie.

   Attribute enthalten weitere lokale Informationen über den Kommunikator, auf die später zugegriffen wird.

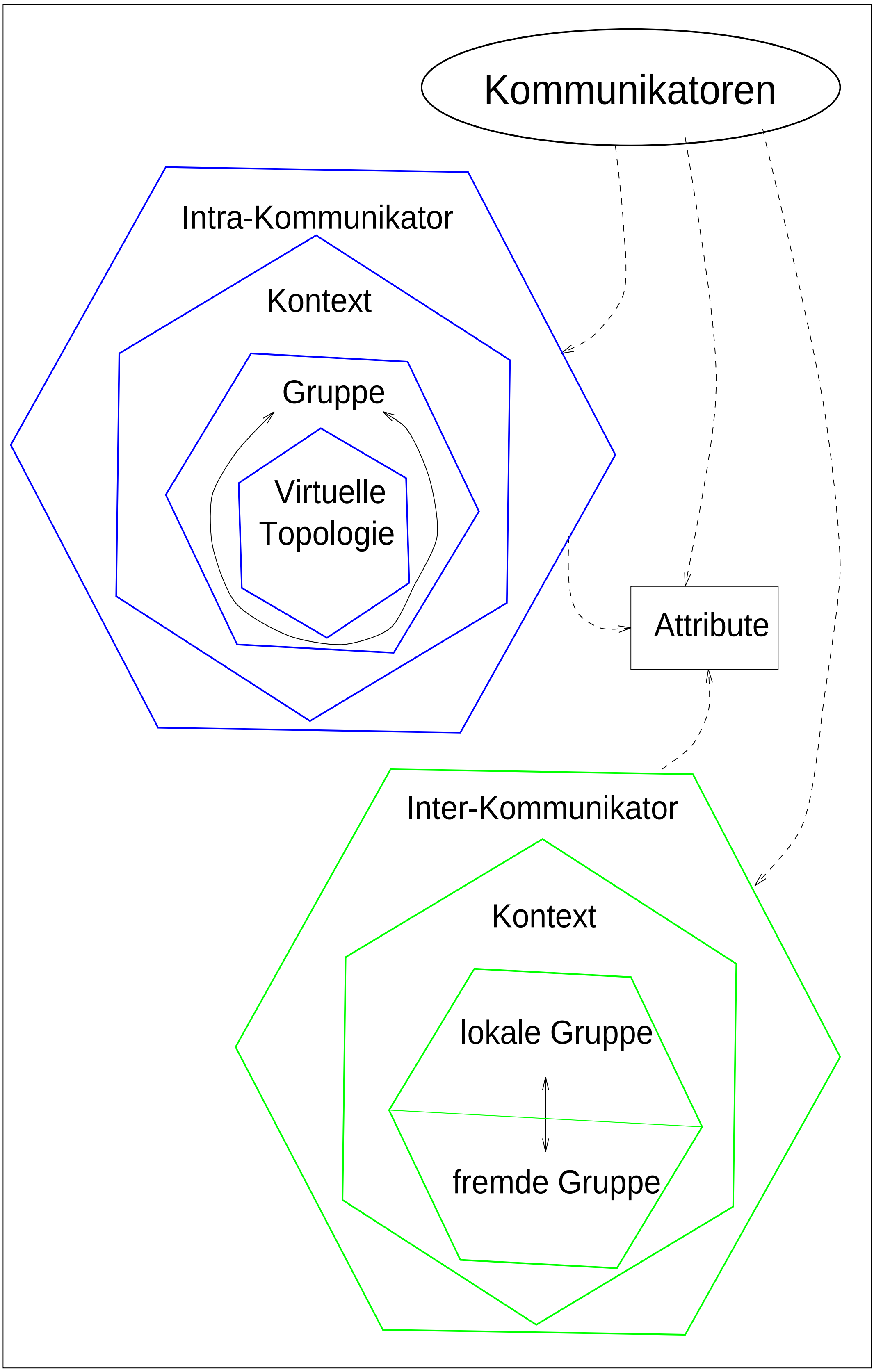
2. Inter-Kommunikatoren dienen der Kommunikation zwischen zwei nicht-überlappenden Gruppen. Dies erlaubt ein dynamisches Kommunikationsmodell, in denen zwischen Gruppen kommuniziert werden kann, die sich nicht zur gleichen Zeit initialisierten.

   Kontexte bieten die Möglichkeit ein sicheres „Universum" zwischen zwei Gruppen zu bilden. Es wird gewährleistet, daß die Kommunikation immer von einer Gruppe zur nächsten stattfindet und nicht innerhalb der Gruppen wechselwirkt. Nur Punkt-zu-Punkt Kommunikationen wird angeboten.

   Gruppen spezifizieren die Empfänger für einen Inter-Kommunikator.

   Virtuelle Topologien sind nicht definiert.

   Attribute ähnlich wie für Intra-Kommunikatoren.



Schematische Darstellung der Kommunikatoren

- Ein Kommunikator wird von einem *opaque intra-communicator* Objekt dargestellt und kann somit nicht von einem Prozeß zum anderen transferiert werden. Kommunikatoren bringen das Konzept der Gruppen und Kontexte zusammen. Um implemenationsbedingte Optimierungen gerecht zu werden, speichert ein Kommunikator zusätzliche Attribute zwischen.

  MPI bietet zwei voreingestellte Kommunikatoren (nach Aufruf von `MPI_INIT`):

  - MPI_COMM_WORLD, dieser Kommunikator hat in einer statischen MPI-Konfiguration für alle Prozesse den gleichen Wert, für eine dynamische Prozeßkonstellation unterscheidet sich der Wert dieser „Konstanten".

    Vorsicht: Prozesse die zu einem bestimmten Zeitpunkt initialisiert werden, können mit MPI_COMM_WORLD nur die Prozesse erreichen, die zum gleichen Zeitpunkt der Initialisierung zur Verfügung standen.
  - MPI_COMM_SELF, dieser Kommunikator bezieht sich immer auf den eigenen Prozeß.

  MPI gibt bei Angabe eines ungültigen *communicator* „handles" die Konstante MPI_COMM_NULL zurück.

- Gruppen werden von einem *opaque group* Objekt verköpert. Gruppen beschreiben die Prozesse, die innerhalb eines Universums miteinander kommunizieren. MPI bietet eine vordefinierte Gruppe an: MPI_GROUP_EMPTY, eine Gruppe mit keinem Prozeß. Die Konstante MPI_GROUP_NULL wird für fehlerhafte *group* „handles" zurückgegeben.

- Ein Kontext ist eine *Eigenschaft* der Kommunikatoren, die es erlaubt einen Kommunikationsraum einzuteilen. Nachrichten in einem Kontext können diesen nicht verlassen. Außerdem kommunizieren Punkt-zu-Punkt und kollektive Kommunikation in separaten Kontexten. Kontexte werden nicht durch *opaque* Objekte dargestellt.

5.1 Gruppen-Management

```
MPI_GROUP_SIZE(
          group,                                          IN
          size                                           OUT
          )
```

Im Folgenden werden MPI-Aufrufe vorgestellt, die keine Interprozeßkommunikation erfordern.

Der obige Befehl ermittelt die Anzahl (`size`) der Prozesse in der Gruppe `group`.

```
MPI_GROUP_RANK(
        group,                                 IN
        rank                                   OUT
        )
```

hiermit erhält der aufrufende Prozeß seinen Rang (`rank`) in der Gruppe oder MPI_UNDEFINED, falls er nicht Mitglied der Gruppe ist.

```
MPI_GROUP_RANKS(
          group1,                                    IN
          n,                                         IN
          rank1,                                     IN
          group2,                                    IN
          rank2                                     OUT
          )
```

Die Felder `rank1` und `rank2` haben `n` Elemente (Ganzzahlgrößen). In `rank2` stehen die übersetzten Ränge der Prozesse aus der Gruppe `group1` in der Gruppe `group2`. Ist keine Korrespondenz vorhanden, so ist der Rückgabewert MPI_UNDEFINED.

```
MPI_GROUP_COMPARE(
        group1,                     IN
        group2,                     IN
        result                      OUT
        )
```

Der letzte Befehl vergleicht beide Gruppen (group1 und group2) nach gleichen Prozessen und Rängen:

- MPI_IDENT: beide Gruppen haben die gleichen Prozesse und Ränge,
- MPI_SIMILAR: beide Gruppen haben die gleichen Prozesse, aber unterschiedliche Ränge,
- MPI_UNEQUAL: sonst.

MPI bietet die Möglichkeit, neue Gruppen zu bilden. Neue Gruppen können nur aus bestehenden Gruppen entstehen. Die Basisgruppe aller neuen Gruppen ist die, die zum Kommunikator MPI_COMM_WORLD gehört: MPI_COMM_GROUP, aus ihr leiten sich alle anderen Gruppen her. Ein Prozeß der eine Gruppe bildet, muß der neue Gruppe nicht angehören.

```
MPI_COMM_GROUP(
          comm,                                   IN
          group                                   OUT
          )
```

Mit `MPI_COMM_GROUP` ermittelt der aufrufende Prozeß die zum Kommunikator `comm` zugehörige Gruppe `group`.

```
MPI_GROUP_UNION/_INTERSECTION/_DIFFERENCE(
          group1,                                    IN
          group2,                                    IN
          newgroup                                   OUT
          )
```

Die drei obigen MPI-Aufrufe erstellen mit Hilfe zweier Gruppen (`group1` und `group2`) eine neue Gruppe (`newgroup`). Hierbei gelten nach der obigen Reihenfolge:

1. `_UNION`: in `newgroup` sind alle Prozesse aus `group1` und alle Prozesse aus `group2`, die nicht in `group1` sind.
2. `_INTERSECTION`: in `newgroup` sind alle Prozesse, die in `group1` als auch in `group2` sind.
3. `_DIFFERENCE`: in `newgroup` sind alle Prozesse, die in `group1` sind und nicht in `group2`.

Die Ordnung der Prozeßränge in der neuen Gruppe orientiert sich nach der Ordnung in `group1` und erst dann nach der in `group2`. Gegebenenfalls wird der Wert MPI_GROUP_EMPTY zurückgegeben.

```
MPI_GROUP_INCL(
        group,                              IN
        n,                                  IN
        ranks,                              IN
        newgroup                            OUT
        )
```

Mit dem obigen MPI-Befehl läßt sich einer neuen Gruppe `newgroup` `n` Prozesse aus einer existierenden Gruppe `group` zuordnen; hierbei stehen im Feld `ranks` die Ränge der Prozesse, die übernommen werden sollen.

```
MPI_GROUP_EXCL(
        group,                                   IN
        n,                                       IN
        ranks,                                   IN
        newgroup                                 OUT
        )
```

In die neue Gruppe `newgroup` kommen alle Prozesse aus der Gruppe `group`, die nicht in der Feld `ranks` aufgelistet sind.

```
MPI_GROUP_RANGE_INCL/_EXCL(
          group,                                   IN
          n,                                       IN
          ranges,                                  IN
          newgroup                                OUT
          )
```

Eine allgemeinere Erstellung einer Gruppe (`newgroup`) mit Hilfe von Prozessen einer (`n, ranks`) bestehenden Gruppe (`group`) erlauben die obigen zwei Befehle. Das Feld `ranges` ist eine Liste von Triples (`first, last, stride`), mit

- `first` ist der erste ausgewählte Prozeßrang aus der Gruppe `group`,
- `last` ist der letzte ausgewählte Prozeßrang aus der Gruppe `group` und
- `stride` ist die Schrittweite, mit der die Prozeßränge ausgewählt werden.

Etwas formaler geschrieben, läßt sich das Triple wie folgt interpretieren: Beinhaltet das Feld `ranges` die Triple

$$(first_1, last_1, stride_1) \ldots (first_n, last_n, stride_n)$$

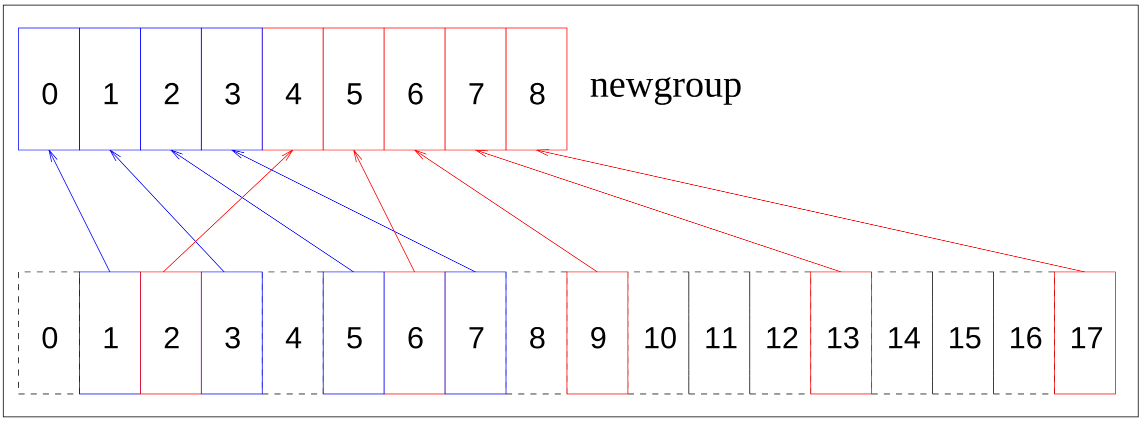
dann hat `newgroup` folgende Sequenz von Prozessen aus der Gruppe `group`

$$\begin{array}{l} first_1, \quad first_1 + stride_1, \ldots, first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, \quad \ldots \\ first_n, \quad first_n + stride_n, \ldots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n. \end{array}$$

Alle Ränge dürfen nur einmal vorkommen, schließlich kann

$$first_i > last_i$$

gelten mit $stride_i$ negativ.



MPI_Group_range_incl(group, 2, ranges, &newgroup)

mit ranges=((2,8,2),(3,18,4))

```
MPI_GROUP_FREE(
          group                                              INOUT
          )
```

Mit dem obigen „Gruppenbefehl“ läßt sich ein Gruppen-Objekt deallokieren. Der Gruppen-„handle“ bekommt den Wert MPI_GROUP_NULL. Alle laufenden Operationen, die diese Gruppe benutzen, werden ordnungsgemäß beendet.

5.2 Kommunikator-Management

```
MPI_COMM_SIZE(
        comm,                                    IN
        size                                     OUT
        )
```

Nach der Erwähnung der „Gruppenbefehle“ folgen jetzt die Befehle zur Behandlung von Kommunikatoren. Der Zugriff auf die Kommunikatoren ist lokal, nicht aber die Operationen, die Kommunikatoren erzeugen. Die letzteren Befehle sind kollektiv und können IPC verlangen.

Mit dem obigen Befehl erhält das aufrufende Programm die Anzahl (`size`) der Prozesse im Kommunikator `comm`. Das Resultat ist das gleiche, wie der nacheinanderfolgende Aufruf von:

`MPI_COMM_GROUP`, `MPI_GROUP_SIZE` und `MPI_GROUP_FREE`

```
MPI_COMM_RANK(
          comm,                                          IN
          rank                                           OUT
          )
```

Mit diesem MPI-Aufruf erhält der aufrufende Prozeß den eigenen Rang (`rank`) im Kommunikator `comm`. Auch hier gibt es eine äquivalente Reihenfolge von MPI-Befehlen, die das gleiche Ergebnis liefert:

`MPI_COMM_GROUP`, `MPI_GROUP_RANK` und `MPI_GROUP_FREE`

Bemerkung: Üblich ist die Vorgehensweise im Rahmen eines Master-Slave-Modells, den Prozeß mit dem Rang 0 als Master auszuwählen.

```
MPI_COMM_COMPARE(
        comm1,                                IN
        comm2,                                IN
        result                                OUT
        )
```

Der letzte Zugriffsbefehl vergleicht zwei Kommunikatoren (`comm1` und `comm2`) miteinander. `result` kann, je nach Fall, folgende Ergebnisse liefern:

- MPI_IDENT, wenn die zwei Gruppen-„handles“ `comm1` und `comm2` auf das gleiche Objekt weisen (identische Gruppen und Kontexte),
- MPI_CONGRUENT, wenn die zugehörigen Gruppen im Aufbau und Rangfolge identisch sind, die Kommunikatoren sich aber im Kontext unterscheiden,
- MPI_SIMILAR, wenn die zugehörigen Gruppen die gleichen Prozesse als Mitglieder haben, aber sie sich in der Rangfolge unterscheiden,
- MPI_UNEQUAL sonst.

Mit den bisherigen MPI-Aufrufen konnten die aufrufenden Prozesse Gruppen bilden und Informationen über Gruppen und Kommunikatoren erhalten. Eine sinnvolle Nutzung entsteht mit den MPI-Aufrufen zu Erzeugung von Kommunikatoren. Die nun folgenden vier Befehle sind global, d.h. sie müessen von allen Prozessen aufgerufen werden, die in der Gruppe `comm` sind (s.u.). Der Basiskommunikator aller anderen Gruppen ist MPI_COMM_WORLD und wird „außerhalb“ von MPI definiert.

```
MPI_COMM_DUP(
          comm,                                  IN
          newcomm                                OUT
          )
```

Die obige Funktion dupliziert den Kommunikator `comm` nach `newcomm`. `newcomm` besitzt die gleiche Gruppe und alle zusätzlichen Informationen des Kommunikators (*attribute caching*), aber einen neuen Kontext. Das obige Duplizieren bezieht sich auf Intra- als auch Inter-Kommunikatoren. Eine sinnvolle Verwendung des Befehls ist der Wunsch einen Bibliotheksaufruf parallel aufzurufen.

<u>Bemerkung:</u> Die Gruppe muß nicht kopiert werden, es reicht wenn ein weiterer Verweis auf ihr stattfindet.

```
MPI_COMM_CREATE(
        comm,                                   IN
        group,                                  IN
        newcomm                                 OUT
        )
```

Der obige MPI-Aufruf erzeugt einen Kommunikator (`newcomm`), der die Gruppe `group` besitzt, die wiederum im Kommunikator `comm` (zumindest als Untermenge) existieren muß.

Diese Funktion hat nur für Intra-Kommunikatoren Gültigkeit. Zusätzlich zu den verschiedenen Kontexten der beiden Kommunikatoren, werden die zusätzlichen Eigenschaften von `comm` nicht übernommen.

Bemerkung: Alle Prozesse von `comm` müssen den obigen Aufruf tätigen, auch wenn sie nicht der Gruppe `group` angehören. MPI fordert von der Implementation, daß wenn Kommunikatoren kreiert werden ohne die zugehörigen Prozesse zu synchronisieren, Nachrichten die an einem Prozeß geschickt werden, der noch nicht im neuen Kontext allokiert ist, „richtig" behandelt werden.

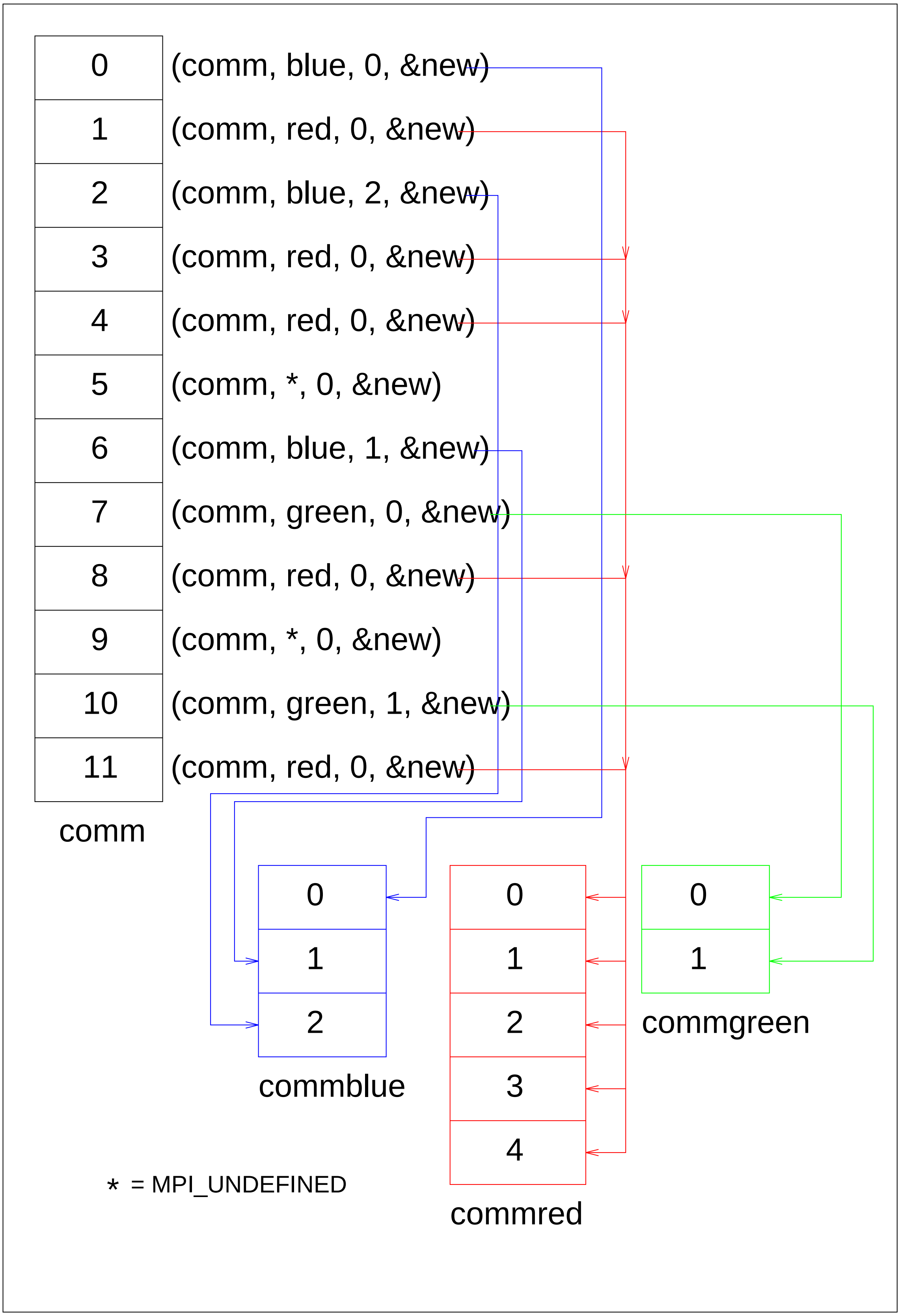
`MPI_COMM_CREATE` gibt dem/der ProgrammiererIn die Mittel in die Hand, einer Untermenge von Prozessen einen eigenen Kommunikationsraum zur Verfügung zu stellen (z.B. in einer separaten MIMD-Rechnung).

```
MPI_COMM_SPLIT(
        comm,                     IN
        color,                    IN
        key,                      IN
        newcomm                   OUT
        )
```

Eine Verallgemeinerung der Erzeugung von Kommunikatoren ist der obige MPI-Aufruf.

Jeder Prozeß im Kommunikator `comm` muß den obigen Aufruf tätigen. Mit diesem Aufruf trägt sich der Prozeß in den neuen Kommunikator `newcomm` ein. `newcomm` „verwaltet" eine neue Gruppe, in dieser Gruppe sind alle Prozesse der Farbe `color`, jeder dieser Prozesse hat den Rang, den er selber gewählt hat (`key`). Möchte ein Prozeß nicht in einen neuen Kommunikator, gibt er den `color`-Wert MPI_UNDEFINED an.

<u>Bemerkung:</u> Geben alle Prozesse gleicher Farbe (`color`) als `key` den Wert Null an, so sortiert der Kommunikator `newcomm` die Prozesse nach einer beliebigen Reihenfolge. Geben alle Prozesse gleicher Farbe als `key` den gleichen Wert, aber ungleich Null an, dann werden sie nach der relativen Rangfolge im alten Kommunikator sortiert. Auch dieser Aufruf ist nur auf Intra-Kommunikatoren anwendbar.



Beispiel von `MPI_COMM_SPLIT`

```
MPI_COMM_FREE(
          comm                                    INOUT
          )
```

„Wie zu erwarten“, kommt als nächster Befehl, das Deallokieren eines Kommunikators. Der Aufruf einer solchen Deallokation muß von allen Prozessen des zu deallokierenden Kommunikators `comm` ausgeführt werden.

Der zurückgegebene „Handle“ wird auf MPI_COMM_NULL gesetzt. Jede „anhängende Operation“ wird ordnungsgemäß abgeschlossen. Eine tatsächliche Deallokation findet nur dann statt, wenn keine weiteren Verweise auf diesen Kommunikator existieren. Der obige Aufruf gilt sowohl für Intra- als auch Inter-Kommunikatoren.

```
main( int argc, char **argv )
{
  int me, count, count2, ranks[] = {0};
  void *sbuf, *rbuf, *sbuf2, *rbuf2;
  MPI_Group MPI_GROUP_WORLD, grprem;
  MPI_Comm commslave;
  ...
  MPI_Init( &argc, &argv );
  MPI_Comm_group( MPI_COMM_WORLD, &MPI_GROUP_WORLD );
  MPI_Comm_rank( MPI_COMM_WORLD, &me );  /* lokal */

  /* lokal */
  MPI_Group_excl( MPI_GROUP_WORLD, 1, ranks, &grprem );

  MPI_Comm_create( MPI_COMM_WORLD, grprem, &commslave );

  if( me != 0 ){ /* Rechnung auf dem Sklaven */
    ...
    MPI_Reduce( sbuf, rbuf, count, MPI_INT, MPI_SUM, 0,
                commslave );
    ...
  }
  /* Null faellt sofort auf diese Zeile, der Rest spaeter .. */
  MPI_Reduce( sbuf2, rbuf2, count2,
              MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );

  MPI_Comm_free( &commslave );
  MPI_Group_free( &MPI_GROUP_WORLD );
  MPI_Group_free( &grprem );
  MPI_Finalize();
}
```

Das folgende Beispiel, soll demonstrieren, daß Punkt-zu-Punkt und kollektive Kommunikation sich nicht überlappen, „geschweige denn“ miteinander wechselwirken:

```
#define TAG_ARBITRARY 12345
#define SOME_COUNT      50

main( int argc, char **argv )
{
  int me;
  MPI_Request request[2];
  MPI_Status status[2];
  MPI_Group MPI_GROUP_WORLD, subgroup;
  int ranks[] = {2, 4, 6, 8};
  MPI_Comm the_comm;
  ...
  MPI_Init( &argc, &argv );
  MPI_Comm_group( MPI_COMM_WORLD, &MPI_GROUP_WORLD );

  /* lokal */
  MPI_Group_incl( MPI_GROUP_WORLD, 4, ranks, &subgroup );
  MPI_Group_rank( subgroup, &me );

  MPI_Comm_create( MPI_COMM_WORLD, subgroup, &the_comm );

  if ( me != MPI_UNDEFINED )
  {
    MPI_Irecv( buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE,
               TAG_ARBITRARY, the_comm, request );
    MPI_Isend( buff2, count, MPI_DOUBLE, ( me + 1 ) % 4,
               TAG_ARBITRARY, the_comm, request+1 );
  }

  for ( i = 0; i < SOME_COUNT, i++ )
    MPI_Reduce( ..., the_comm );
  MPI_Waitall( 2, request, status );

  MPI_Comm_free( &the_comm );
  MPI_Group_free( &MPI_GROUP_WORLD );
  MPI_Group_free( &subgroup );
  MPI_Finalize();
}
```

5.3 Inter-Kommunikation

Alle bisherigen Kommunikationsbefehle in MPI bezogen sich auf Prozesse innerhalb einer Gruppe: Intra-Kommunikation. Möchten zwei Prozesse miteinander kommunizieren, die in verschiedenen Gruppen sich aufhalten, so fällt das in MPI unter dem Begriff Inter-Kommunikation.

Ein Beispiel solcher Kommunikation ist das Client-Server-Modell, in dem jeder Server einer Prozeßgruppe angehört und mit verschiedenen Clients kommuniziert, die wiederum in verschiedenen Prozeßgruppen gegliedert sind.

Die Inter-Kommunikation in MPI ist auf Punkt-zu-Punkt Kommunikation beschränkt. Hierzu einige Begriffe:

- Eine Gruppe, die einen Prozeß besitzt, der eine Inter-Kommunikation initiiert, heißt „lokale Gruppe", das ist der Sender einer Sendung und der Empfänger in einem Empfang.
- Die Gruppe, die den Targetprozeß besitzt wird „*remote* Gruppe" genannt, das ist der Empfänger in einer Sendung und der Sender in einem Empfang.
- Der Targetprozeß in einer Inter-Kommunikation wird mit einem (Rang, Kommunikator)-Paar gekennzeichnet (der Rang ist relativ zur *remote* Gruppe).

Folgende Eigenschaften gelten:

- Kommunikation, die einen Inter-Kommunikator verwendet kommt nicht in Konflikt mit Kommunikation, die einen anderen Kommunikator verwendet;
- ein Inter-Kommunikator kann nicht für kollektive Kommunikaton verwendet werden;
- ein Kommunikator bietet entweder Inter- oder Intra-Kommunikation an, aber nicht beides.

```
MPI_COMM_TEST_INTER(
        comm,                                   IN
        flag                                    OUT
        )
```

Der aufrufende Prozeß erfährt mit der obigen Funktion, ob der Kommunikator `comm` ein Inter-Kommunikator ist; (`flag` ist in diesem Fall true).

Werden die weiter vorne stehenden MPI-Aufrufe mit einem Inter-Kommunikator als Argument aufgerufen, so erhält man folgende Ergebnisse:

MPI_COMM_* Funktionsverhalten (im Inter-Kommunikationsmodus)	
`MPI_COMM_SIZE`	Größe der lokalen Gruppe
`MPI_COMM_GROUP`	Lokale Gruppe
`MPI_COMM_RANK`	Rang in der lokalen Gruppe

<u>Bemerkung:</u> Die Funktion `MPI_COMM_COMPARE` sollte als Argumente nur Kommunikatoren bekommen, die entweder beide nur Inter- oder nur Intra-Kommunikation ausführen.

```
MPI_COMM_REMOTE_SIZE/_GROUP(
        comm,                                   IN
        size/group                              OUT
        )
```

Symmetrische MPI-Aufrufe zu den vorherigen Informationsfunktionen werden für *remote* Gruppen angeboten. Vergleiche hierzu die symmetrischen Funktionen `MPI_COMM_SIZE` und `MPI_COMM_GROUP`, mit dem Unterschied, daß die Informationen sich auf die *remote* Gruppen beziehen.

```
MPI_INTERCOMM_CREATE(
        loc_comm,                         IN
        loc_leader,                       IN
        peer_comm,                        IN
        remote_leader,                    IN
        tag,                              IN
        newintercomm                     OUT
        )
```

Die nun folgenden vier Inter-Kommunikations Operationen blockieren, und verlangen deshalb, daß die lokale und *remote* Gruppe disjunkt sind, um so Deadlocks zu vermeiden.

Bemerkung: Die zwei Gruppen in einem Inter-Kommunikator dürfen nicht überlappen, tun sie es doch, besteht die Gefahr eines Deadlocks.

Sämtliche Prozesse in den Gruppen der beiden Kommunikatoren (`loc_comm` und `peer_comm`) müssen den obigen Aufrut tätigen. Die Prozesse bestimmen den Rang `loc_leader` des Prozesses in der lokalen Gruppe, der mit dem Prozeß in der *remote* Gruppe (`remote_leader`) kommunizieren soll. Der Paar-Kommunikator `peer_comm` (sollte in den meißten Fällen MPI_COMM_WORLD sein) ist der Kommunikator, der die Punkt-zu-Punkt Verbindung zwischen den beiden Gruppenführern (`loc_leader` und `remote_leader`) aufbaut. Der obige Aufruf benutzt eine Punkt-zu-Punkt Kommunikation mit dem Kommunikator `peer_comm` und der Kennung `tag`.

Zusammenfassung: Wollen zwei Gruppen miteinander kommunizieren, muß mindestens ein Prozeß in jeder Gruppe einen gemeinsamen Kommunikator haben, das ist auf jeden Fall MPI_COMM_WORLD. Über diese Prozesse und Kommunikator verläuft schließlich die Inter-Kommunikation.

Vorsicht wird empfohlen, daß im Paar-Kommunikator keine weitere Kommunikation anhängt, die mit der obigen Kommunikation „kollidieren“ könnte.

```
MPI_INTERCOMM_MERGE(
        intercomm,                  IN
        high,                       IN
        newintracomm                OUT
        )
```

Zwei Gruppen, die sich in einem Inter-Kommunikator (`intercomm`) aufhalten, können zu einem Intra-Kommunikator (`newintracomm`) zusammengefaßt werden. Hierbei bestimmt der Flag `high` in welcher Rangfolge die Prozesse beider Gruppen angeordnet werden sollen. Rufen die Prozesse in einer Gruppe das `high`-Argument mit dem Wert false auf, so werden die Prozesse dieser Gruppe vor den Prozessen der anderen Gruppe angeordnet. Ist der Wert in beiden Gruppen gleich, so ist die Rangfolge beliebig.

Bemerkung: Alle Prozesse in einer Gruppe sollten für das `high`-Argument den gleichen Wert wählen.

Die letzten zwei MPI-Operationen für Inter-Kommunikatoren sind weiter vorne aufgeführt. Es handelt sich um `MPI_COMM_DUP` und `MPI_COMM_FREE`. Diese Operationen akzeptieren als Argumente auch Inter-Kommunikatoren.

```
main( int argc, char **argv )
{
  MPI_Comm   myComm;    /* Intra-Kommunik. der lok. Untergruppe */
  MPI_Comm   myFirstComm;  /* Inter-Kommunikator */
  MPI_Comm   mySecondComm; /* 2. Inter-Kommunik. (nur Gruppe 1) */
  int membershipKey, rank;

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  /* Die Gruppenzugehoerigkeit wird bestimmt [0, 1, 2] */
  membershipKey = rank % 3;

  /* Allok. eines Intra-Kommunik. fuer die lok. Untergruppe */
  MPI_Comm_split( MPI_COMM_WORLD, membershipKey, rank, &myComm );

  /* Bau der Inter-Kommunikatoren, Kennung ist "hart verdrahtet" */
  if ( membershipKey == 0 ){      /* Gruppe 0 mit Gruppe 1 */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          1, &myFirstComm);
  }
  else if ( membershipKey == 1 ){ /* Gruppe 1 mit Gruppe 0 und 2 */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          1, &myFirstComm );
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          12, &mySecondComm );
  }
  else if ( membershipKey == 2 ){ /* Gruppe 2 mit Gruppe 1 */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          12, &myFirstComm );
  }
  ...
  switch( membershipKey ){  /* Kommunikatoren deallokieren */
    case 1:
      MPI_Comm_free( &mySecondComm );
    case 0:
    case 2:
      MPI_Comm_free( &myFirstComm );
      break;
  }

  MPI_Finalize();
}
```

5.4 Caching

MPI bietet die Möglichkeit zusätzliche Eigenschaften von Kommunikatoren zu verändern oder Informationen darüber zu erhalten. Diese zusätzlichen Eigenschaften (*attributes*) und deren Manipulation erlauben einer portablen Bibliothek folgende Möglichkeiten:

- Informationen zwischen zwei Aufrufen weitergeben, dies passiert mit einem Intra- oder Inter-Kommunikator,
- Schneller Zugriff auf diese Informationen und
- die Garantie, daß auf veraltete Informationen nicht zugegriffen werden kann.

Diese *caching*-Informationen werden auch von MPI intern verwendet, z.B. für die kollektive Kommunikation und Topologie einer Applikation (siehe nächstes Kapitel).

Attribute sind fest an einen Kommunikator gebunden und lokal an einen Prozeß. Attribute können mit einer Ausnahme, nicht von einem Kommunikator zum nächsten propagiert werden. Nur der MPI-Aufruf `MPI_COMM_DUP` erlaubt das Kopieren von Attributen, und das nur, wenn die dazugehörige Applikation eine spezielle Erlaubnis erteilt. Attribute werden von MPI in den Kommunikatoren „verdeckt“ (*opaque*) gehalten.

Der/die ProgrammierIn kann über einen Schlüssel (*key*) auf die Attribute eines Kommunikators zugreifen (deren Werte erhalten und speichern).

```
MPI_KEYVAL_CREATE(
        copy_fn,                IN
        delete_fn,              IN
        keyval,                 OUT
        extra_state             IN
        )
```

Diese Funktion erstellt einen neuen Attributenschlüssel. Dieser ist in einem Prozeß einzigartig und undurchsichtig für den/die BenutzerIn. Dieser Schlüssel dient in der Zukunft zum Assoziieren eines Attributes und zum Zugriff auf diesen.

Beschreibung der Argumente:

Das erste Argument ist ein Zeiger auf eine Funktion, die aufgerufen wird, wenn ein Kommunikator mit `MPI_COMM_DUP` kopiert werden soll. `copy_fn` kopiert (oder auch nicht) die Attribute des „alten" Kommunikators in den neuen Kommunikator. Diese Kopierfunktion sollte vom Typ MPI_Copy_function sein.

Für ANSI C sieht dieser Typ wie folgt aus:

```
typedef int MPI_Copy_function(
                 MPI_Comm *oldcomm, int *keyval,
                 void     *extra_state,
                 void     *attribute_val_in,
                 void     **attribute_val_out, int *flag
                             );
```

und für FORTRAN 77:

```
FUNCTION COPY_FUNCTION(
             OLDCOMM, KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG
                     )
  INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
            ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

`copy_fn` wird für jeden Schlüsselwert des alten Kommunikators aufgerufen. Ist der Rückgabewert von `flag` gleich Null (0) gesetzt, so wurde der Attributwert nicht in den neuen Kommunikator kopiert, ist `flag`=1 wird der Attributwert mit dem Feld `attribute_val_out` im neuen Kommunikator gesetzt. MPI selber bietet zwei Voreinstellungen für `copy_fn` an:

- MPI_NULL_FN: diese Funktion macht nichts anderes, als keine Attributwerte zu kopieren und den `flag`-Wert Null zurückzugeben.
- MPI_DUP_FN: diese Funktion kopiert die „alten“ Attributwert in den neuen Kommunikator (`flag`=1, und die Werte von `attribute_val_in` finden sich in `attribute_val_out`).

Bemerkung: Das Kopieren von Attributen kann entweder dadurch erfolgen, daß die ganze Informationsstruktur des Attributs kopiert wird oder ein weiterer Verweis auf diese Struktur hergestellt wird (hierbei ist eine Verwaltung dieser Zeiger notwendig).

Analog zu copy_fn wird delete_fn definiert:

```
typedef int MPI_Delete_function(
                    MPI_Comm *comm, int *keyval,
                    void     *attribute_val,
                    void     *extra_state
                               );


FUNCTION DELETE_FUNCTION(
               COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE
                       )
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE
```

delete_fn wird von MPI_COMM_FREE und MPI_ATTR_DELETE aufgerufen, um ein Attribut zu löschen. Auch hier kann MPI_NULL_FN als Funktionszeiger angegeben werden.

```
MPI_KEYVAL_FREE(
         keyval                                   INOUT
         )
```

Der obige MPI-Aufruf ermöglicht die Deallokation eines Schlüsselwertes für ein Attribut. Der Schlüssel bekommt den Wert MPI_KEYVAL_INVALID. Tatsächlich wird dieser Schlüssel erst dann deallokiert, wenn alle Referenzen auf diesen (von anderen Kommunikatoren des Prozesses) gelöscht werden.

```
MPI_ATTR_PUT(
        comm,                                    IN
        keyval,                                  IN
        attribute_val                            IN
        )
```

```
MPI_ATTR_GET(
        comm,                                    IN
        keyval,                                  IN
        attribute_val,                           OUT
        flag                                     OUT
        )
```

Mit den obigen zwei MPI-Operationen kann der/die ProgrammiererIn Attribute (`atribute_val`) setzen bzw. lesen. Die Attribute werden für den (vom) Kommunikator `comm` gesetzt (gelesen). Den Schlüssel `keyval` erhielt die Applikation von `MPI_KEYVAL_CREATE`. Ist in der Funktion `MPI_ATTR_GET` der Wert von `flag`=true so hat `attribute_val` einen Wert, sonst ist kein Attribut mit diesem Schlüssel assoziiert `flag`=false).

```
MPI_ATTR_DELETE(
         comm,                                    IN
         keyval                                   IN
         )
```

Wie der Funktionsname schon ausdrückt, löscht die obige Funktion ein Attribut des Kommunikators `comm`, das mit dem Schlüssel `keyval` verbunden ist.

6 Prozeßtopologien

Dieses Kapitel führt die MPI-Mechanismen für Topologien ein. Topologien sind zusätzliche Attribute die ein Intra-Kommunikator besitzen kann. Eine Topologie bietet einer Gruppe von Prozessen eine „bequeme" Namensgebung an und kann dem darunterliegendem Betriebssystem dabei unterstützen, einzelne Prozesse auf bestimmte Knoten zu „mappen". Topologien werden nicht für Inter-Kommunikatoren angeboten.

Hintergrund solcher Topologien ist die Erfahrung, die bisher bzgl. der Entwicklung von Algorithmen zum Lösen von Problemen gemacht worden ist:

- Prozesse in Gruppen haben einen Rang der linear von 0 bis n-1 verläuft.
- Die meisten Algorithmen und die daraus entstehende Kommunikation, lassen sich nicht linear darstellen und verlangen eine andere Topologie (z.B. ein 2-D- oder 3-D-Gitter).
- Allgemein gesprochen, wird eine logische Prozeßanordnung mit Hilfe eines Graphen dargestellt.
- MPI erlaubt einer Gruppe von Prozessen sich gemäß einem vom/von der BenutzerIn vorgegebenen Graphen anzuorden: Virtuelle Topologie.
- Es liegt im Aufgabenbereich des darunterliegenden Betriebssystems, ob dieses die angebotene virtuelle Topologie auf die tatsächliche physikalische Anordnung der Prozesse anpaßt, um die Kommunikationswege klein zu halten.
- Die Informationen über die virtuelle Topologie werden mit den Kommunikatoren gespeichert. Die Informationen werden mit dem Konzept des „Caching" eingebunden.

Ein Graph besteht aus Knoten und Kanten:

- Die Knoten repräsentieren die Prozesse und die Kanten stellen die Kommunikationsverbindungen dieser Prozesse dar.
- Die Kanten sind nicht gewichtet, d.h. entweder es besteht eine Verbindung oder es besteht keine.
- MPI verbietet nicht die Kommunikation zweier Prozesse, die nicht mit einer Kante verbunden sind. Solche Verbindungen werden in der Anordnung und Namensgebung ignoriert bzw. nicht unterstützt.

6.1 Topologie-Erzeuger

```
MPI_GRAPH_CREATE(
        comm_old,                IN
        nnodes,                  IN
        index,                   IN
        edges,                   IN
        reorder,                 IN
        comm_graph               OUT
        )
```

Im Folgenden werden zwei MPI-Operationen vorgestellt, mit denen Graphen konstruiert werden können.

Das ist eine kollektive Operation, die die Mitglieder der Gruppe im Kommunikator `comm_old` einem neuen Kommunikator zuordnet, dem die Informationen der virtuellen Topologie mitgegeben werden. Die Attribute des Kommunikators ohne Topologie werden nicht mitkopiert. `nnodes` gibt die Anzahl der Prozesse an, die den neuen Graphen bilden sollen. `nnodes` kann kleiner oder gleich der Anzahl der Prozesse aus `comm_old` sein. Ist `reorder`=true, werden die Ränge der Prozesse neu geordnet, sonst nicht.

Die drei Parameter `nnodes`, `index` und `edges` definieren vollständig den zu erstellenden Graphen:

`nnodes` ist die Gesamtanzahl der Knoten,

`index` ist ein Feld von Ganzzahlen mit folgenden Inhalt: das i. Element dieses Feldes speichert die gesamte Anzahl der Nachbarn der ersten i Knoten. Das Feld hat `nnodes` Elemente.

`edges` ist ein Feld von Ganzzahlen, das in aufeinanderfolgenden Reihenfolge die Liste von Nachbarn der Knoten 0, 1, ..., `nnodes`-1 speichert. Die Anzahl der Elemente dieser Liste ist gleich der Anzahl der Kanten im Graph.

Es folgt ein Beispiel mit insgesamt <u>vier</u> Knoten:

Prozeß	Nachbarn	Graph	Eingabeparameter
0	1, 3	1-0-3-2	`nnodes` = 4
1	0		`index` = 2, 3, 4, 6
2	3		`edges` = 1, 3, 0, 3, 0, 2
3	0, 2		

```
MPI_CART_CREATE(
        comm_old,                    IN
        ndims,                       IN
        dims,                        IN
        periods,                     IN
        reorder,                     IN
        comm_cart                    OUT
        )
```

Der obige kollektive Aufruf erzeugt ein Gitter der Dimension `ndims`, wobei das Feld `dims` angibt, wieviele Prozesse in jeder Dimensionsichtung das Gitter „lang“ ist. Das Feld `periods`, das `ndims` groß ist, gibt an, ob die entsprechende Dimension periodisch ist (true) oder nicht (false).

In kartesischen Topologien beginnt die Nummerierung mit Null (0). Die Prozesse werden spaltenweise durchnummeriert. Die Beziehung zwischen Gruppenrang und Koordinaten sieht im Fall eines (2×2)-Gitters wie folgt aus (Spalte, Reihe):

```
coord( 0, 0 ): Rang 0
coord( 0, 1 ): Rang 1
coord( 1, 0 ): Rang 2
coord( 1, 1 ): Rang 3
```

```
MPI_DIMS_CREATE(
        nnodes,                    IN
        ndims,                     IN
        dims                       OUT
        )
```

Der obige Befehl versucht eine optimale Verteilung der Prozesse in einem kartesischen Gitter zu erreichen. `dims` ist ein Feld, mit der Angabe, wie „lang“ die Dimension ist (in Einheiten von Prozessen).

`dims` vorher	Funktionsaufruf	`dims` nachher
(0, 0)	`MPI_DIMS_CREATE( 6, 2, dims )`	(3, 2)
(0, 0)	`MPI_DIMS_CREATE( 7, 2, dims )`	(7, 1)
(0, 3, 0)	`MPI_DIMS_CREATE( 6, 3, dims )`	(2, 3, 1)
(0, 3, 0)	`MPI_DIMS_CREATE( 7, 3, dims )`	fehlerhaft

Ist `nnodes` kein Vielfaches von $\prod_{i,dims[i]\neq 0} dims[i]$, tritt ein Fehler auf.

6.2 Informationen über Topologien

```
MPI_TOPO_TEST(
        comm,                                      IN
        status                                     OUT
        )
```

Wie der Titel schon sagt, erhält die Applikation mit den folgenden MPI-Befehlen Informationen über virtuelle Topologien.

Der erste Befehl ermittelt bei einem Kommunikator `comm`, ob dieser in einer bestimmten Topologie (MPI_GRAPH, MPI_CART) aufgebaut ist oder nicht (MPI_UNDEFINED). `status` kann einen der obigen Werte annehmen.

Mit den nächsten zwei Befehle, erhält der/die ProgrammiererIn die Informationen, die während des Aufrufes von `MPI_GRAPH_CREATE` eingegeben wurden.

```
MPI_GRAPHDIMS_GET(
          comm,                                   IN
          nnodes,                                 IN
          nedges                                  OUT
          )
```

```
MPI_GRAPH_GET(
          comm,                                   IN
          maxindex,                               IN
          maxedges,                               IN
          index,                                  OUT
          edges                                   OUT
          )
```

Der erste Befehl ermittelt die Anzahl der Knoten `nnodes` (Prozesse) und Kanten `nedges` eines Graphen in einem Kommunikator `comm`. Die Rückgabewerte können für den zweiten Befehl zur richtigen Dimensionierung der Felder `index` und `edges` dienen.D ie Argumente für den zweiten Befehl haben die gleiche Bedeutung, wie unter dem MPI-Aufruf `MPI_GRAPH_CREATE`.

MPI_CARTDIM_GET(

comm,	IN
ndims	OUT
)	

coords ist die Koordinate des aufrufenden Prozesses in der kartesischen Struktur (coords ist ein Feld der Länge ndims).

```
MPI_CART_RANK(
        comm,                    IN
        coords,                  IN
        rank                     OUT
        )
```

```
MPI_CART_COORDS(
        comm,                    IN
        rank,                    IN
        maxdims,                 IN
        coords                   OUT
        )
```

Mit den obigen zwei MPI-Aufrufen ist es möglich, die Beziehung zwischen dem Rang (`rank`) eines Prozesses in einem Kommunikator (`comm` und seinen Koordinaten (`coords`) im kartesischen Gitter zu ermitteln.

Mit dem ersten Befehl wird der Rang (`rank`) eines Prozesses mit den Koordinaten `coords` (siehe vorherigen Befehl) ermittelt. Das „inverse“ erreicht der zweite Aufruf.

Durch Angabe des Ranges, werden dem Aufrufer im Feld `coords`, der Länge `maxdims` die Koordinaten des Prozesses übergeben.

```
MPI_GRAPH_NEIGHBORS_COUNT(
        comm,                 IN
        rank,                 IN
        nneighbors            OUT
        )
```

```
MPI_GRAPH_NEIGHBORS(
        comm,                 IN
        rank,                 IN
        mneighbors,           IN
        neighbors             OUT
        )
```

„Nachbarschaftliche“ Informationen über die Nachbarn eines Prozesses in einem Graphen, erhält der aufrufende Prozeß mit den obigen zwei MPI-Befehlen.

Durch Angabe des Prozeßranges `rank` wird die Anzahl der Nachbarn `nneighbors` zurückgegeben. Der zweite Befehl gibt zusätzlich die Ränge `neighbors` der Nachbarn zurück. Das Feld `neighbors` besitzt `mneighbors` Elemente.

6.3 Weitere Topologie-Aufrufe

```
MPI_CART_SHIFT(
        comm,                    IN
        direction,               IN
        disp,                    IN
        rank_source,             OUT
        rank_dest                OUT
        )
```

Möchte ein Prozeß in einem kartesischen Gitter, Daten von einem Nachbarn zum nächsten Nachbarn bewegen (der Prozeß, wie auch die zwei Nachbarn müssen sich in der gleichen Dimension befinden), so bietet MPI einem solchen Prozeß Hilfestellung an. Zum Bewegen („Shiften") der Daten eignet sich der MPI-Aufruf `MPI_SENDRECV`. Dieser benötigt die Rangnummer des Senders und Empfängers der Daten - in unserem Fall die Rangnummern der Nachbarn. MPI bietet zur Ermittlung dieser Ränge die obige Funktion an.

Diese Funktion ermittelt die Ränge der Nachbarn (`rank_source` und `rank_dest`) im Kommunikator `comm`, die vom Aufrufer `disp` „Einheiten" in der Richtung `direction` entfernt sind. `direction` kann sowohl positiv als auch negativ sein.

Bemerkung: Ist die angegebene Richtung bzw. Dimension periodisch wird an den Rändern zirkuliert, ansonsten wird als Rang der Wert MPI_PROC_NULL zurückgegeben.

Als Beispiel soll eine 2-D-Gitter dienen, in dem die i. Spalte um i Schritte entlang der Spalte (vertikal) verschoben werden soll:

```
....
C Prozessrang herausfinden
      CALL MPI_COMM_RANK( comm, rank, ierr )
C Kartesische Koordinaten ermitteln
      CALL MPI_CART_COORDS( comm, rank, maxdims, coords, ierr )
C Welche sind meine Nachbarn?
      CALL MPI_CART_SHIFT( comm, 1, coords(2), source, dest, ierr )
C Spalte verschieben
      CALL MPI_SENDRECV_REPLACE( A, 1, MPI_REAL, dest, 0, source, 0,
     +                           comm, status, ierr )
```

```
MPI_CART_SUB(
        comm,                        IN
        remain_dims,                 IN
        newcomm                      OUT
        )
```

Ist eine kartesische Topologie mit `MPI_CART_CREATE` entworfen worden, lassen sich mit dem obigen Befehl die Prozesse in Untergruppen des mit `MPI_CART_CREATE` enstandenen Kommunikators einteilen. Hierbei gibt das Feld `remain_dims` an, welche Dimension in der Untergruppen „überleben" wird. Im Feld `newcomm` werden die Handler zu den neuen Kommunikatoren zurückgegeben.

```
MPI_CART_MAP(
          comm,                                        IN
          ndims,                                       IN
          dims,                                        IN
          periods,                                     IN
          newrank                                      OUT
          )
```

```
MPI_GRAPH_MAP(
          comm,                                        IN
          nnodes,                                      IN
          index,                                       IN
          edges,                                       IN
          newrank                                      OUT
          )
```

Die beiden obigen Routinen sind nicht dafür gedacht, um von einem/r AnwenderIn aufgerufen zu werden. Diese Routinen berechnen eine „optimale“ Anordnung der Prozesse auf der physikalischen Maschine. Die Argumente sind die gleichen wie unter `MPI_CART_CREATE` bzw. `MPI_GRAPH_CREATE`.

7 Die MPI Umgebung

In diesem Kapitel werden die Befehle behandelt, die es ermöglichen Parameter über die verwendete MPI Implementation zu erhalten und ggf. zu setzen. Zusätzlich werden die Befehle zum Ein- und Austritt der MPI Programmumgebung vorgestellt.

7.1 Informationen über die Implementation

Beim Initialisieren von MPI werden dem Kommunikator MPI_COMM_WORLD einige Attribute mitgegeben. Die können mit `MPI_ATTR_GET` erfragt werden. Diese Attribute zu löschen oder zu ändern führt zu einem Fehler. Folgende Attribute werden definiert (die jeweiligen Anbieter einer Implementation können weitere Attribute definieren):

MPI_TAG_UB : Die höchst mögliche Nachrichtenkennung. Diese fängt bei 0 an und geht bis inklusive dem obigen Wert, mindestens 32767. Dieses Attribut ist für alle Prozesse in MPI_COMM_WORLD gleich.

MPI_HOST : Die Rangnummer des Prozesses im HOST. MPI spezifiziert nicht, was ein HOST bedeutet. Ist ein solcher HOST nicht vorhanden hat das Attribut den Wert MPI_PROC_NULL.

MPI_IO : Dieses Attribut gibt den Rang des Prozesses zurück, welcher in der jeweiligen Programmiersprache die Standardmöglichkeiten für E/A bietet. Trifft dies für keinen Prozeß zu, wird der Wert MPI_PROC_NULL zurückgegeben. Trifft dies für mehr als einen Prozeß zu, wird nur ein Rang von mehreren zurückgegeben.

MPI_WTIME_IS_GLOBAL : Werden alle Uhren der Prozesse im Kommunikator MPI_COMM_WORLD synchronisiert hat dieses Attribut den Wert 1, sonst 0. Die Uhren sind dann synchronisiert, wenn der Unterschied zwischen zwei Uhren kleiner als die Hälfte der *round-trip*-Zeit (der Länge 0) beträgt. Wird die Zeit am Prozeß kurz vor der Versendung einer Nachricht gemessen, so muß diese kleiner sein, als die Zeit am anderen Prozeß gleich nach dem Empfang der gleichen Nachricht.

Die obigen Attribute dürfen sich in der Zeit von der Initialisierung bis zur Komplettierung nicht ändern.

```
MPI_GET_PROCESSOR_NAME(
          name,                                      OUT
          resultlen                                  OUT
          )
```

Die obigen Funktion gibt dem aufrufenden Prozeß die Bezeichnung (`name`) des Prozessors zurück auf dem der Prozeß läuft. Der zurückgegebene String ist `resultlen` Zeichen lang. Die maximale Länge kann MPI_MAX_PROCESSOR_NAME Zeichen lang sein. MPI verlangt von der Implementierung nicht, daß dieser Aufruf relevante Informationen zurückgibt. Sinnvoll ist eine solche Information beim Migrieren von Prozessen von einem Knoten zu einem anderen.

7.2 Fehler und deren Behandlung

Abhängig von der MPI Implementation kann diese Fehler abfangen. In diesem Fall spricht MPI von *MPI exception* (MPI Ausnahme). Jeder Prozeß kann lokal für einen Kommunikator eine Funktion bereitstellen, die eine solche Ausnahme behandelt, einen Errorhandler. Mit dieser Funktion werden Ausnahmen behandelt, die im Rahmen dieses Kommunikators eine Kommunikation ausführen. Ein Kommunikator vererbt seinen Errorhandler an die aus ihm stammenden Kommunikatoren. In der Voreinstellung werden zwei Errorhandler angeboten (die Anbieter können weitere Errorhandler bereitstellen):

`MPI_ERRORS_ARE_FATAL` : Dieser Handler verlangt vom Programm, daß dieses von allen ausführbaren Prozessen aussteigt. Dieser Errorhandler wird gleich nach der Initialisierung von MPI_COMM_WORLD dem Kommunikator assoziiert.

`MPI_ERRORS_RETURN` : Dieser Handler hat keine Funktion, außer die, dem/der BenutzerIn den Fehlercode zurückzugeben.

<u>Bemerkung:</u> Es ist sinnvoll nicht den Rückgabewert jedes MPI-Aufrufes zu prüfen. Statt dessen sollte ein umfangreicher Errorhandler implementiert werden.

<u>Vorsicht:</u> MPI verlangt nicht, daß die Implementation nach einem Fehler weiterlaufen kann.

```
MPI_ERRHANDLER_CREATE(
        function,                    IN
        errhandler                   OUT
        )
```

Dieser Befehl registiert eine benutzereigene Funktion als Errorhandler. Diese Funktion wird einem *opaque object* (`errhandler`) zugeordnet. Konkret muß der Errorhandler in ANSI C wie folgt aussehen:

```
typedef void (MPI_Handler_function)(MPI_Comm *, int *, ...);
```

Das erste Argument zeigt auf den Kommunikator der verwendet wird und das zweite Argument ist die eigentliche Fehlerkennung. Abhängig von der MPI Implemention können weitere Argumente übergeben werden (diese müssen vom Anbieter ausführlich dokumentiert sein).

```
MPI_ERRHANDLER_SET(
        comm,                           IN
        errhandler                      IN
        )
```

```
MPI_ERRHANDLER_GET(
        comm,                           IN
        errhandler                      OUT
        )
```

Die erste Funktion verbindet den Errorhandler `errhandler` mit dem Kommunikator `comm`. Die zweite Funktion ermittelt den aktuellen Errorhandler der mit dem Kommunikator `comm` verbunden ist.

```
MPI_ERRHANDLER_FREE(
          errhandler                              IN
          )
```

Dieser Aufruf deallokiert einen Errorhandler. Die Deallokation erfolgt erst dann, wenn alle damit verbundenen Kommunikatoren deallokiert sind.

```
MPI_ERROR_STRING(
        errorcode,                         IN
        string,                            OUT
        resultlen                          OUT
        )
```

Die obigen Funktion gibt den Text (`string`), der mit der Fehlerkennung oder -klasse `errorcode` verbunden ist zurück. Der Text ist `resultlen` Zeichen lang, kann aber bis zu MPI_MAX_ERROR_STRING Zeichen lang sein.

```
MPI_ERROR_CLASS(
        errorcode,                              IN
        errorclass                              OUT
        )
```

Die Fehlerkennungen sind von der MPI Implementation abhängig und genügen folgender Bedingung:

$$0 = \text{MPI\_SUCESS} < \text{MPI\_ERR...} \leq \text{MPI\_ERR\_LASTCODE}$$

Um die Klassifizierung der Fehlerkennungen zu erleichtern, bietet MPI Fehlerklassen an. Die obige Funktion ordnet eine Fehlerkennung einer zugehörigen Fehlerklasse zu. Folgende Fehlerklassen sind definiert:

MPI_SUCCESS	No error
MPI_ERR_BUFFER	Invalid buffer pointer
MPI_ERR_COUNT	Invalid count argument
MPI_ERR_TYPE	Invalid datatype argument
MPI_ERR_TAG	Invalid tag argument
MPI_ERR_COMM	Invalid communicator
MPI_ERR_RANK	Invalid rank
MPI_ERR_REQUEST	Invalid request (handle)
MPI_ERR_ROOT	Invalid root
MPI_ERR_GROUP	Invalid group
MPI_ERR_OP	Invalid operation
MPI_ERR_TOPOLOGY	Invalid topology
MPI_ERR_DIMS	Invalid dimension argument
MPI_ERR_ARG	Invalid argument of some other kind
MPI_ERR_UNKNOWN	Unknown error
MPI_ERR_TRUNCATE	Message truncated on receive
MPI_ERR_OTHER	Known error not in this list
MPI_ERR_INTERN	Internal MPI (implementation) error
MPI_ERR_IN_STATUS	Error code is in status
MPI_ERR_PENDING	Pending request
MPI_ERR_LASTCODE	Last error code

7.3 Zeiten und Startups

MPI_WTIME()

MPI_WTICK()

Die obigen Funktionen beziehen sich auf Zeitmessungen. Die erste Funktion gibt die verstrichenen Sekunden seit einem Zeitpunkt in der Vergangenheit an. MPI garantiert, das dieser Zeitpunkt in der Vergangenheit während der Lebenszeit eines Prozesses konstant bleibt.

Der zweite Aufruf gibt die verstrichenen Sekunden zwischen zwei Uhrenschlägen an, d.h. die Auflösung von MPI_WTIME.

MPI_INIT()

MPI_FINALIZE()

Der erste Aufruf muß vor allen anderen MPI-Aufrufen getätigt werden (mit Ausnahme von MPI_INITIALIZED). Ein mehrmaliges Aufrufen verursacht Fehler. MPI überläßt der jeweiligen Implementation die Funktionalität hinter diesem Aufruf.

Der zweite Aufruf „säubert“ alle Zustände von MPI. Danach darf kein MPI-Aufruf erfolgen. Der/Die BenutzerIn hat dafür Sorge zu tragen, daß die Kommunikation vor dem obigen Aufruf ordnungsgemäß beendet worden ist.

```
MPI_INITIALIZED(
        flag                                      OUT
        )
```

Der Flag (`flag`) ist true wenn der Prozeß `MPI_INIT` aufgerufen hat, sonst false. Das ist die einzige Routine, die vor `MPI_INIT` aufgerufen werden darf.

```
MPI_ABORT(
        comm,                                   IN
        errorcode                               IN
        )
```

Diese Routine versucht alle Prozesse im Kommunikator `comm` zu stoppen und der Umgebung, die das Programm aufgerufen hat, den Rückgabewert `errorcode` zurückzugeben. MPI verlangt von den Implementationen, daß zumindest für den Kommunikator MPI_COMM_WORLD die obige Routine definiert ist. Die Implementation kann das Argument `comm` ignorieren und sich auf MPI_COMM_WORLD beziehen. Das Argument für den Kommunikator wird angeboten, um für spätere Erweiterungen von MPI verwendet zu werden (dynamische Prozeßkonfiguration).

A Übungsaufgaben

A.1 Matrix-Vektor-Multiplikation

Es soll mit Hilfe von N Prozessoren/Knoten ein Programm geschrieben werden, das eine Matrix-Vektor-Multiplikation ausführt:

Hierbei betrachten wir eine $N \times N$-Matrix, mit $A = (a_{ij})$ und einen N-Vektor, mit $x = (x_j)$ $(i, j = 1 \ldots N)$. Eine Matrix-Vektor-Multiplikation kann wie folgt geschrieben werden:

$$y = Ax = (y_i) \quad \text{mit} \quad y_i = \sum_{j=1}^{N} a_{ij} x_j \quad (j = 1 \ldots N)$$

Zur sequentiellen Berechnung der obigen Operation werden $2N^2 - N$ Schritte benötigt: N Multiplikationen und $N-1$ Additionen. Bei Verwendung eines N-zelligen linearen Feldes von Prozessoren werden nur $2N - 1$ Rechenschritte benötigt. Der Algorithmus geht wie folgt:

Die Elemente x_j werden Schritt für Schritt von links in das obige Feld eingegeben (beginnend mit x_1, x_2, ...) und die Matrixelemente a_{ij} dienen als Eingabe oberhalb des Feldes (siehe unten, mit $N = 4$):

$$\begin{array}{cccccccc}
 & & & & & & & a_{44} \\
 & & & & & & a_{34} & a_{43} \\
 & & & & & a_{24} & a_{33} & a_{42} \\
 & & & & a_{14} & a_{23} & a_{32} & a_{41} \\
 & & & & a_{13} & a_{22} & a_{31} & \cdot \\
 & & & & a_{12} & a_{21} & \cdot & \cdot \\
 & & & & a_{11} & \cdot & \cdot & \cdot \\
 & & & & \Downarrow & \Downarrow & \Downarrow & \Downarrow \\
x_4 & x_3 & x_2 & x_1 \Longrightarrow & y_1 & y_2 & y_3 & y_4
\end{array}$$

A.2 Matrix-Matrix-Multiplikation

Der im vorherigen Kapitel vorgestellte Algorithmus einer Matrix-Vektor-Multiplikation kann leicht auf eine Matrix-Matrix-Multiplikation im zweidimensionalen Fall erweitert werden. Sind zwei Matrizen $A = (a_{ij})$ und $B = (b_{ij})$ gegeben, so läßt sich ihre Multiplikation wie folgt schreiben:

$$C = AB = (c_{ij}) \quad \text{mit} \quad c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$$

Wird ein $N \times N$-Feld von Knoten verwendet läßt sich die Multiplikation mit $3N - 2$ Schritten ausführen. Der Algorithmus geht wie folgt:

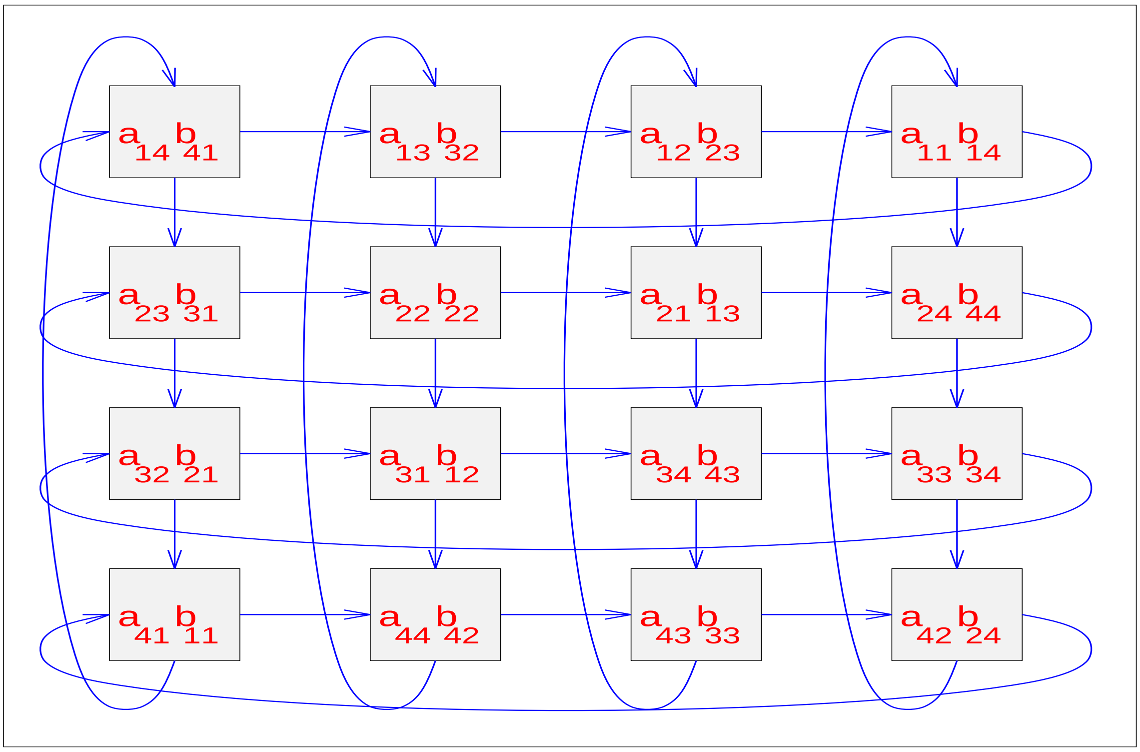
Die Reihe i von A dient als Eingabe für die Reihe i des obigen Feldes (beginnend mit a_{i1} beim Schritt i). Ähnlich dient die Spalte j von B als Eingabe für die Spalte j des Feldes (beginnend mit b_{1j}) beim Schritt j). Auf der nächsten Seite ist der Initialzustand dargestellt (mit $N = 4$).

<u>Vorsicht:</u> In der Matrix-Matrix-Multiplikation wird die Matrix A „von links“ eingeben und nicht wie in der Vektor-Matrix-Multiplikation „von oben“. In der Vektor-Matrix-Multiplikation war der resultierende Vektor transponiert, was bei einem Vektor nicht weiter beachtet werden muß. In diesem Fall ergibt der Algorithmus die „richtige“ Matrix und nicht die transponierte.

$$
\begin{array}{cccccccccccc}
 & & & & & & & & & & & b_{44} \\
 & & & & & & & & & & b_{43} & b_{34} \\
 & & & & & & & & & b_{42} & b_{33} & b_{24} \\
 & & & & & & & & b_{41} & b_{32} & b_{23} & b_{14} \\
 & & & & & & & & b_{31} & b_{22} & b_{13} & \cdot \\
 & & & & & & & & b_{21} & b_{12} & \cdot & \cdot \\
 & & & & & & & & b_{11} & \cdot & \cdot & \cdot \\
 & & & & & & & & \Downarrow & \Downarrow & \Downarrow & \Downarrow \\
 & & & a_{14} & a_{13} & a_{12} & a_{11} & \Longrightarrow & y_{11} & y_{12} & y_{13} & y_{14} \\
 & & a_{24} & a_{23} & a_{22} & a_{21} & \cdot & \Longrightarrow & y_{21} & y_{22} & y_{23} & y_{24} \\
 & a_{34} & a_{33} & a_{32} & a_{31} & \cdot & \cdot & \Longrightarrow & y_{31} & y_{32} & y_{33} & y_{34} \\
a_{44} & a_{43} & a_{42} & a_{41} & \cdot & \cdot & \cdot & \Longrightarrow & y_{41} & y_{42} & y_{43} & y_{44}
\end{array}
$$

Matrix-Matrix-Multiplikation mit $N = 4$

Der Algorithmus läßt sich um eine konstanten Faktor verbessern, wenn bei jedem Schritt eine Operation stattfindet, somit läßt sich die Multiplikation mit N Schritten ausführen:



Ein 4×4 Torus für die Matrix-Matrix-Multiplikation

B Literatur zu MPI

Alle hier aufgeführten Dokumente sind über den lokalen WWW-Server am Rechenzentrum erhältlich. Für das Projekt „paralleles und numerisches Rechnen“ existiert derzeit ein eigener Server:

http://alphamor.zdv.uni-tuebingen.de/

Die Dokumente bzgl. MPI befinden sich an folgender Adresse:

http://alphamor.zdv.uni-tuebingen.de/PARALLEL/mpi/mpi.html

- „A Proposal for a User-Level, Message Passing Interface in a Distributed Memory Environment“, J. Dongarra, R. Hempel, A. J. G. Hey und D. W. Walker, *Engineering Physics and Mathematics Division, Mathematical Sciences Section - Oak Ridge National Laboratory, ORNL/TM-12231*, Juni 1993
- „The design of a standard message-passing interface for distributed memory concurrent computers“, D. W. Walker, *Parallel Computing*, Vol. 20, No. 4, 657-673, April 1994
- „MPI: A Message-Passing Interface Standard“, Message Passing Interface Forum, *Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN*, 5. Mai 1994
- „The MPI Message Passing Interface Standard“, L. Clark, I. Glendinning und R. Hempel, *ftp://par.soton.ac.uk/pub/mpi/paper.ps*, 1994
- „A Model Implementation of MPI“, N. Doss, W. Gropp, E. Lusk und A. Skjellum, *Mathematics and Computer Science Division, Argonne National Laboratory MCS-P393-1193*, 1993

- „A Test Implementation of the MPI Draft Message-Passing Standard“, W. Gropp und E. Lusk, *Mathematics and Computer Science Division, Argonne National Laboratory, ANL-92/47*, Dezember 1992
- „An Efficient Implementation of MPI“, H. Franke, P. Hochschild, P. Pattnaik und M. Snir, *Proceedings of the 1994 International Converence on Parallel Processing*, 1994
- „Writing Libraries in MPI“, A. Skjellum, N. E. Doss und P. V. Bangalore, *Proceedings of the Scalable Parallel Libraries Conference, IEEE Computer Society Press*, 166-173, Oktober 1993
- „Performance Analysis of MPI Programs“, E. Karrels und E. Lusk, *ftp://info.mcs.anl.gov/pub/mpi/misc/heath.ps*, 1994
- „MPI++: Issues and Features“, P. V. Bangalore, N. E. Doss und A. Skjellum, *OON-SKI '94*, 323-338, 1994
- „MPI-1, ein Industriestandard für den Nachrichtenaustausch“, L. Servissoglou, *BI 95/7+8*, Seiten 7-11, Juli 1995