1. Einführung

1.1 Über dieses Tutorial ...

Gegenstand dieses Tutorials ist MPI. In diesem Tutorial wird anhand von Beispielen erklärt, wie Programme mit MPI geschrieben werden. Dabei werden Beispiele die erklärten Befehle und Konzepte verdeutlichen. Über das gesamte Tutorial wird sich ein laufendes Beispiel ziehen, anhand dessen einige Konzepte von MPI verdeutlicht werden. Mehr dazu in Abschnitt 1.6. Das Tutorial ist in zehn Kapitel gegliedert:

- Dieses erste Kapitel wird MPI und das Modell von MPI vorstellen. Am Ende wird auf das laufende Beispiel eingegangen.
- Im zweiten Kapitel wird anhand einiger Beispiels vorgestellt, wie sich MPI während der Ausführung verhält.
- Das dritte Kapitel widmet sich dem Senden und Empfangen von Nachrichten.
- Die Funktionen von MPI, von denen man wirklich profitiert, werden im vierten Kapitel vorgestellt.
- Im fünften Kapitel geht es um die Anordnung von Prozessen in Topologien.
- Um erweiterte Datentypen geht es im sechsten Kapitel.
- Das siebente Kapitel dreht sich um die Zusammenfassung von Prozessen zu Kommunikatoren und Gruppen.
- Im achten Kapitel wird die Fehlerbehandlung in MPI-Programmen näher beleuchtet..
- Das neunte Kapitel wird sich den Dingen widmen, die in den vorherigen Kapiteln nicht behandelt wurden.
- Das letzte Kapitel bietet als Referenz eine Übersicht über alle Funktionen Konstanten und Typen von MPI.

In diesem Tutorial werden die üblichen Konventionen benutzt :

- Code wird in `Courier New` dargestellt.
- **Fetter Text** stellt in der Referenz Bezeichner hervor.
- Drei Punkte `...` in Funktionsbeschreibungen bedeuten, daß an dieser Stelle Parameter folgen können, die von der Implementierung von MPI auf der Zielmaschine abhängen.
- Drei vertikale Punkte bedeuten in Code-Beispielen, daß an ihrer Stelle weiterer Code steht, der nicht mit aufgeführt wird.

Dieses Tutorial gibt es auch als Druckversion in einer PDF-Datei und als HTML-Datei (gepackt als zip-Archiv).

1.2 Was ist MPI?

MPI ist eine Abkürzung für message passing interface. Wörtlich übersetzt bedeutet dies soviel wie "Nachrichtenübertragungsschnittstelle". Im engeren Sinne ist MPI eine Bibliothek von Funktionen, die es ermöglichen, Nachrichten zwischen Prozessen auszutauschen. MPI ist also keine Programmiersprache, sondern eine Sammlung von Funktionen und Konstanten, die in Programmiersprachen eingebunden werden kann.

Eine weitverbreitete Implementation von MPI ist MPICH. Es gibt auch Varianten mit MPE, einer Schnittstelle zu einem X-Server, mit der Grafikausgaben parallelisiert werden können und mit der die Arbeitsweise von MPI visualisiert werden können. Zum Beispiel kann man damit feststellen, wieviel Zeit MPI auf Broadcasts oder Sends verwendet hat.

1.3 Wer benutzt MPI?

MPI ist ein Quasi-Standard bezüglich Interprozeßkommunikation. Dementsprechend (und weil es einfach zu implementieren ist) wird es überall dort eingesetzt, wo man mit einer Parallelisierung der Algorithmen einen Geschwindigkeitszuwachs erwarten kann. Physiker benutzen MPI, um physikalische Modelle, die als Differentialgleichungen aufgestellt worden sind, durchzurechnen. Als Beispiel sei da das Stichwort Wärmeleitungsgleichung genannt. In der Mathematik bringt das parallele Lösen von Interpolationsaufgaben und von Differentialgleichungssystemen von Ordnungen jenseits der 10000 einen enormen Geschwindigkeitsgewinn. Als Stichworte seien Runge-Kutta-Verfahren und Newton-Cotes-Formeln genannt. Auch in der linearen Algebra leistet Parallelisierung gute Dienste. Das Gauß-Jordan-Verfahren und die simple Ermittlung eines Durchschnitts läßt sich gut parallelisieren. Letzteres wird in diesem Tutorial auch als Programmbeispiel auftauchen.

Einer der Vorteile von MPI ist, daß man nur die Funktionen zu kennen braucht, die man auch wirklich benötigt. MPI hat rund 130 Funktionen. Um eine vollständige Kommunikation zwischen Prozessen zu erreichen, benötigt

man jedoch nur sechs Stück. Der Rest ist sozusagen "zur Verfeinerung". In diesem Tutorial werden alle Funktionen in entsprechenden Gruppen vorgestellt.

1.4 MPI-Implementierungen

Im wesentlichen gibt es zwei Implementierungen von MPI. Eine für Fortran 77 und eine für C. Da C eine Teilmenge von C++ ist, sollte sich MPI auch in C++ verwenden lassen. Dies ist von besonderem Interesse, da mit dem Klassenkonzept in C++ eine Kapselung von MPI in Klassen möglich ist. Die Implementierung in Fortran 77 sollte sich wegen der Teilmengenbeziehung auch in Fortran 90 verwenden lassen. Es ergeben sich in Fortran einige Unterschiede zu C:

- In C werden diverse Parameter (insbesondere Puffer und Handles) als Referenz übergeben. Dies geschieht mit dem üblichen Referenzierungsoperator `&`. In Fortran ist dies nicht nötig, da die Typbibliothek an den ensprechenden Stellen eine Referenzierung vorschreibt und der Compiler dies automatisch berücksichtigt. Solche Speicherbereiche werden in der Signatur der Funktionen als `<type> para(*)` ausgezeichnet, also als typisierte dynamische Reihungen.
- Fehlerwerte werden in C von den Funktionen selbst zurückgegeben. In Fortran haben alle Funktionen als letzten Parameter eine Variable, in die der Fehlerwert eingetragen wird.
- C ist strenger, was Typen angeht. MPI-Funktionen benötigen in C bei fast allen Funktionen Parameter, deren Typ speziell von MPI festgelegt wird, z.B. `MPI_Comm`. In Fortran stehen an diesen Stellen einfach Parameter vom Typ `INTEGER`. Zwar sind diese MPI-Datentypen Aufzählungstypen, also Ausschnittstypen des C-Typs `int`, dennoch sollten die richtigen Typen anstelle der `int`-Repräsentationen verwendet werden.

Allen Implementierungen ist gemeinsam, daß die Funktionen, Konstanten und Datentypen stets mit `MPI_` beginnen. In diesem Tutorial sind die Beispiele in C geschrieben.

Meistens wird man ein MPI-Programm nicht auf der Maschine ausführen wollen, auf der man arbeitet. Üblicherweise wird dazu der Quellcode eines MPI-Programms per `ftp` oder auch `sftp` auf die Zielmaschine übertragen. Per `telnet` (oder auch aus Sicherheitsgründen per `ssh`) loggt man sich auf der Zielmaschine ein. Dort wird die Übersetzung eines MPI-Programms mit einem Befehl gestartet, der so aussieht:

```
mpicc myProg.c -o myProg
```

`mpicc` ist hierbei der Name eines Skripts, welches den eigentlichen C-Übersetzer `cc` mit den richtigen Parametern und den Binder mit den richtigen Bibliotheken versorgt. Dabei sollte aber beachtet werden, daß sich die Bibliotheken eventuell im Pfad befinden muß. Es gibt ein weiteres Skript, `mpiCC`, welches dem Programmierer größere Freiheiten einräumt, da der benutzte Übersetzer `g++` nicht so streng ist. Aber dann schlägt z.B. die Funktion `exit()` fehl. Für einen Fortran-Compiler heißt das Skript `mpif77`. Ausgeführt wird das übersetzte Programm mit einem Befehl, der so aussieht:

```
mpirun -np n myProg
```

`mpirun` ist ein Skript, welches die Ausführung startet, wobei *n* die Anzahl der Prozesse ist, auf denen das Programm gestartet wird. Die Anzahl der Prozesse, auf denen das MPI-Programm läuft, steht also schon vor der Programmausführung fest. Dies ist einer der Gründe, warum man MPI-Programme skalierbar schreiben sollte. Es gibt auch Maschinen, auf denen das Programm nicht mit einem Skript gestartet wird, sondern direkt:

```
myProg -np n
```

Der nächste Abschnitt erklärt das Modell der Programmausführung.

1.5 Das Modell von MPI

MPI folgt dem Prozessor-Speicher-Modell des message-passing-model. In diesem Modell hat jeder Prozeß seinen eigenen Adreßraum. Die Prozesse werden entweder auf die vorhandene Hardware abgebildet oder mit einem Emulator auf einem Prozessor gestartet werden. Sie können nur über Nachrichten kommunizieren. Diese Kommunikation auf der untersten Ebene muß vom Betriebssystem bereitgestellt werden. Ein Programm, welches zur Ausführung mit MPI bestimmt ist, wird auf allen Prozessen gestartet. Jeder Prozeß verarbeitet also das gleiche Programm. Wie die Prozesse einander unterscheiden können, zeigt das Beispiel in Kapitel 2. Variablen, die in

einem MPI-Programm deklariert sind und von Prozeß *x* belegt werden, sind aber nicht automatisch im Prozeß *y* belegt, auch wenn sie dort denselben Namen haben, denn die Prozesse haben ja verschiedene Adreßräume. Zentraler Punkt bei der MPI-Programmierung muß also der Nachrichtenaustausch sein. Nachrichten werden in MPI ausgetauscht, indem den entsprechenden Operationen mitgeteilt wird, wo die Nachricht beginnt `(address)`, aus wievielen Teilen sie besteht `(count)` und welchen Datentyp die Elemente besitzen `(datatype)`. Dieses `(address, count, datatype)`-Modell durchzieht alle MPI-Funktionen, die etwas mit Senden oder Empfangen von Nachrichten zu tun haben. Desweiteren haben alle solche Funktionen einen `source`-Parameter und/oder einen `dest`-Parameter, welche die Quelle und/oder das Ziel der Nachricht angeben. Daneben gibt es diverse Hilfsfunktionen, die das Senden und Empfangen von bestimmten Variablen einfacher gestalten. Zum Beispiel kann man neue Typen definieren und Prozesse zu Gruppen zusammenfassen.

1.6 Das laufende Beispiel

Wie eingangs schon erwähnt, gibt es in diesem Tutorial ein laufendes Beispiel, an dem diverse Konzepte von MPI verdeutlicht werden. Das Beispiel dient zum numerischen Lösen von steifen und nichtsteifen gewöhnlichen Differentialgleichungen. Das Programm wurde von Scott D. Cohen, Alan C. Hindmarsh und George D. Byrne an der Universität von Kalifornien am Lawrence Livermore National Laboratory im Auftrag der Energieministeriums der VSA geschrieben. Man beachte die Kopierrechthinweise in der Datei `README`. Für genaue Erklärungen wird an dieser Stelle auf den Quelltext verwiesen.

Das Beispiel steht als zip-Archiv hier zum Download bereit und muß auf der Zielmaschine mittels eines makefiles übersetzt werden. Am Ende jedes Kapitels wird die Anwendung dessen, was Stoff des Kapitels war, anhand dieses Beispiels gezeigt. Dazu wird ein Ausschnitt aus dem Quell-Code präsentiert.

2. Erste Beispiele

2.1 Einleitende Bemerkungen

In diesem Kapitel wird anhand von einfachen Beispielen die Arbeitsweise von MPI vorgestellt. Das erste Beispiel ist eine Variante von "Hello World". Anhand dieses Beispiels wird auch der Übersetzungsprozeß erläutert.

2.2 Der Quellcode

```
#include <stdio.h>                    // Zeile A
#include <mpi.h>                      // Zeile B

main (int argc, char* argv[]) {

 int myrank;
 int size;

 MPI_Init(&argc, &argv);                    // Zeile C
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);    // Zeile D
 MPI_Comm_size(MPI_COMM_WORLD, &size);      // Zeile E

 printf("Hello World. I'm process %d of %d processes.", myrank, size);

 MPI_Finalize();                            // Zeile F
 exit(0);

}
```

2.3 Die Erklärung

Übersetzt wird das Programm mit dem Kommando `mpicc Hello.c -o Hello`, wenn `Hello.c` der Name der Quelltextdatei ist und das ausführbare Programm `Hello` heißen soll. Beim Übersetzen erscheint in etwa eine solche Ausgabe:

```
spp2000 66:/usr/ccs/bin/ld: (Warning) At least one PA 2.0 object file (4.o)
was detected.
The linked output may not run on a PA 1.x system.
```

Auf einigen Systemen erscheint auch gar keine Meldung, sofern das Programm korrekt ist. Startet man das Programm mit dem Kommando `mpirun -np 5 Hello` (auf einigen Systemen auch nach dem Muster `Hello -np 5`), also mit fünf Prozessen, ergibt sich zum Beispiel diese Ausgabe:

```
Hello World. I'm process 2 of 5 processes.
Hello World. I'm process 0 of 5 processes.
Hello World. I'm process 4 of 5 processes.
Hello World. I'm process 3 of 5 processes.
Hello World. I'm process 1 of 5 processes.
```

Startet man es nochmal, erscheint vielleicht dies auf dem Schirm:

```
Hello World. I'm process 2 of 5 processes.
Hello World. I'm process 3 of 5 processes.
Hello World. I'm process 4 of 5 processes.
Hello World. I'm process 1 of 5 processes.
Hello World. I'm process 0 of 5 processes.
```

Die Prozesse arbeiten zwar gleichzeitig, aber man sieht, daß die Prozesse nicht in der Reihenfolge ihrer Numerierung arbeiten. Man kann auch nicht davon ausgehen, daß die Reihenfolge der Prozesse immer gleich ist. Auch kann es passieren, daß Prozesse andere Prozesse während der Programmausführung überholen. Für den Benutzer ist also sowohl die Reihenfolge der Prozesse als auch deren Arbeitsgeschwindigkeit nicht vorherbestimmbar. Dies muß man beim Schreiben von MPI-Programmen beachten. Auch die Nummern der Prozesse sind willkürlich von MPI (bei 0 beginnend) zugeordnet, denn MPI bildet die vorhandene Hardware-Topologie zunächst auf ein eindimensionales Netz (also eine "Linie") ab.

In diesem Beispiel werden bereits 4 MPI-Funktionen benutzt. Die markanten Zeilen sind kommentiert. In den Zeilen A und B werden die nötigen Header-Dateien eingebunden. Mit `stdio.h` werden (auch im Rest dieses Tutorials) die Funktionen für Eingaben und Ausgaben deklariert. `mpi.h` ist die Header-Datei, die die Funktionen von MPI deklariert. Die Header-Dateien sollten sich im `include`-Verzeichnis der Zielmaschine befinden. Ansonsten müssen die Pfade entweder in der umgebenden Shell gesetzt oder in `mpicc` (bzw. im entsprechenden Skript) angepaßt werden.

In Zeile C wird MPI mit der Funktion `MPI_Init` initialisiert. Dabei werden die Argumente der `main`-Prozedur übergeben. Was genau MPI mit diesen Parametern macht, ist für die Benutzung von MPI unwichtig. Alle Prozesse müssen diese Funktion aufrufen. Jeder andere MPI-Funktion kann erst nach `MPI_Init` aufgerufen werden. Eine Ausnahme bildet die Funktion `MPI_Initialized`, die man benutzen kann, um festzustellen, ob MPI schon initialisiert wurde.

In Zeile D wird mit `MPI_Comm_rank` die Nummer (rank) des Prozesses im Kommunikator `MPI_COMM_WORLD` abgefragt. Ein Kommunikator ist in MPI ein Bereich von Prozessen, der bei einer Kommunikationsfunktion als Einheit angesprochen werden kann. `MPI_COMM_WORLD` ist der Kommunikator, der alle Prozesse beinhaltet.

In Zeile E wird mit `MPI_Comm_size` die Anzahl der Prozesse im Kommunikator `MPI_COMM_WORLD` ermittelt. Dies ist also die Gesamtanzahl aller Prozesse. In Zeile F wird schließlich mit `MPI_Finalize` MPI beendet. Diese Funktion sollte am Ende eines MPI-Programms aufgerufen werden, und zwar von allen Prozessen. Nach diesem Funktionsaufruf befindet sich MPI in demselben Status wie vor dem Aufruf von `MPI_Init`. Danach kann also keine MPI-Funktion erfolgreich aufgerufen werden bis auf `MPI_Init`, `MPI_Initialized` oder

`MPI_Finalized`. Letztere Funktion ist dazu da, um abzufragen, ob MPI schon beendet wurde.

In diesem und in allen folgenden Beispielen wird übrigens darauf verzichtet, die Rückgabewerte der MPI-Funktionen auszuwerten.

2.4 Ein Deadlock

Beim Programmieren mit MPI muß man aufpassen, daß man keinen Deadlock (ein Anhalten des Programms ohne Beenden) fabriziert. Das folgende Beispiel zeigt eine Deadlock-Situation:

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

main (int argc, char* argv[]) {

 int myrank;
 int size;
 int from;
 int to;
 MPI_Status status;
 char message[8];

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 sprintf(message, "MESSAGE");

 from=(size+myrank-1) % size;
 MPI_Recv (message,8,MPI_CHAR,from,0,MPI_COMM_WORLD,&status);
 printf("Process %d received message %s from process %d.\n", myrank, message,
from);

 to=(myrank+1) % size;
 printf("Process %d is sending message %s to process
%d.\n",myrank,message,to);
 MPI_Send (message,strlen(message)+1,MPI_CHAR,to,0,MPI_COMM_WORLD);

 MPI_Finalize();
 exit(0);

}
```

In diesem Beispiel wartet jeder Prozeß mit `MPI_Recv` darauf, daß ein anderer Prozeß ihm etwas mit `MPI_Send` schickt. Da aber `MPI_Recv` solange blockiert, bis tatsächlich etwas empfangen wurde, bleiben alle Prozesse an eben dieser Stelle stehen.

3. Versenden und Empfangen

Die elementaren Operationen in MPI sind das Senden an einen Prozeß und das Empfangen von einem Prozeß. MPI stellt vier Varianten zur Verfügung, um Daten zu versenden, nämlich blockierend, gepuffert, synchronisiert und im "ready mode". Zu jeder Sende-Operation gehört eine Empfangsoperation. Daneben bietet MPI noch für jede Variante die Möglichkeit an, die Daten sofort oder später zu versenden und dann per "Handles" zu verwalten.

3.1 Versenden von Daten

Das Versenden von Daten geschieht mit den Funktionen `MPI_Send`, `MPI_Bsend`, `MPI_Rsend` und `MPI_Ssend`. Diese Funktionen sind blockierend, d.h. sie geben die Kontrolle an das Programm erst wieder zurück, wenn das Betriebssystem die Sende-Operation ausführt, es sei denn, das Betriebssystem (oder die Hardware) stellt einen Zwischenpuffer zur Verfügung. In diesem Fall wirken diese Funktionen nicht synchronisierend.

Die erste Funktion `MPI_Send` ist für das "normale" Versenden da.

Die zweite Funktion `MPI_Bsend` dient zum Versenden eines Puffers. Dazu mehr in Kapitel 3.5.

Die Funktion `MPI_Rsend` arbeitet im "ready mode", also genau wie `MPI_Send`, nur daß hier angenommen werden kann, daß der Empfänger das Empfangen schon mit `MPI_Irecv` eingeleitet hat und nun auf die Daten wartet, daß er also schon bereit (ready) ist.

Die vierte Funktion `MPI_Ssend` dient zum sychronisierten Versenden von Daten. Sie gibt die Kontrolle erst wieder an das Programm zurück, wenn die Daten wirklich empfangen wurden. Hier werden also die beiden beteiligten Prozesse synchronisiert. Wenn die Daten nicht empfangen werden, tritt ein Deadlock auf. Programme, in denen jedes `MPI_Send` durch ein `MPI_Ssend` ausgetauscht werden kann, ohne daß ein Deadlock auftritt, werden auch als "sicher" bezeichnet.

3.2 Ein Beispiel zum Versenden

Das folgende Beispiel demonstriert des Versenden einer Nachricht durch alle Prozesse hindurch:

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

main (int argc, char* argv[]) {

 int myrank; // beinhaltet die Nummer des Prozesses, auf dem diese Instanz des
Programms läuft
 int size;   // beinhaltet die Anzahl der Prozesse insgesamt
 int from;                                  // wird den Vorgänger eines
Prozesses beinhalten
 int to;                                    // wird den Nachfolger eines
Prozesses beinhalten
 MPI_Status status;                         // eine Status-Variable, die hier
ignoriert wird
 char message[8];                           // beinhaltet die Nachricht

 MPI_Init(&argc, &argv);                    // MPI initialisieren
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);    // Nummer des Prozesses holen, auf
dem diese Instanz des Programms läuft
 MPI_Comm_size(MPI_COMM_WORLD, &size);      // Anzahl der Prozesse merken

 sprintf(message, "MESSAGE");

 if (myrank!=0) {                           // wenn diese Instanz nicht auf dem
ersten Proozeß läuft
  from=(size+myrank-1) % size;              // Vorgänger bestimmen
  MPI_Recv (message,8,MPI_CHAR,from,0,MPI_COMM_WORLD,&status);
                                            // vom Vorgänger empfangen
  // Eigentlich müßte es heißen : Darauf warten, daß der Vorgänger eine
Nachricht sendet

  printf("Process %d received message %s from process %d.\n", myrank, message,
from);
 }

 if (myrank!=(size-1)) {                    // wenn diese Instanz nicht auf dem
letzten Prozeß läuft
  to=(myrank+1) % size;                     // Nachfolger bestimmen
  printf("Process %d is sending message %s to process
%d.\n",myrank,message,to);
  MPI_Send (message,strlen(message)+1,MPI_CHAR,to,0,MPI_COMM_WORLD);
                                            // an den Nachfolger eine Nachricht
schicken
 }
```

```
 MPI_Finalize();                              // MPI beenden
 exit(0);

}
```

Die Ausgabe gestaltet sich von System zu System unterschiedlich. Auf einer X-Class Convex erhält man folgende Ausgabe:

```
Process 0 is sending message MESSAGE to process 1.
Process 1 received message MESSAGE from process 0.
Process 1 is sending message MESSAGE to process 2.
Process 2 received message MESSAGE from process 1.
Process 2 is sending message MESSAGE to process 3.
Process 3 received message MESSAGE from process 2.
Process 3 is sending message MESSAGE to process 4.
Process 4 received message MESSAGE from process 3.
Process 4 is sending message MESSAGE to process 5.
Process 5 received message MESSAGE from process 4.
```

Man sieht, daß jeder Prozeß (bis auf den ersten mit der Nummer 0) von seinem "linken" Nachbar eine Nachricht erh&aum;lt und dann an den "rechten" Nachbar weitersendet (bis auf den letzten Prozeß, hier mit der Nummer 5). Damit wird das Wandern einer Nachricht durch alle Prozesse simuliert. Das funktioniert, da hier die Funktionen `MPI_Send` und `MPI_Recv` synchronisierend sind.

Auf einer Parastation erhät man jedoch folgende Ausgabe:

```
Process 0 is sending message MESSAGE to process 1.
Process 1 received message MESSAGE from process 0.
Process 1 is sending message MESSAGE to process 2.
Process 2 received message MESSAGE from process 1.
Process 2 is sending message MESSAGE to process 3.
Process 3 received message MESSAGE from process 2.
Process 3 is sending message MESSAGE to process 4.
Process 4 received message MESSAGE from process 3.
Process 4 is sending message MESSAGE to process 5.
Process 5 received message MESSAGE from process 4.
Process 5 is sending message MESSAGE to process 6.
Process 6 received message MESSAGE from process 5.
Process 6 is sending message MESSAGE to process 7.
Process 9 received message MESSAGE from process 8.
Process 7 received message MESSAGE from process 6.
Process 7 is sending message MESSAGE to process 8.
Process 8 received message MESSAGE from process 7.
Process 8 is sending message MESSAGE to process 9.

PSIlogger: done
```

Hier tanzt der Prozeß mit der Nummer 9 aus der Reihe (warum?).

3.3 Handles

Das Vorbereiten und Versenden der Daten geschieht bei den obigen vier Funktionen in einem Zug. Diese beiden Schritte kann man aber auch trennen. Dazu gibt es die vier Funktionen `MPI_Send_init`, `MPI_Bsend_init`, `MPI_Rsend_init` und `MPI_Ssend_init`. Diese Funktionen arbeiten wie ihre Pendants ohne "`_init`" im Namen, nur mit dem Unterschied, daß hier die Daten nur vorbereitet, aber nicht gesendet werden. Stattdessen liefern sie ein "Handle" vom Typ `MPI_Request` zurück, welches einen Verweis auf diese Kommunikationsoperation darstellt.

Gestartet wird die Kommunikationsoperation mit der Funktion `MPI_Start` oder `MPI_Startall`. Danach befindet sich MPI in einem Zustand, als ob man die zugrundeliegende Operation `MPI_Send`, `MPI_Bsend`, `MPI_Rsend` und `MPI_Ssend` benutzt hätte.

Mit der Funktion `MPI_Cancel` kann ein Handle **vor** dem Aufruf von `MPI_Start` oder `MPI_Startall` gelöscht werden. Danach ist das nicht mehr möglich, da diese beiden Operationen blockierend sind, also erst ihre Funktion ausführen und dann die Kontrolle an das Programm zurückgeben. Der große Vorteil von Handles ist der, daß sie mehrfach benutzt werden können.

Gelöscht wird ein Handle mit `MPI_Request_free`, egal, ob es schon benutzt wurde oder nicht.

3.4 Beispiel für Handles

Dieses Beispiel demonstriert mittels Handles einen Trip eines Datenpakets durch alle Prozesse:

```
/*
 Kommunikation mittels Handles in MPI
*/

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

int main(int argc, char *argv[]) {

 int num;
 int rank, size, next, from;                    // das Übliche
 MPI_Request send_request, recv_request;        // zwei Variable, die die
Handles aufnehmen

 MPI_Status status;

 /* MPI initialisieren */
 MPI_Init(&argc, &argv);
 MPI_Comm_rank( MPI_COMM_WORLD, &rank);
 MPI_Comm_size( MPI_COMM_WORLD, &size);

 next = (rank + 1) % size;
 from = MPI_ANY_SOURCE;

 /* nur Prozeß 0 soll eine Eingabe entgegennehmen */
 if (rank == 0) {
  printf("Enter the number of times around the ring: ");
  scanf("%d", &num);

//  printf("Process %d sending %d to %d\n", rank, num, next);
//  MPI_Send(&num, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
 }

 /*
  Hier warten alle Prozesse darauf, daß ihnen Prozeß 0 den Inhalt der Variable
num mitteilt.
  Diese Funktion wird in Kapitel 4 diskutiert.
 */
 MPI_Bcast(&num, 1, MPI_INT, 0, MPI_COMM_WORLD);

 /* Handles aufbauen */
 MPI_Send_init(&num, 1, MPI_INT, next, 0, MPI_COMM_WORLD, &send_request);
 MPI_Recv_init(&num, 1, MPI_INT, from, 0, MPI_COMM_WORLD, &recv_request);

 /*
  Eine Variable wird durch alle Prozesse geschickt, kommt sie beim Prozeß 0
an,
  wird diese Variable um 1 dekrementiert.
 */
 while (num > 0) {
  MPI_Start(&recv_request);
  MPI_Wait(&recv_request, &status);
```

```
  printf("Process %d received %d\n", rank, num);

  if (rank == 0) {
   num--;
   printf("Process 0 decremented num\n");
  }

  printf("Process %d sending %d to %d\n", rank, num, next);

  MPI_Start(&send_request);
  MPI_Wait(&send_request, &status);
 }

 printf("Process %d exiting\n", rank);

 /* der letzte Prozeß muß noch einmal empfangen */
 if (rank == 0) {
  MPI_Recv(&num, 1, MPI_INT, from, 0, MPI_COMM_WORLD, &status);
 }

 /* Handles löschen */
 MPI_Request_free(&send_request);
 MPI_Request_free(&recv_request);

 MPI_Finalize();
 return 0;

}
```

Das Programm liefert, mit drei Prozesse gestartet, folgende Ausgabe:

```
Enter the number of times around the ring: 2
Process 0 sending 2 to 1
Process 1 received 2
Process 1 sending 2 to 2
Process 2 received 2
Process 2 sending 2 to 0
Process 0 received 2
Process 0 decremented num
Process 0 sending 1 to 1
Process 1 received 1
Process 1 sending 1 to 2
Process 2 received 1
Process 2 sending 1 to 0
Process 0 received 1
Process 0 decremented num
Process 0 sending 0 to 1
Process 0 exiting
Process 1 received 0
Process 1 sending 0 to 2
Process 1 exiting
Process 2 received 0
Process 2 sending 0 to 0
Process 2 exiting
```

3.5 Verwenden von Puffern

Mit den bisherigen Funktionen wird grundsätzlich eine Datenstruktur versendet, die in Form einer Variable vorliegt. Es ist aber auch möglich, MPI diese Variable mittels `MPI_Buffer_attach` als Puffer bekannt zu machen. Nachdem MPI einen Puffer kennt, kann eine Sende-Operation `MPI_Bsend` auf diesem Puffer ausgeführt werden. Dabei ist zu beachten, daß dies spätestens dann geschehen sein muß, bevor die zugehörige Empfangsoperation ausgeführt wird. Das Verwenden von Puffern zum Senden verlangsamt Programme etwas,

denn MPI muß die Daten im Puffer nach dem `MPI_Bsend` in einen internen Puffer kopieren, von wo die zugehörige Empfangsoperation sich diese Daten dann abholen kann. Wenn ein Puffer nicht mehr gebraucht wird, kann er mit `MPI_Buffer_detach` zerstört werden.

3.6 Nichtblockierende Kommunikation

Manchmal ist es wünschenswert, dem Betriebssystem eine Kommunikationsoperation "als Aufgabe" zu geben, die dann parallel zum eigentlichen Programm ausgeführt wird. Genau dies wird in MPI mit nichtblockierender Kommunikation implementiert.

Dazu gibt es die Funktionen `MPI_Isend`, `MPI_Ibsend`, `MPI_Irsend` und `MPI_Issend`. Diese Funktionen arbeiten wie ihre blockierenden Gegenstücke ohne "`I`", nur daß sie genau wie die entsprechenden Funktionen mit "`_init`" ein Handle vom Typ `MPI_Request` zurückgeben. Dieses Handle repräsentiert dann im weiteren Verlauf die zugrundeliegende Sende-Operation. Die Operation selbst wird "irgendwann" nach dem Funktionsaufruf gestartet. "Irgendwann" deshalb, weil man als Benutzer den Zeitpunkt nicht bestimmen kann.

Da diese Funktionen nicht blockierend sind, gibt MPI die Kontrolle sofort nach Funktionsaufruf an das Programm zurück. Deshalb bietet MPI Funktionen an, um den Status der Sende-Operation weiterzuverfolgen. Zum einen gibt es die Funktionen `MPI_Test`, `MPI_Testall`, `MPI_Testany` und `MPI_Testsome`, mit denen der Status der Operationen abgefragt werden kann.

Zum anderen gibt es die Funktionen `MPI_Wait`, `MPI_Waitall`, `MPI_Waitany` und `MPI_Waitsome`, die das Programm solange blockieren, bis die entsprechenden Operationen abgeschlossen sind. Damit kann man also eine Synchronisation erzwingen. Der umgekehrte Weg ist selbstverständlich nicht möglich.

3.7 Beispiel zur nichtblockierenden Kommunikation

Folgendes Beispiel demonstriert die Berechnung des Durchschnitts mittels nichtblockierender Kommunikation:

```
/*
 Durchschnittsberechnung mit MPI
 und nichtblockierender Kommunikation
*/

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

int main(int argc, char *argv[]) {

 int source, dest;
 int total, workers;
 double avg;
 int *data, *sum, mysum;
 int rank, size;
 int i;
 int len = 100;

 MPI_Request *request;

 /* MPI initialisieren */

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

 MPI_Status status;

 /* Prozeß 0 spielt den Manager */

 if (rank == 0) {

  /* Daten vorbereiten */
```

```
  workers = size - 1;
  data = malloc(sizeof(int) * workers * len);
  sum = malloc(sizeof(int) * workers);
  request = malloc(sizeof(MPI_Request) * workers);

  for (i = 0; i < workers * len; ++i) {
   data[i] = i;
  }

  /* Daten zu den Arbeitern senden */

  for (i = 0; i < workers; ++i) {
   dest = i + 1;

   /* nichtblockierendes Senden verwenden */

   MPI_Isend(data + i * len, len, MPI_INT, dest, dest, MPI_COMM_WORLD,
&request[i]);
   /* Hier wird an fünfter Stelle ein Tag-Parameter benutzt, der die Operation
kennzeichnet.
      Dasselbe passiert beim Empfangen. Zum Tag-Parameter siehe Kapitel 3.10
*/
  }

  /* warten, bis alle Daten versendet wurden */

  MPI_Waitall(workers, request, MPI_STATUSES_IGNORE);

  /* Ergebnisse erwarten */

  for (i = 0; i < workers; ++i) {
   source = i + 1;

   /* nichtblockierendes Empfangen */

   MPI_Irecv(sum + i, 1, MPI_INT, source, source, MPI_COMM_WORLD,
&request[i]);
  }

  /* warten, bis alle Daten empfangen wurden */

  MPI_Waitall(workers, request, MPI_STATUSES_IGNORE);

  /* Durchschnitt berechnen */

  total = 0;
  for (i = 0; i < workers; ++i) {
   total += sum[i];
  }

  avg = total / (len * workers);
  printf("The average is %lf\n", avg);

  /* und aufrämen */

  free(data);
  free(sum);
  free(request);
 }

 /* Code für die Arbeiter */

 else {

  /* Behälter für Daten vorbereiten */
```

```
  data = malloc(sizeof(int) * len);
  mysum = 0;

  /* Daten vom Manager empfangen */

  MPI_Recv(data, len, MPI_INT, 0, rank, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  /* und eigene Summe berechnen */

  for (i = 0; i < len; ++i) {
   mysum += data[i];
  }

  /* und wieder zurück an den Manager senden */

  MPI_Send(&mysum, 1, MPI_INT, 0, rank, MPI_COMM_WORLD);

  /* und aufrämen */

  free(data);
 }

 MPI_Finalize();
 return 0;

}
```

Die Ausgabe hierzu ist nur eine Zeile:

```
The average is 149.000000
```

3.8 Empfangen von Daten

Zum Empfangen von Daten gibt es die Funktionen `MPI_Recv` zum blockierenden Empfangen, `MPI_Recv_init` zum Empfangen mit einem Handle und `MPI_Irecv` zum nichtblockierenden Empfangen. Es ist zu beachten, daß es für jede Empfangsoperation eine Sende-Operation geben muß. Dabei kann man z.B. blockierendes Senden mit Empfangen per Handle kombiniert werden.

Beim Empfangen von Daten muß angegeben werden, von welchem Prozeß die Nachricht gesendet sein soll. Mit der Konstante `MPI_ANY_SOURCE` für diesen Parameter kann von jedem Prozeß empfangen werden.

Für das Empfangen per Handle können dieselben Start- und Abbruch-Operationen wie für die entsprechenden Sende-Operationen `MPI_????_init` benutzt werden. Für das nichtblockierende Empfangen werden dieselben Abbruch-, Test- und Warte-Funktionen benutzt wie für die nichtblockierenden Sende-Operationen `MPI_I????` bereitgestellt.

3.9 Gleichzeitig Senden und Empfangen

Manchmal ist es nötig, daß Prozesse eine Variable senden und eine andere empfangen wollen. Für solche Vertauschungen gibt es die Funktionen `MPI_Sendrecv` und `MPI_Sendrecv_replace`. Ein `MPI_Sendrecv` ist äquivalent zu einem `MPI_Send`, gefolgt von einem `MPI_Recv`. `MPI_Sendrecv_replace` funktioniert genauso mit dem Unterschied, daß Sende-Puffer und Empfangspuffer gleich sind. Mit `MPI_Sendrecv_replace` kann also eine Variable ausgetauscht werden.

3.10 Der Tag-Parameter

Sende- und Empfangsoperationen haben einen Parameter namens `tag` (also Kennzeichnung). Damit lassen sich Operationen kennzeichnen. Eine Empfangsoperation "empfängt" nur dann etwas von einer zugehörigen Sende-Operation, wenn beim Senden und Empfangen der Tag-Parameter übereinstimmt. Um den Tag-Parameter zu ignorieren und Nachrichten mit jeder Kennzeichnung zu empfangen, kann der Parameter `MPI_ANY_TAG` benutzt werden.

3.11 Das laufende Beispiel ff

Die grundlegenden Funktionen zum Senden und Empfangen werden an einigen Stellen benutzt, zum Beispiel an folgender:

```
.
.
.
/* Compute related parameters. */
my_pe_m1 = my_pe - 1;
my_pe_p1 = my_pe + 1;
last_pe = npes - 1;
my_last = my_length - 1;

/* Store local segment of u in the working array z. */
for (i = 1; i <= my_length; i++)
  z[i] = udata[i - 1];

/* Pass needed data to processes before and after current process. */
if (my_pe != 0)
  MPI_Send(&z[1], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
if (my_pe != last_pe)
  MPI_Send(&z[my_length], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);
.
.
.
```

Hier werden Daten an den Vorgänger und Nachfolger (sofern sie existieren) in der Nummerierung der Prozesse geschickt. Die Reihung `udata[]` enthät Daten eines Vektors. Die zugehörige Empfangsoperation sieht folgendermaßen aus:

```
.
.
.
/* Receive needed data from processes before and after current process. */
if (my_pe != 0)
  MPI_Recv(&z[0], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
else z[0] = 0.0;
if (my_pe != last_pe)
  MPI_Recv(&z[my_length+1], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm,
&status);
else z[my_length + 1] = 0.0;
.
.
.
```

Hier werden die Daten vom Vorgänger und vom Nachfolger geholt (falls sie existieren). Zu beachten ist, daß der Tag-Parameter bei `MPI_Send` und beim zugehörigen `MPI_Recv` identisch (hier 0) sind.

An anderen Stellen wird nichtblockierende Kommunikation benutzt. Hier zunächst die blockierenden Sende-Operationen:

```
.
.
.
if (isuby != 0)
  MPI_Send(&uarray[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);

/* If isuby < NPEY-1, send data from top x-line of u */

if (isuby != NPEY-1) {
  offsetu = (MYSUB-1)*dsizex;
  MPI_Send(&uarray[offsetu], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
```

```
}

/* If isubx > 0, send data from left y-line of u (via bufleft) */

if (isubx != 0) {
  for (ly = 0; ly < MYSUB; ly++) {
    offsetbuf = ly*NVARS;
    offsetu = ly*dsizex;
    for (i = 0; i < NVARS; i++)
      bufleft[offsetbuf+i] = uarray[offsetu+i];
  }
  MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
}

/* If isubx < NPEX-1, send data from right y-line of u (via bufright) */

if (isubx != NPEX-1) {
  for (ly = 0; ly < MYSUB; ly++) {
    offsetbuf = ly*NVARS;
    offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVARS;
    for (i = 0; i < NVARS; i++)
      bufright[offsetbuf+i] = uarray[offsetu+i];
  }
  MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
}
.
.
.
```

Dies sind die zugehörigen nichtblockierenden Empfangsoperationen:

```
.
.
.
/* If isuby > 0, receive data for bottom x-line of uext */
if (isuby != 0)
  MPI_Irecv(&uext[NVARS], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm,
&request[0]);

/* If isuby < NPEY-1, receive data for top x-line of uext */
if (isuby != NPEY-1) {
  offsetue = NVARS*(1 + (MYSUB+1)*(MXSUB+2));
  MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm,
&request[1]);
}

/* If isubx > 0, receive data for left y-line of uext (via bufleft) */
if (isubx != 0) {
  MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm,
&request[2]);
}

/* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
if (isubx != NPEX-1) {
  MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm,
&request[3]);
}
.
.
.
```

4. Verteilen und Sammeln

Ein häufig vorkommendes Szenario beim parallelen Programmieren ist das Verteilen einer Datenstruktur (z.B. einer Reihung) auf andere Prozesse. Anstatt nun diese Reihung im "Heimat-Prozeß" aufzuteilen und die Einzelstücke in einer Schleife zu übertragen, kann man diese Aufgabe MPI übertragen. Dies hat zwei Vorteile :

- Überläßt man es MPI, eine Datenstruktur aufzuteilen, versucht MPI, diese Datenstruktur möglichst unter allen beteiligten Prozessen gleich aufzuteilen. Manuell muß man dafür selbst sorgen.
- Das Aufteilen der Datenstruktur und die oben erwähnte Schleife zum Versenden haben eine gewisse Laufzeit, die entsprechenden MPI-Funktionen sind aber viel schneller.

Noch wichtiger als die Funktionen zum Verteilen sind die Funktionen zum Sammeln von Daten. Denn dabei können die Daten von MPI nicht nur zusammengefügt werden, sondern auch gleich verrechnet (z.B. aufsummiert) werden. Auch hierbei verringert sich wieder die Laufzeit. Als Dreingabe stellt MPI auch noch Funktionen bereit, die Daten von Prozessen sammeln, dann verrechnen oder zusammenfügen und dann das Ergebnis allen anderen Prozessen bekanntmachen.

4.1 Das Prinzip

Die Funktionen zum Sammeln und Verteilen folgen alle (mit Ausnahme von `MPI_Bcast`) demselben Prinzip. Diese Funktionen haben alle sozusagen einen "Topf" (einen Speicherbereich ohne Typ), in den jeder Prozeß seinen Anteil hineinpackt, sofern die Funktion von diesem Prozeß Daten "einsammelt", der Prozeß bei dieser Funktion also am Senden beteiligt ist. Sobald alle Prozesse ihren Anteil beigetragen haben, verteilt die Funktion diese Daten an alle Prozesse, die bei dieser Funktion am Empfangen beteiligt sind. Die Speicherbereiche für das Empfangen und Senden dürfen sich dabei nicht überlappen. Zu bemerken ist, daß alle Funktionen zum Sammeln und Verteilen (auch `MPI_Bcast`) blockierend auf die beteiligten Prozesse wirken.

Daneben gibt es die Funktion `MPI_Barrier`, die Prozesse ohne den Austausch von Daten synchronisiert.

4.2 Verteilen von Datenstrukturen

MPI stellt zum Verteilen von Daten drei Funktionen zur Verfügung. Es sind dies `MPI_Bcast`, `MPI_Scatter` und `MPI_Scatterv`. Während `MPI_Bcast` einen Datenbereich allen Prozessen bekannt macht, verteilen die beiden letzten Funktionen Datenbereiche zu gleichen Anteilen auf alle Prozesse. Dabei kann bei `MPI_Scatterv` der Datenbereich **vor** dem Senden in Einzelstücke (durch Angabe von Anfangspunkt und Länge) unterteilt werden. Diese Einzelstücke werden dann von der Funktion aufgeteilt.

4.3 Verteilen einer Matrix

Das folgende Beispiel zeigt, wie mit `MPI_Scatter` eine Matrix an andere Prozesse verteilt werden kann:

```
/*
 Verteilen von Datenstrukturen mit MPI
*/

#include <mpi.h>
#include <stdio.h>
#define SIZE 4

int main(int argc,char *argv[]) {

 int numtasks, rank, sendcount, recvcount, source;

 float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}
 };

 float recvbuf[SIZE];

 /* MPI initialisieren */

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

```
 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

 if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;

  // Matrix verteilen, jeder Prozeß bekommt eine Spalte
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank=%d,  result : %f %f %f %f\n",rank,recvbuf[0],
recvbuf[1],recvbuf[2],recvbuf[3]);
 }
 else {
  printf("number of processes must be %d, exiting ...\n",SIZE);
 }

 MPI_Finalize();

}
```

Aus Ausgabe präsentiert die aufgespaltene Matrix:

```
rank=1,  result : 5.000000 6.000000 7.000000 8.000000
rank=2,  result : 9.000000 10.000000 11.000000 12.000000
rank=3,  result : 13.000000 14.000000 15.000000 16.000000
rank=0,  result : 1.000000 2.000000 3.000000 4.000000
```

4.4 Sammeln von Daten

Zum reinen Sammeln von Daten stellt MPI vier Funktionen zu Verfügung, nämlich `MPI_Gather`, `MPI_Gatherv`, `MPI_Allgather` und `MPI_Allgatherv`.

Die Funktionen `MPI_Gather` und `MPI_Gatherv` sind die Umkehrfunktionen von `MPI_Scatter` und `MPI_Scatterv`. Sie sammeln Daten von allen Prozessen ein, ein einzelner Prozeß, der "Besitzer" der Funktion erhält die Daten.

Beim Aufruf der Funktionen `MPI_Allgather` und `MPI_Allgatherv` werden ebenfalls Daten eingesammelt, jedoch werden allen Prozessen die Daten bekannt gemacht.

4.5 Verrechnen von Daten beim Sammeln

Mitunter möchte man Daten einsammeln und sie gleich verrechnen. Dafür gibt es die drei Funktionen `MPI_Reduce`, `MPI_Allreduce` und `MPI_Scan`. Diese Funktionen arbeiten wie `MPI_Gather(v)`, nur daß man noch mitteilen muß wie die Daten zu verrechnen sind.

`MPI_Scan` berechnet nur Teilergebnisse. Das Ergebnis der Berechnung beim Aufruf der Funktion im Prozeß *n* ist das Ergebnis der Berechnung mit den Werten von Prozeß 0 bis Prozeß *n*.

Bei den Operationen `MPI_MAXLOC` und `MPI_MINLOC` gibt es eine Besonderheit. Das Ergebnis ist hier eine Strukturvariable (in Fortran eine Reihung), dessen erste Komponente des Maximum (Minimum) der übergebenen Werte angibt und dessen zweite Komponente die Nummer des Prozesses angibt, der diesen Wert beigesteuert hat.

4.6 Parallele Summenberechnung

Im folgenden Beispiel werden Prozesse mit `MPI_Reduce` Summen von Teilstücken einer Reihung berechnen:

```
/*
 Summenberechnung eines Arrays mit MPI
*/
```

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

#define ARRAYSIZE 16000000
#define MASTER 0

/* Behälter für Daten */

float data[ARRAYSIZE];

int main(int argc,char *argv[]) {

 int numtasks, taskid, rc, dest, source, offset;
 int i, j, tag1, tag2, chunksize;

 float mysum, sum;
 float update(int myoffset, int chunk, int myid);

 MPI_Status status;

 /* MPI initialisieren */

 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
 MPI_Comm_rank(MPI_COMM_WORLD,&taskid);

 /* Anzahl der Prozesse muß durch 4 teilbar sein */
 if (numtasks % 4 != 0) {
  printf("Exiting, number of tasks must be divisible by 4, it is currently
%d.\n", numtasks);
  MPI_Abort(MPI_COMM_WORLD, rc);
  exit(0);
 }

 printf ("MPI task %d has started...\n", taskid);

 chunksize = (ARRAYSIZE / numtasks);
 tag2 = 1;
 tag1 = 2;

 /* Code des Managers */
 if (taskid == MASTER) {

 /* Daten vorbereiten */
 sum = 0;
 for(i=0; i < ARRAYSIZE; i++) {
  data[i] =  i * 1.0;
  sum = sum + data[i];
 }

 /* Daten auf alle Prozesse verteilen */
 offset = chunksize;
 for (dest=1; dest < numtasks; dest++) {
  MPI_Send(&offset, 1, MPI_INT, dest, tag1, MPI_COMM_WORLD);
  MPI_Send(&data[offset], chunksize, MPI_FLOAT, dest, tag2, MPI_COMM_WORLD);
  offset = offset + chunksize;
 }

 /* der Manager muß seinen Teil beitragen */
 offset = 0;
 mysum = update(offset, chunksize, taskid);

 /* warten auf die Ergebnisse der Arbeiter */
 for (i=1; i < numtasks; i++) {
  source = i;
  MPI_Recv(&offset, 1, MPI_INT, source, tag1, MPI_COMM_WORLD, &status);
```

```
  MPI_Recv(&data[offset], chunksize, MPI_FLOAT, source, tag2,
  MPI_COMM_WORLD, &status);
 }

 /* die einzelnen Summen der Arbeiter sammeln und aufaddieren */
 MPI_Reduce(&mysum, &sum, 1, MPI_FLOAT, MPI_SUM, MASTER, MPI_COMM_WORLD);

 printf("\nresults: \n");
 offset = 0;
 for (i=0; i < numtasks; i++) {
  for (j=0; j<5; j++) {
   printf("  %e",data[offset+j]);
  }
  printf("\n");
  offset = offset + chunksize;
 }
 printf("sum= %e\n",sum);

 }

 /* Code für Arbeiter */

 if (taskid > MASTER) {

  /* Daten vom Manager empfangen */
  source = MASTER;
  MPI_Recv(&offset, 1, MPI_INT, source, tag1, MPI_COMM_WORLD, &status);
  MPI_Recv(&data[offset], chunksize, MPI_FLOAT, source, tag2, MPI_COMM_WORLD,
&status);

  /* aufaddieren

  mysum = update(offset, chunksize, taskid);

  /* und Daten zurückschicken */
  dest = MASTER;
  MPI_Send(&offset, 1, MPI_INT, dest, tag1, MPI_COMM_WORLD);
  MPI_Send(&data[offset], chunksize, MPI_FLOAT, MASTER, tag2, MPI_COMM_WORLD);

  /* an der Summenbildung teilnehmen */

  MPI_Reduce(&mysum, &sum, 1, MPI_FLOAT, MPI_SUM, MASTER, MPI_COMM_WORLD);

 }

 MPI_Finalize();

}

/* Summe der Daten eines einzelnen Prozesses berechnen */
float update(int myoffset, int chunk, int myid) {

 int i;
 float mysum;

 mysum = 0;
 for(i=myoffset; i < myoffset + chunk; i++) {
  data[i] = data[i] + i * 1.0;
  mysum = mysum + data[i];
 }

 return(mysum);
}
```

Die Ausgabe präsentiert die Ergebnisse:

```
MPI task 2 has started...
MPI task 3 has started...
MPI task 0 has started...
MPI task 1 has started...
results:
  0.000000e+00  2.000000e+00  4.000000e+00  6.000000e+00  8.000000e+00
  4.000000e+06  4.000001e+06  4.000002e+06  4.000003e+06  4.000004e+06
  8.000000e+06  8.000001e+06  8.000002e+06  8.000003e+06  8.000004e+06
  1.200000e+07  1.200000e+07  1.200000e+07  1.200000e+07  1.200000e+07
sum= 1.598859e+13
```

4.7 Benutzerdefinierte Operationen

Die Funktionen `MPI_Reduce`, `MPI_Allreduce` und `MPI_Scan` nehmen eine Operation entgegen, die angibt, wie die Daten verrechnet werden sollen. Die von MPI vordefinierten Operationen sind jedoch nicht mehr adäquat, wenn die Operationen, die vorgenommen werden sollen, komplexer werden sollen. Deshalb bietet MPI die Möglichkeit an, benutzerdefinierte Operationen einzuführen.

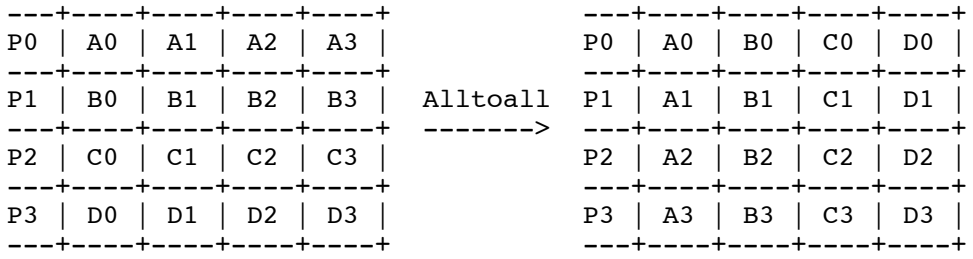
Mit der Funktion `MPI_Op_create` können aus selbstgeschriebenen Funktionen des Typs `MPI_User_function` neue Operationen erstellt werden. Die neuen Funktionen sind wieder vom Typ `MPI_Op`, können also anstelle von den vorgegebenen Operationen benutzt werden.

Die Funktion `MPI_Op_free` "zerstört" eine benutzerdefinierte Funktion wieder.

4.8 Gleichzeitiges Sammeln und Verteilen

Die Funktion `MPI_Reduce_scatter` kombiniert die Möglichkeiten der Funktionen `MPI_Reduce` und `MPI_Scatterv`. Diese Funktion führt ein `MPI_Reduce`, gefolgt von einem `MPI_Scatterv` aus.

Daneben gibt es die Funktionen `MPI_Alltoall` und `MPI_Alltoallv`, die bestimmte Elemente einer Datenstruktur (nicht notwendigerweise die gleichen!) eines jeden Prozesses allen Prozessen mitteilt. Dazu ein Bild (Px ist der Prozeß mit der Nummer x):



4.9 Das laufende Beispiel ff

An z.B. folgender Stelle wird die Funktion `MPI_Allreduce` benutzt, um die Summe von Elementen von Vektoren zu berechnen und um diese Summe allen Prozessen bekannt zu machen:

```
/* Compute global length as sum of local lengths */
n = local_vec_length;
MPI_Allreduce(&n, &Nsum, 1, PVEC_INTEGER_MPI_TYPE, MPI_SUM, comm);
```

Hier steuert jeder Prozeß im Kommunikator `comm` **einen** Wert (wegen der 1 als drittes Argument) von der Stelle, auf die `n` zeigt, zur Summenberechnung bei. Die Summe wird in jedem Prozeß an der Stelle abgelegt, auf die `Nsum` zeigt.

5. Topologien mit MPI

In den meisten Anwendungen, die mit MPI arbeiten, stellen die Prozesse ein Abbild eines Ausschnitts der Realität dar. Bei der Berechnung eines Integrals wird man die Prozesse längs der reellen Achse anordnen, wobei diese Anordnung eine logische Anordnung ist, also keinesfalls in der zugrundeliegenden Hardware so existieren muß. Bei der "Berechnung" von Wettermodellen werden die Prozesse logisch auf einen Quader aufgeteilt, der den beobachteten Luftraum darstellt. MPI unterstützt solche virtuellen Topologien und bietet zwei Arten von Topologien an: zum einen kartesische Gitter und zum anderen Graphen. In einem kartesischen Gitter werden die Prozesse in einem *n*-dimensionalem Quader angeordnet. Ein Prozeß ist dann mit seinen jeweiligen Nachbarn in jeder Dimension direkt verbunden. Bei einem Graphen werden die Prozesse auf die Knoten des Graphen abgebildet. Die Kanten stellen virtuelle Verbindungen zwischen den Knoten dar, über sie läuft die Kommunikation ab.

5.1 Erzeugen von Topologien

Bevor eine virtuelle Topologie benutzt werden kann, muß sie erzeugt werden. Für kartesische Gitter gibt es die Hilfsfunktion `MPI_Dims_create`. Diese Funktion versucht, eine gewisse Anzahl (hier Prozesse) bestmöglichst auf eine gegebene Anzahl von Dimensionen aufzuteilen.

Ein kartesisches Gitter wird mit der Funktion `MPI_Cart_create` erzeugt, ein Graph mit der Funktion `MPI_Graph_create`. Beide Funktionen ordnen die beteiligten Prozesse einem neuen Kommunikator zu, wobei die beteiligten Prozesse nicht explizit angegeben werden können. Vielmehr kann nur ein "Quell-Kommunikator" angegeben werden, aus dem dann der neue Kommunikator aufgebaut wird. Die Topologie wird ab ihrer Erzeugung unter dem neuen Kommunikator vom Typ `MPI_Comm` angesprochen.

5.2 Arbeiten mit Topologien

Nachdem eine Topologie erzeugt wurde, können Informationen über sie abgefragt werden. Dies geschieht mit den Funktionen `MPI_Cart_get`, `MPI_Cartdim_get`, `MPI_Graph_get` und `MPI_Graphdims_get`. Die Funktionen mit "`dim`" im Namen liefern die Größe der Topologie zurück, also die Dimension des Gitters bzw. die Anzahl der Knoten und Kanten des Graphs. Die beiden anderen Funktionen liefern dann weitere Werte.

Man kann einen Prozeß auf eine bestehende Topologie abbilden. Dazu gibt es die Funktionen `MPI_Cart_map` und `MPI_Graph_map`. Diese Funktionen ordnen den aufrufenden Prozeß jedoch nicht in die Topologien ein. Die Funktion `MPI_Cart_sub` erstellt aus einem kartesischem Gitter ein anderes kartesisches Gitter mit weniger Dimensionen.

Im allgemeinen wird nun die Position eines Prozesses innerhalb einer Topologie von Interesse sein. Da Topologien aber spezielle Kommunikatoren sind, kann man die Funktionen für Kommunikatoren dafür verwenden. Mehr dazu im Kapitel 7, Kommunikatoren und Gruppen. In einem kartesischen Gitter kann ein Prozeß aber nicht nur durch seine Nummer (rank) identifiziert werden, sondern auch durch seine Koordinaten. Die Funktion `MPI_Cart_coords` liefert die Koordinaten eines Prozesses zurück, dessen Nummer (rank) bekannt ist. Die Funktion `MPI_Cart_rank` gibt hingegen die Nummer (rank) eines Prozesses zurück, dessen Koordinaten bekannt sind. Die Funktion `MPI_Cart_shift` ermittelt schließlich die Nummern (ranks) der Nachbarn eines Prozesses.

In Graphen gibt es keine solche topologische Anordnung. Vielmehr hat dort ein Prozeß andere Prozesse als Nachbarn, die durch Kanten verbunden sind. Die Anzahl der Knoten, die mit einem bestimmten Knoten adjazent sind, und die Anzahl der Kanten, die mit diesem Knoten inzident sind, können mit der Funktion `MPI_Graph_neighbors_count` in Erfahrung gebracht werden. Mit der Funktion `MPI_Graph_neighbors` können die Nummern (ranks) der Nachbarn eines Knotens ermittelt werden.

5.3 Ein kartesisches Gitter

Das folgende Beispiel erzeugt ein zweidimensionales Gitter:

```
/*
 kartesisches Gitter in MPI
*/
```

```
#include <mpi.h>
#include <stdio.h>
#define SIZE 16
#define UP    0
#define DOWN  1
#define LEFT  2
#define RIGHT 3

int main(int argc, char *argv[]) {

 int numtasks, rank, source, dest, outbuf, i, tag=1;
 int nbrs[4], dims[2]={4,4};
 int periods[2]={0,0}, reorder=0, coords[2];

 MPI_Comm cartcomm;

 /* MPI initialisieren */

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

 if (numtasks == SIZE) {

  /* kartesisches Gitter erzeugen */
  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);

  /* Nummern (ranks) der Nachbarn ermitteln */
  MPI_Comm_rank(cartcomm, &rank);
  MPI_Cart_coords(cartcomm, rank, 2, coords);
  MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
  MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);

  printf("rank= %d coords= %d %d  neighbours(up,down,left,right)= %d %d %d
%d\n", rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT], nbrs[RIGHT]);

 }
 else {
  printf("There must be %d processes, exiting ...\n",SIZE);
 }

 MPI_Finalize();

}
```

Die Ausgabe listet die einzelnen Prozesse, deren Koordinaten und Nachbarn auf:

```
rank= 1 coords= 0 1  neighbors(u,d,l,r)= -1 5 0 2
rank= 0 coords= 0 0  neighbors(u,d,l,r)= -1 4 -1 1
rank= 8 coords= 2 0  neighbors(u,d,l,r)= 4 12 -1 9
rank= 10 coords= 2 2  neighbors(u,d,l,r)= 6 14 9 11
rank= 9 coords= 2 1  neighbors(u,d,l,r)= 5 13 8 10
rank= 6 coords= 1 2  neighbors(u,d,l,r)= 2 10 5 7
rank= 2 coords= 0 2  neighbors(u,d,l,r)= -1 6 1 3
rank= 4 coords= 1 0  neighbors(u,d,l,r)= 0 8 -1 5
rank= 3 coords= 0 3  neighbors(u,d,l,r)= -1 7 2 -1
rank= 11 coords= 2 3  neighbors(u,d,l,r)= 7 15 10 -1
rank= 7 coords= 1 3  neighbors(u,d,l,r)= 3 11 6 -1
rank= 12 coords= 3 0  neighbors(u,d,l,r)= 8 -1 -1 13
rank= 5 coords= 1 1  neighbors(u,d,l,r)= 1 9 4 6
rank= 14 coords= 3 2  neighbors(u,d,l,r)= 10 -1 13 15
rank= 13 coords= 3 1  neighbors(u,d,l,r)= 9 -1 12 14
rank= 15 coords= 3 3  neighbors(u,d,l,r)= 11 -1 14 -1
```

Eine Nummer -1 als Nachbarn bedeutet, daß es in dieser Richtung keinen Nachbarn gibt. Zum Beispiel hat Prozeß 1 keinen Nachbarn in der Richtung "oben".

6. Typen in MPI

Bisher beschränkten sich die versendeten Daten auf primitive Datentyen, also Typen wie `int`, `double`, `char`. In der Realität wird man in C (oder in Fortran 77) die Vorteile von zusammengesetzten Typen nutzen. Um nun einen solchen Typ mit MPI zu verschicken, müßte man nun jedes einzelne Mitglied dieses Typs verschicken. Hat man beispielsweise einen Typ `Point3D`, der aus drei Mitgliedern vom Typ `double` besteht, müßte man nun drei `double`-Werte verschicken. Dies ist nicht nur umständlich, sondern auch fehleranfällig. Letztlich besteht (in C) auch keine Möglichkeit, generisch einen Typ zur Laufzeit in seine Bestandteile aufzulösen. Aus diesen Gründen bietet MPI die Unterstützung von benutzerdefinierten Datentypen an.

6.1 Erzeugen von neuen Typen

MPI kann weder Typ-Aliase (`typedef`) noch Zeiger (`*`) verschicken. Auch kann MPI nicht die zusammengesetzten Typen verarbeiten, die mit `[]` oder `struct` erstellt werden. Vielmehr muß man die Reihungen oder Strukturen mit MPI noch einmal erstellen, um eine Referenz auf eine neue Variable vom Typ `MPI_Datatype` zu bekommen. Diese Variable kann dann in MPI_Funktionen überall dort eingesetzt werden, wo eine Variable vom Typ `MPI_Datatype` gefordert ist.

MPI kennt vier Arten zusammengesetzter Typen: Vektoren, Strukturen, indizierte Listen und zusammenhängende Typen. Strukturen haben dieselbe Semantik wie in C. Vektoren bzw. indizierte Listen dagegen sind Speicherbereiche, die aus Teilen gleicher bzw. unterschiedlicher Größe bestehen. Der Anfang eines jeden solchen Speicherfragments hat einen Typ, zwischen zwei Elementen eines Vektors oder einer Liste kann jedoch Platz einer fest definierten Größe gelassen werden. Ein zusammenhängender Typ ist einfach eine Hintereinanderreihung von gleichen Typen ohne Lücken.

Eine Struktur wird mit der Funktion `MPI_Type_struct` erstellt. Als letztes Element muß eine Struktur eine Variable vom Typ `MPI_UB` enthalten.

Für Vektoren und indizierte Listen gibt es jeweils zwei Funktionen: `MPI_Type_vector`, `MPI_Type_hvector`, `MPI_Type_indexed` und `MPI_Type_hindexed`. Die Varianten mit "`h`" und ohne "`h`" im Namen arbeiten bis auf einen Unterschied gleich: Die mit einem "`h`" im Namen erwarten die Angabe der Lücken als Typ `MPI_Aint`, die anderen beiden als Typ `int`. Die Hilfsfunktion `MPI_Address` liefert zu einer Speicherposition (also zum Wert eines Zeigers) den zugehörigen Wert im Typ `MPI_Aint` zurück.

Zusammenhängende Typen werden mit der Funktion `MPI_Type_contiguous` erzeugt. Ein zusammengesetzter Typ kann selbst wieder aus zusammengesetzten Typen bestehen, es sind jedoch keine Rekusivitäten möglich.

Um MPI den neu erzeugten Typ bekanntzumachen, ist ein Aufruf der Funktion `MPI_Type_commit` nötig. Die Funktion `MPI_Type_free` zerstört einen Typ wieder. Er kann dann nicht mehr benutzt werden. Es ist zu beachten, daß die Lebensdauer eines MPI-Typs hier vom Programmierer vorgegeben wird und nicht vom Compiler.

6.2 Arbeiten mit Typen

Wie bereits erwähnt, kann ein einmal erzeugter Typ immer dann eingesetzt werden, wenn ein Argument vom Typ `MPI_Datatype` gefordert wird. Um Informationen über einen Datentyp zu erhalten, werden die Funktionen `MPI_Type_size`, `MPI_Type_extent`, `MPI_Type_lb` und `MPI_Type_ub` bereitgestellt.

Während die Funktion `MPI_Get_elements` die gesamte Anzahl der Mitglieder eines zusammengesetzten Typs ermittelt, liefert `MPI_Get_count` nur die Anzahl der Mitglieder zurück, die unmittelbar an diesem Typ beteiligt sind. Besteht bespielsweise ein Typ `Body` aus den Mitgliedern `Position` und `Mass`, dann liefert `MPI_Get_count` 2 zurück. Ist `Mass` vom Typ `int` und `Position` vom Typ `Point3D`, der wie oben drei `double`-Mitglieder hat, dann liefert `MPI_Get_count` 4 zurück: Dreimal `double` und einmal `int`.

6.3 Ein Beispiel zum N-Körper-Problem

Das folgende Code-Fragment ist ein Teil einer parallelen Lösung des N-Körper-Problems. Es zeigt, wie sich ein Struktur-Typ versenden läßt:

```
/*
 Strukturen in MPI, Ausschnitt eines N-Körper-Problems
*/

#include <mpi.h>
#include <stdio.h>
#define NELEM 25

int main(int argc,char *argv[]) {

 int numtasks, rank, source=0, dest, tag=1, i;

 typedef struct {
  float x, y, z;
  float velocity;
  int  n, type;
 } Particle;

 Particle p[NELEM], particles[NELEM];
 MPI_Datatype particletype, oldtypes[2];
 int blockcounts[2];

 MPI_Aint offsets[2], extent;

 MPI_Status stat;

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

 /* Typen vorbereiten */
 offsets[0] = 0;
 oldtypes[0] = MPI_FLOAT;
 blockcounts[0] = 4;

 MPI_Type_extent(MPI_FLOAT, &extent);
 offsets[1] = 4 * extent;
 oldtypes[1] = MPI_INT;
 blockcounts[1] = 2;

 /* neuen Typ anlegen */
 MPI_Type_struct(2, blockcounts, offsets, oldtypes, &particletype);
 MPI_Type_commit(&particletype);

 if (rank == 0) {
  for (i=0; i < NELEM; i++) {
   particles[i].x = i * 1.0;
   particles[i].y = i * -1.0;
   particles[i].z = i * 1.0;
   particles[i].velocity = 0.25;
   particles[i].n = i;
   particles[i].type = i % 2;
  }
  for (i=0; i < numtasks; i++) {
   MPI_Send(particles, NELEM, particletype, i, tag, MPI_COMM_WORLD);
  }
 }

 MPI_Recv(p, NELEM, particletype, source, tag, MPI_COMM_WORLD, &stat);
```

```
 printf("rank= %d   particle coordinates : %3.2f %3.2f %3.2f, velocity :
%3.2f, type : %d %d\n", rank,p[3].x,
p[3].y,p[3].z,p[3].velocity,p[3].n,p[3].type);

 MPI_Type_free(&particletype);
 MPI_Finalize();

}
```

7. Kommunikatoren und Gruppen

Ein Kommunikator in MPI ist eine Menge von Prozessen, die als Einheit zusammengefaßt werden. Es gibt einen speziellen Kommunikator, `MPI_COMM_WORLD`, der alle Prozesse beinhaltet. Kommunikatoren bestehen aus mindestens einem Prozeß und müssen nicht disjunkt sein, d.h. ein Prozeß kann durchaus zu mehreren Kommunikatoren gehören. Ein Kommunikator dient als Entität, auf die eine Kommunikationsoperation (z.B. Verteilen und Sammeln) angewandt wird. Eine Gruppe hingegen ist eine Menge von Prozessen, die nichts mit Kommunikation zu tun hat. Im Gegensatz zu Kommunikatoren kann der Programmierer bestimmen, welcher Prozeß zu einer Gruppe gehört und welcher nicht. Auf Gruppen können Mengenoperationen (Schnitt, Vereinigung, Differenz) angewandt werden, was mit Kommunikatoren nicht direkt möglich ist. Ein Gruppe ist letztlich nicht anderes als eine benutzerdefinierte Ansammlung von Prozessen.

7.1 Erzeugen von Kommunikatoren und Gruppen

Kommunikatoren können nicht explizit erzeugt werden. Vielmehr kann ein Kommunikator nur aus ein einem bestehenden Kommunikator oder einer bestehenden Gruppe erzeugt werden. Die Funktion `MPI_Comm_create` erzeugt einen Kommunikator aus einer bestehenden Gruppe, die Funktion `MPI_Comm_dup` dupliziert einen Kommunikator, und `MPI_Comm_split` teilt einen bestehenden Kommunikator auf. Einem Kommunikator ist automatisch eine Gruppe zugewiesen. Die Funktion `MPI_Comm_group` liefert die einem Kommunikator zugewiesene Gruppe. In MPI werden Kommunikatoren bzw. Gruppen durch die Typen `MPI_Comm` bzw. `MPI_Group` repräsentiert.

7.2 Prozesse in Kommunikatoren und Gruppen

In Kommunikatoren und Gruppen haben Prozesse eine eindeutige Nummer (rank). Diese Nummer ermittelt die Funktion `MPI_Comm_rank` bzw. `MPI_Group_rank`. Auch hat ein Kommunikator bzw. eine Gruppe eine Anzahl an Prozessen. Diese Anzahl liefert die Funktion `MPI_Comm_size` bzw. `MPI_Group_size`. Wenn ein Kommunikator oder eine Gruppe nicht mehr gebraucht wird, kann er/sie mit `MPI_Comm_free` bzw. `MPI_Group_free` aufgelöst werden.

Topologien (siehe Kapitel 5, Topologien mit MPI) sind spezielle Kommunikatoren. Alles, was für Kommunikatoren gilt, gilt auch für Topologien, wobei z.B. das Aufteilen von Topologien mit `MPI_Comm_split` besonderer Deutung bedarf.

7.3 Gruppen

Eine Gruppe bildet eine vom Benutzer explizit erzeugte Zusammenfassung mehrerer Prozesse. Zu jedem Kommunikator gehört eine Gruppe, und eine Gruppe kann auch nur aus einem Kommunikator erstellt werden. Mit der Funktion `MPI_Group_incl` wird eine Gruppe erzeugt, die nur die angegebenen Prozesse (identifiziert durch ihre Nummer [rank] im zugehörigen Kommunikator) enthält. `MPI_Group_excl` erzeugt hingegen eine Gruppe, die alle Prozesse des zugehörigen Kommunikators beinhaltet außer die beim Funktionsaufruf angegebenen Prozesse. So können also Gruppen mit nur den gewünschten Prozessen erzeugt werden. Daneben gibt es noch `MPI_Group_range_excl` und `MPI_Group_range_incl`, die genauso arbeiten, nur beim Funktionsaufruf nicht die einzelnen Prozesse angegeben werden, sondern ganze Bereiche von Prozessen.

Auf Gruppen sind Operationen definiert, wie sie auch für Mengen definiert sind. Man kann den Durchschnitt (`MPI_Group_intersection`), die Vereinigung (`MPI_Group_union`) und die Differenz (`MPI_Group_difference`) bilden. Die Nummern (ranks) der Prozesse einer Gruppe können an die Nummern einer anderen Gruppe mittels `MPI_Group_translate_ranks` angeglichen werden.

7.4 Kommunikation zwischen Kommunikatoren

Kommunikation findet grundsätzlich nur zwischen Prozessen im gleichen Kommunikator statt. Damit sich auch Prozesse zweier verschiedener Kommunikatoren unterhalten können, bietet MPI sogenannte Inter-Kommunikatoren. Diese sind spezielle Kommunikatoren (auch vom Typ `MPI_Comm`), die jedoch keine Sammel- und Verteilungsoperationen unterstützen. Erzeugt wird ein Inter-Kommunikator mit `MPI_Intercomm_create`. Diese Funktion verbindet einen Kommunikator *c1* mit einem anderen Kommunikator *c2*. Die Funktionen `MPI_Comm_remote_size` und `MPI_Comm_remote_group` liefern die Größe von *c2* bzw. der *c2* zugeordnete Gruppe.

Eine Kommunikationsoperation zwischen zwei Prozessen verschiedener Kommunikatoren wird genauso gestartet wie in einem normalen Kommunikator. Man muß aber darauf achten, daß die Nummern der beiden Prozesse dabei nicht im Kontext des Inter-Kommunikators stehen, sondern in den Kontexten beider Kommunikatoren *c1* und *c2*. Es ist also mit Prozeß 0 beim Senden der Prozeß mit der Nummer 0 im Kommunikator *c1* gemeint, wenn der Sender aus Kommunikator *c1* stammt. Befindet sich das Ziel in *c2*, ist beim Empfangen mit Prozeß 0 der Prozeß im Kommunikator *c2* gemeint, der dort die Nummer 0 hat. Sender und Empfänger können hier also die gleiche Nummer haben. Diese Nummern haben beim Senden und Empfangen nur andere Bedeutungen.

Um zu ermitteln, ob ein Kommunikator ein Inter-Kommunikator ist, kann die Funktion `MPI_Comm_test_inter` benutzt werden. Ein Inter-Kommunikator kann mittels `MPI_Intercomm_merge` in einen "normalen" Kommunikator umgewandelt werden.

7.5 Ein Beispiel zu Gruppen

Das folgende Beispiel teilt einen Kommunikator explizit in zwei disjunkte Gruppen auf:

```
/*
 Gruppen in MPI
*/

#include <mpi.h>
#include <stdio.h>
#define NPROCS 8

int main(int argc,char *argv[]) {

 int rank, new_rank, sendbuf, recvbuf, numtasks;
 int ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};

 MPI_Group orig_group, new_group;
 MPI_Comm new_comm;

 /* MPI initialisieren */

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

 if (numtasks != NPROCS) {
  printf("There must be %d processes, exiting ...\n",NPROCS);
  MPI_Finalize();
  exit(0);
 }

 sendbuf = rank;

 /* Gruppe des Kommunikatores MPI_COMM_WORLD ermitteln */

 MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

 /* Gruppe aufteilen in 2 Hälften, diese beiden Hälften sind neue Gruppen */

 if (rank < NPROCS/2) {
  MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
 }
```

```
 else {
  MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
 }

 /* neuen Kommunikator aus der neuen Gruppe erzeugen */

 MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);

 /* und eine Operation auf dem neuen Kommunikator ausführen. */

 MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

 MPI_Group_rank (new_group, &new_rank);
 printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

 MPI_Finalize();

}
```

Die Ausgabe zeigt die Prozesse, die nun in zwei Gruppen unterteilt wurden:

```
rank= 5 newrank= 1 recvbuf= 22
rank= 1 newrank= 1 recvbuf= 6
rank= 3 newrank= 3 recvbuf= 6
rank= 7 newrank= 3 recvbuf= 22
rank= 2 newrank= 2 recvbuf= 6
rank= 6 newrank= 2 recvbuf= 22
rank= 0 newrank= 0 recvbuf= 6
rank= 4 newrank= 0 recvbuf= 22
```

Der Wert `newrank` gibt die Nummer in der neuen Gruppe an.

8. Fehlerbehandlung

Eine Fehlerbehandlung ist bei den meisten Programmen unerläßlich. Beim Ausführen eines Programms kann immer etwas passieren, das man beim Entwurf nicht vorhersehen konnte (Speicherüberlauf etc.). MPI bietet zwei Arten von Fehlerbehandlungen an. Zum einen geben alle MPI-Funktionen (außer `MPI_WTime` und `MPI_WTick`) einen Fehler-Code zurück, zum anderen gibt es die Status-Variable, mit der Informationen über die Ausführung einer MPI-Routine abgefragt werden können.

Sollte es an einem bestimmten Punkt nicht mehr möglich sein, die Situation zu retten, kann mit der Funktion `MPI_Abort` die Ausführung des Programms in den Prozessen eines Kommunikators mit einem Exit-Code beendet werden.

8.1 Fehlerwerte

MPI bietet für die Auswertung von Fehlerwerten sogenannte Error-Handler an. Der Programmierer kann mittels `MPI_Errhandler_set` registriert werden. Es gibt zwei Standard-Error-Handler: `MPI_ERRORS_ARE_FATAL` und `MPI_ERRORS_RETURN`. Standardmäßig ist ersterer registriert, welcher das Programm bei allen Prozessen nach Ausgabe eines Fehlertextes abbricht. Letzterer überläßt es dem Programmierer, sämtliche Fehlerwerte selbst auszuwerten. Neben den Standard-Error-Handlern kann man auch benutzerdefinierte Error-Handler registrieren. Diese stellen Handles dar, die von `MPI_Errhandler_create` aus normalen Funktionen vom Typ `MPI_Handler_function` erzeugt werden. Diese Prozeduren werden dann aufgerufen, wenn ein Fehler auftritt.

Zu einem Zeitpunkt kann nur genau ein Error-Handler registriert sein. Wird also ein Error-Handler registriert, wird automatisch der alte verworfen. Mit der Funktion `MPI_Errhandler_get` kann man ermitteln, welcher Error-Handler gerade registriert ist. Mit `MPI_Errhandler_free` kann ein Handle auf einen benutzerdefinierten Error-Handler gelöscht werden.

Die Fehlerwerte werden von der MPI-Implementation festgelegt und sind somit von Implementation zu

Implementation verschieden. Jeder Fehlerwert ist aber einer Fehlerklasse zugeordnet, diese kann mit `MPI_Error_class` abgefragt werden. Diese Fehlerklasse wiederum ist für alle MPI-Implementationen gleich, so daß man also die Fehlerklasse abfragen sollte und nicht den Fehlerwert. Eine Tabelle mit allen Fehlerklassen findet sich in der Referenz. Zu einem Fehlerwert gehört auch ein aussagekräftiger Fehlertext, der mit `MPI_Error_string` ermittelt werden kann.

8.2 Die Status-Variable

Diverse MPI-Funktionen, darunter viele Kommunikationsoperationen, füllen eine Status-Variable aus. Unter Fortran ist diese Variable eine Reihung der Länge `MPI_STATUS_SIZE`, in C ist der Typ ein Strukturtyp. Dieser Typ hat folgenden Aufbau:

```
typedef struct {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
    int size;
    int reserved[2];
} MPI_Status;
```

Das Mitglied `MPI_SOURCE` gibt die Quelle an, die bei der Kommunikationsoperation beteiligt war, das Mitglied `MPI_TAG` gibt die Kennzeichnung (tag) der Nachricht zurück, die mit der Kommunikationsoperation behandelt worden ist. `MPI_ERROR` gibt den Fehler/Erfolg wieder, der bei dieser Nachricht aufgetreten ist. Das Mitglied `size` schließlich gibt die Größe der Nachricht an.

9. Weitere Konzepte der MPI

Dieses Kapitel beschäftigt sich mit weiteren Funktionen, von MPI.

9.1 Zeitmessung in MPI

MPI bietet eine Zeitmessung an. Damit ist es ohne Umstände möglich, die Laufzeit von Algorithmen zu messen, ohne weitere Bibliotheken laden zu müssen. Die Funktion `MPI_Wtime` ermittelt die Anzahl der Sekunden seit dem letzten Aufruf der Funktion zurück. Die Dauer in Sekunden eines "System-Ticks" ermittelt die Funktion `MPI_Wtick`. Ein "System-Tick" ist die kleinste atomare Zeiteinheit einer Maschine. Diese beiden Funktionen sind die beiden einzigen MPI-Funktionen, bei denen der Rückgabewert kein Fehlerwert ist, sondern der Semantik der Funktion entspricht.

9.2 Matrixmultiplikation mit Zeitmessung

Das folgende Beispiel demonstiert die Zeitmessung anhand einer Matrixmultiplikation:

```
/*
Matrix-Multiplikation in MPI
*/

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

#define NRA 62                  /* Anzahl Zeilen in Matrix A */
#define NCA 15                  /* Anzahl der Spalten in Matrix A (=Anzahl der
Zeilen von Matrix B) */
#define NCB 7                   /* Anzahl der Spalten in Matrix B */
#define MASTER 0
#define FROM_MASTER 1
#define FROM_WORKER 2

int main(int argc, char *argv[]) {

 int numtasks;
 int taskid;
 int numworkers;
```

```
 int source;
 int dest;
 int mtype;
 int rows;
 int averow, extra, offset;
 int i, j, k, rc;
 double a[NRA][NCA];
 double b[NCA][NCB];
 double c[NRA][NCB];
 double starttime, endtime;

 MPI_Status status;

 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
 MPI_Comm_size(MPI_COMM_WORLD,&numtasks);

 if (numtasks < 2 ) {
  printf("Need at least two MPI tasks. Quitting...\n");
  MPI_Abort(MPI_COMM_WORLD, rc);
  exit(1);
 }

 numworkers = numtasks-1;

 /* Code für den Manager */

 if (taskid == MASTER) {
  printf("matrix multiplikation with MPI\n");
  printf("initializing arrays ...\n");
  for (i=0; i<NRA; i++) {
   for (j=0; j<NCA; j++) {
    a[i][j]= i+j;
   }
  }
  for (i=0; i<NCA; i++) {
   for (j=0; j<NCB; j++) {
    b[i][j]= i*j;
   }
  }

  /* Matrizen versenden */

  averow = NRA/numworkers;
  extra = NRA%numworkers;
   // dies ist nötig, da die Anzahl der beteiligten Arbeiter nicht Teiler der
Matrizendimension sein muß

  offset = 0;
  mtype = FROM_MASTER;

  starttime=MPI_Wtime();

  for (dest=1; dest<=numworkers; dest++) {
   rows = (dest <= extra) ? averow+1 : averow;
   printf("Sending %d rows to task %d offset=%d\n",rows,dest,offset);
   MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
   MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
   MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
   MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
   offset = offset + rows;
  }

  /* Ergebnisse empfangen */

  mtype = FROM_WORKER;
```

```
  for (i=1; i<=numworkers; i++) {
   source = i;
   MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
   MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
   MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
MPI_COMM_WORLD, &status);
   printf("Received results from task %d\n",source);
  }

  endtime=MPI_Wtime();

  printf("result :\n");
  for (i=0; i<NRA; i++) {
   printf("\n");
   for (j=0; j<NCB; j++) {
    printf("%6.2f   ", c[i][j]);
   }
  }
  printf("\nIt took %f seconds.\n",endtime-starttime);

 }

 /* Code für die Arbeiter */

 if (taskid > MASTER) {
  mtype = FROM_MASTER;
  MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
  MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
  MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);
  MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);

  for (i=0; i<rows; i++) {
   for (j=0; j<NCB; j++) {
    c[i][j] = 0.0;
    for (k=0; k<NCA; k++) {
     c[i][j]+=(a[i][k] * b[k][j]);
    }
   }
  }

  mtype = FROM_WORKER;
  MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
 }

 MPI_Finalize();

}
```

In der letzten Zeile der Ausgabe ist die benötigte Zeit zu sehen:

```
matrix multiplikation with MPI
initializing arrays ...
Sending 21 rows to task 1 offset=0
Sending 21 rows to task 2 offset=21
Sending 20 rows to task 3 offset=42
Received results from task 1
Received results from task 2
Received results from task 3
result :

  0.00   1015.00   2030.00   3045.00   4060.00   5075.00   6090.00
  0.00   1120.00   2240.00   3360.00   4480.00   5600.00   6720.00
.
```

```
.
.
  0.00   7315.00   14630.00   21945.00   29260.00   36575.00   43890.00
  0.00   7420.00   14840.00   22260.00   29680.00   37100.00   44520.00
It took 0.001913 seconds.

PSIlogger: done
```

9.3 Prüfen auf Nachrichten

Die Empfangsfunktionen aus Kapitel 3 erwarten, daß zum Zeitpunkt ihrer Ausführung (blockierende Varianten) bzw. irgendwann danach (nichtblockierende Varianten) tatsächlich eine Nachricht vorliegt. Es kann aber auch geprüft werden, ob eine Nachricht eingetroffen ist, ohne sie abzuholen. Dazu gibt es die Funktionen `MPI_Probe` (blockierende Version) und `MPI_Iprobe` (nichtblockierende Version). Man beachte, daß diese Funktionen nicht auf eine spezifische Nachricht (spezifiziert durch ein Handle) testen. Dies ist der Unterschied zur Funktion `MPI_Test` und deren Varianten.

9.4 Profiling von MPI

MPI bietet mit der Funktion `MPI_Pcontrol` die Möglichkeit, MPI während der Ausführung zu überwachen.

9.5 Schlüssel und Attribute

Kommunikatoren können mit der Funktion `MPI_Keyval_create` Schlüssel zugeordnet werden. Diesen Schlüsseln können mit `MPI_Attr_put` Werte zugewiesen werden. Gelöscht werden diese Werte mit `MPI_Attr_delete` und abgefragt mit `MPI_Attr_get`.

9.6 Packen von Puffern

Sollte es einmal vorkommen, daß man viele Daten auf einen Rutsch senden und empfangen möchte, der Sende-Puffer (oder der Empfangspuffer) aber zu klein dafür sein sollte, kann man mit der Funktion `MPI_Pack` diese Daten in einen Puffer packen und unter einem gemeinsamen Namen ansprechen, statt jede einzelne Variable zu versenden. Die Funktion `MPI_Unpack` entpackt diese Daten wieder, `MPI_Pack_size` ermittelt die maximale Größe eines Puffers, der nötig ist, um die Daten zu beinhalten.

Der Puffer ist nach dem Packen unter einem neuen Namen ansprechbar (er ist eine gewöhnliche Zeigervariable) und hat für MPI den Typen `MPI_PACKED`.

9.7 Prozessornamen ermitteln

Mit der Funktion `MPI_Get_processor_name` kann MPI den Namen des Prozessors ermitteln, auf dem es ausgeführt wird.

9.8 Version von MPI

Die Funktion `MPI_Get_version` ermitteln die verwendete Version von MPI.

Referenz - Initialisierung

Hinweis: Einige hier aufgeführten Definitionen sind nicht von allen MPI-Implementationen übernommen worden. Außerdem sind einige Definitionen entweder nur für C oder nur für Fortran77 sinnvoll.
Desweiteren sind einige hier ausgeführten Funktionen Wrapper-Funktionen für andere Funktionen, die hier nicht aufgelistet sind.

Initialisierung

```
int MPI_Init(int *argc, char ***argv);
```
Initialisiert MPI.
Parameter:

`argc` - Zeiger auf eine `int`-Variable, die die Anzahl der Argumente angibt
`argv` - Zeiger auf ein `char`-Feld, das die Argumente enthält

Diese Funktion sollte die erste MPI-Funktion sein, die in einem Programm aufgerufen wird. Die Argumente können von der `main`-Prozedur übernommen werden. Die Variable, auf die `argv` zeigt, enthält dann die Anzahl der Zeichen, die hinter dem Programmnamen auf der Kommandozeile angegeben wurden; `argc` selbst zeigt auf die Kommandozeile. MPI kann nur einmal initialisiert werden. Alle Prozesse müssen diese Funktion aufrufen.

In Fortran hat diese Funktion nur ein Argument, nämlich die Rückgabevariable für den Fehlerwert.

`int` **`MPI_Initialized`**`(int *flag);`
Ermittelt, ob MPI schon initialisiert (mit der Funktion `MPI_Init`) wurde.
Parameter:
`flag` - Zeiger auf eine `int`-Variable, in die das Ergebnis eingrtragen wird

In die Variable, auf die `flag` zeigt, wird eine 0 eingetragen wird, falls MPI noch nicht initialisiert wurde, und ein von 0 verschiedener Wert, falls MPI schon initialisiert wurde.

`int` **`MPI_Abort`**`(MPI_Comm comm, int exitcode);`
Bricht die Ausführung von Prozessen ab.
Parameter:
`comm` - Kommunikator, dessen Prozesse abbrechen sollen
`exitcode` - `int`-Wert, der als exitcode der Prozesse zurückgegeben werden soll

Diese Funktion ist nicht zu verwechseln mit `MPI_Finalize`, die MPI kontrolliert beendet.

`int` **`MPI_Finalize`**`(void);`
Beendet die Ausführung von MPI.
Parameter:
keine

Diese Funktion ist nicht zu verwechseln mit `MPI_Abort`, die MPI abbricht. Alle Prozesse müssen diese Funktionen aufrufen. Normalerweise ist diese Funktion die letzte MPI-Funktion, die in einem Programm aufgerufen wird.

`int` **`MPI_Finalized`**`(int *flag);`
Stellt fest, ob MPI schon beendet wurde.
Parameter:
`flag` - Zeiger auf eine `int`-Variable, in die eine 0 eingetragen wird, falls MPI nicht nicht beendet wurde, und ein von 0 verschiedener Wert, falls MPI beendet wurde

Diese Funktion steht erst ab Version 2 zur Verfügung.

Referenz - Senden und Empfangen

Hinweis: Einige hier aufgeführten Definitionen sind nicht von allen MPI-Implementationen übernommen worden. Außerdem sind einige Definitionen entweder nur für C oder nur für Fortran77 sinnvoll.

Desweiteren sind einige hier ausgeführten Funktionen Wrapper-Funktionen für andere Funktionen, die hier nicht aufgelistet sind.

`int` **`MPI_Send`**`(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);`

Sendet Daten blockierend an einen Prozeß.

Parameter:

`sendbuf` - Zeiger auf den Speicherbereich, der gesendet werden soll
`count` - Anzahl der Daten, die gesendet werden sollen
`type` - Datentyp der Daten, die gesendet werden
`dest` - Nummer des Prozesses (rank) im Kommunikator `comm`, der die Daten erhalten soll
`tag` - Kennzeichnung der Operation
`comm` - Kommunikator, in dem die Operation abläft

`int` **`MPI_Send_init`**`(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req);`

Liefert ein Handle auf eine blockierende Sende-Operation.

Parameter:

`sendbuf` - Zeiger auf den Speicherbereich, der gesendet werden soll
`count` - Anzahl der Daten, die gesendet werden sollen
`type` - Datentyp der Daten, die gesendet werden
`dest` - Nummer des Prozesses (rank) im Kommunikator `comm`, der die Daten erhalten soll
`tag` - Kennzeichnung der Operation
`comm` - Kommunikator, in dem die Operation abläft
`req` - Zeiger auf das Handle, das erzeugt wird

Das Handle `req`, das zurückgeliefert wird, wird ab der Stelle des Funktionsaufrufs verwendet, um die Kommunikationsoperation zu repräsentieren.

`int` **`MPI_Isend`**`(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req);`

Liefert ein Handle auf eine normale nichtblockierende Sende-Operation.

Parameter:

`sendbuf` - Zeiger auf den Speicherbereich, der gesendet werden soll
`count` - Anzahl der Daten, die gesendet werden sollen
`type` - Datentyp der Daten, die gesendet werden sollen
`dest` - Nummer des Prozesses (rank) im Kommunikator `comm`, der die Daten empfangen soll
`tag` - Kennzeichnung der Sende-Operation
`comm` - Kommunikator, in dem die Operation ablaufen soll
`req` - Zeiger auf das Handle

Das Handle `req`, das zurückgeliefert wird, wird ab der Stelle des Funktionsaufrufs verwendet, um die Kommunikationsoperation zu repräsentieren.

`int` **`MPI_Bsend`**`(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);`

Startet ein Sende-Operation, bei der die Daten zwischengepuffert werden, um Blockierung zu vermeiden.

Parameter:

`sendbuf` - Speicherbereich, der gesendet werden soll
`count` - Anzahl der Elemente, die gesendet werden sollen
`type` - Typ der Elemente, die gesendet werden sollen
`dest` - Nummer (rank) des Prozesses, an den die Daten gesendet werden sollen
`tag` - Information, mit der die Kommunikation ausgestattet werden kann

comm - Kommunikator, in dem die Operation stattfindet

```
int MPI_Bsend_init(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req);
```
Liefert ein Handle auf eine blockierende Bsend-Operation zurück.
Parameter:
sendbuf - Speicherbereich, der gesendet werden soll
count - Anzahl der Elemente, die gesendet werden sollen
type - Typ der Elemente, die gesendet werden sollen
dest - Nummer (rank) des Prozesses, an den die Daten gesendet werden sollen
tag - Information, mit der die Kommunikation ausgestattet werden kann
comm - Kommunikator, in dem die Operation stattfindet
req - das Handle, das zurückgeliefert wird

Das Handle req, das zurückgeliefert wird, wird ab der Stelle des Funktionsaufrufs verwendet, um die Kommunikationsoperation zu repräsentieren.

```
int MPI_Ibsend(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req);
```
Liefert ein Handle auf eine nichtblockierende Sende-Operation, die gepuffert abläuft.
Parameter:
sendbuf - Zeiger auf den Speicherbereich, der gesendet werden soll
count - Anzahl der Daten, die gesendet werden sollen
type - Datentyp der Daten, die gesendet werden sollen
dest - Nummer des Prozesses (rank) im Kommunikator comm, der die Daten empfangen soll
tag - Kennzeichnung der Sende-Operation
comm - Kommunikator, in dem die Operation ablaufen soll
req - Zeiger auf das Handle

Das Handle req, das zurückgeliefert wird, wird ab der Stelle des Funktionsaufrufs verwendet, um die Kommunikationsoperation zu repräsentieren.

```
unsigned int MPI_BSEND_OVERHEAD;
```
Definiert die Konstante, die angibt, wieviele Bytes MPI bei jedem Puffer als Overhead anfügt.

```
int MPI_Rsend(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);
```
Startet ein blockierende Sende-Operation im "ready mode".
Parameter:
sendbuf - Speicherbereich, der gesendet werden soll
count - Anzahl der Elemente, die gesendet werden sollen
type - Typ der Elemente, die gesendet werden sollen
dest - Nummer (rank) des Prozesses, an den die Daten gesendet werden sollen
tag - Information, mit der die Kommunikation ausgestattet werden kann
comm - Kommunikator, in dem die Operation stattfindet

```
int MPI_Rsend_init(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req);
```

Liefert ein Handle auf eine blockierende Rsend-Operation zurück.
Parameter:
`sendbuf` - Speicherbereich, der gesendet werden soll
`count` - Anzahl der Elemente, die gesendet werden sollen
`type` - Typ der Elemente, die gesendet werden sollen
`dest` - Nummer (rank) des Prozesses, an den die Daten gesendet werden sollen
`tag` - Information, mit der die Kommunikation ausgestattet werden kann
`comm` - Kommunikator, in dem die Operation stattfindet
`req` - das Handle, das zurückgeliefert wird

Das Handle `req`, das zurückgeliefert wird, wird ab der Stelle des Funktionsaufrufs verwendet, um die Kommunikationsoperation zu repräsentieren.

```
int MPI_Irsend(void *sendbuf, int count, MPI_Datatype type, int dest, int tag,
MPI_Comm comm, MPI_Request *req);
```
Liefert ein Handle auf eine nichtblockierende Sende-Operation im "ready mode".
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der gesendet werden soll
`count` - Anzahl der Daten, die gesendet werden sollen
`type` - Datentyp der Daten, die gesendet werden sollen
`dest` - Nummer des Prozesses (rank) im Kommunikator `comm`, der die Daten empfangen soll
`tag` - Kennzeichnung der Sende-Operation
`comm` - Kommunikator, in dem die Operation ablaufen soll
`req` - Zeiger auf das Handle

Das Handle `req`, das zurückgeliefert wird, wird ab der Stelle des Funktionsaufrufs verwendet, um die Kommunikationsoperation zu repräsentieren.

```
int MPI_Ssend(void *sendbuf, int count, MPI_Datatype type, int dest, int tag,
MPI_Comm comm);
```
Sendet Daten synchron und blockierend an einen Prozeß.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der gesendet werden soll
`count` - Anzahl der Daten, die gesendet werden sollen
`type` - Datentyp der Daten, die gesendet werden
`dest` - Nummer des Prozesses (rank) im Kommunikator `comm`, der die Daten erhalten soll
`tag` - Kennzeichnung der Operation
`comm` - Kommunikator, in dem die Operation abläft

```
int MPI_Ssend_init(void *sendbuf, int count, MPI_Datatype type, int dest, int
tag, MPI_Comm comm, MPI_Request *req);
```
Liefert ein Handle auf eine synchrone blockierende Sende-Operation.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der gesendet werden soll
`count` - Anzahl der Daten, die gesendet werden sollen
`type` - Datentyp der Daten, die gesendet werden
`dest` - Nummer des Prozesses (rank) im Kommunikator `comm`, der die Daten erhalten soll
`tag` - Kennzeichnung der Operation
`comm` - Kommunikator, in dem die Operation abläft
`req` - Zeiger auf das Handle, das erzeugt wird

Das Handle `req`, das zurückgeliefert wird, wird ab der Stelle des Funktionsaufrufs verwendet, um die

Kommunikationsoperation zu repräsentieren.

`int` **`MPI_Issend`**`(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req);`
Liefert ein Handle auf eine synchrone nichtblockierende Sende-Operation.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der gesendet werden soll
`count` - Anzahl der Daten, die gesendet werden sollen
`type` - Datentyp der Daten, die gesendet werden sollen
`dest` - Nummer des Prozesses (rank) im Kommunikator `comm`, der die Daten empfangen soll
`tag` - Kennzeichnung der Sende-Operation
`comm` - Kommunikator, in dem die Operation ablaufen soll
`req` - Zeiger auf das Handle

Das Handle `req`, das zurückgeliefert wird, wird ab der Stelle des Funktionsaufrufs verwendet, um die Kommunikationsoperation zu repräsentieren.

`int` **`MPI_Recv`**`(void *recvbuf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status);`
Führt eine blockierende Empfangsoperation aus.
Parameter:
`recvbuf` - Zeiger auf einen Speicherbereich, in den die empfangenen Daten eingetragen werden
`count` - Anzahl der Daten, die empfangen werden sollen
`type` - Typ der Daten, die empfangen werden sollen
`source` - Nummer (rank) des Prozesses, der die Daten gesendet hat
`tag` - Kennzeichnung der Operation
`comm` - Kommunikator, in dem die Operation ablaufen soll
`status` - Zeiger auf eine `MPI_Status`-Variable, in die der Status der Operation eingetragen wird

`int` **`MPI_Recv_init`**`(void *recvbuf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Request *req);`
Liefert ein Handle auf eine blockierende Empfangsoperation.
Parameter:
`recvbuf` - Zeiger auf einen Speicherbereich, in den die empfangenen Daten eingetragen werden
`count` - Anzahl der Daten, die empfangen werden sollen
`type` - Typ der Daten, die empfangen werden sollen
`source` - Nummer (rank) des Prozesses, der die Daten gesendet hat
`tag` - Kennzeichnung der Operation
`comm` - Kommunikator, in dem die Operation ablaufen soll
`req` - Zeiger auf das Handle, das erzeugt wird

Das Handle `req`, das zurückgeliefert wird, wird ab der Stelle des Funktionsaufrufs verwendet, um die Kommunikationsoperation zu repräsentieren.

`int` **`MPI_Irecv`**`(void *recvbuf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Request *req);`
Liefert ein Handle auf eine nichtblockierende Empfangsoperation.
Parameter:
`recvbuf` - Zeiger auf den Speicherbereich, in die die empfangenen Daten eingetragen werden sollen
`count` - Anzahl der Daten, die empfangen werden sollen

`type` - Datentyp der Daten, die empfangen werden sollen
`source` - Nummer des Prozesses (rank) im Kommunikator `comm`, der die Daten gesendet hat
`tag` - Kennzeichnung der Operation
`comm` - Kommunikator, in dem die Operation ablaufen soll
`req` - Zeiger auf das Handle

Das Handle `req`, das zurückgeliefert wird, wird ab der Stelle des Funktionsaufrufs verwendet, um die Kommunikationsoperation zu repräsentieren.

`unsigned int` **`MPI_ANY_SOURCE;`**
Definiert die Konstante, die angibt, daß eine Empfangsoperation von allen Prozessen Nachrichten entgegennehmen kann. Diese Konstante kann als Parameter `source` in allen Empfangsoperationen eingesetzt werden. Sie ist gleichzeitig Wert des Schlüssels `MPI_IO`, falls jeder Prozeß Eingaben und Ausgaben tätigen kann.

`unsigned int` **`MPI_ANY_TAG;`**
Definiert die Konstante, die angibt, daß eine Empfangsoperation Nachrichten mit jeder Kennzeichnung annehmen soll. Diese Konstante kann als Parameter tag in jeder Empfangsoperation benutzt werden. Diese Konstante ist auch der Wert des Schlüssels `MPI_IO`, wenn jeder Prozeß Eingaben und Ausgaben tätigen kann.

`int` **`MPI_Start`**`(MPI_Request *req);`
Startet eine Operation, deren Handle übergeben wird.
Parameter:
`req` - Zeiger auf das Handle

Das Handle `req` der Operation muß zuvor mit einer der Funktionen `MPI_Bsend_init`, `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Send_init`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend` oder `MPI_Isend` ermittelt worden sein.

`int` **`MPI_Startall`**`(int count, MPI_Request *req);`
Startet Operationen, deren Handle übergeben werden.
Parameter:
`count` - Gröszlig;e des Feldes `req`
`req` - Feld der Handles

Die Handles müssen; zuvor mit einer der Funktionen `MPI_Bsend_init`, `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Send_init`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend` oder `MPI_Isend` ermittelt worden sein.

`int` **`MPI_Test`**`(MPI_Request *req, int *flag, MPI_Status *status);`
Testet ein Handle auf Beendigung der entsprechenden Operation.
Parameter:
`req` - Zeiger auf das Handle
`flag` - Zeiger auf eine `int`-Variable, in die das Ergebnis eingetragen wird
`status` - Zeiger auf eine `MPI_Status`-Variable, in die der Status der Operation eingetragen wird

Das Handle der Operation muß zuvor mit einer der Funktionen `MPI_Bsend_init`, `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Send_init`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend` oder `MPI_Isend`

ermittelt worden sein. In die Variable, auf die `flag` zeigt, wird eine 0 eingetragen, falls die Operation noch nicht beendet wurde, und ein von 0 verschiedener Wert, falls die Operation beendet wurde.

`int` **`MPI_Testall`**`(int count, MPI_Request *req, int *flag, MPI_Status *status);`
Testet Handles von Kommunikationsoperationen auf deren Beendigung.
Parameter:
`count` - Größe der Felder `req` und `status`
`req` - Feld von Handles, die getestet werden sollen
`flag` - Zeiger auf eine `int`-Variable, in die das Ergebnis eingetragen wird
`status` - Feld von `MPI_Status`-Variablen, in dessen Elemente der Status der jeweiligen Operationen eingetragen wird

Die Handles müssen zuvor mit einer der Funktionen `MPI_Bsend_init`, `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Send_init`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend` oder `MPI_Isend` ermittelt worden sein. Die Variable, auf die `flag` zeigt, wird dann und nur dann auf einen von 0 verschiedenen Wert gesetzt, falls alle Operationen beendet wurden.

`int` **`MPI_Testany`**`(int count, MPI_Request *req, int *index, int *flag, MPI_Status *status);`
Testet Handles von Kommunikationsoperationen auf deren Beendigung.
Parameter:
`count` - Größe der Felder `req` und `status`
`req` - Feld von Handles, die getestet werden sollen
`index` - Zeiger auf eine `int`-Variable, in die eingetragen wird, welche Operation beendet wurde
`flag` - Zeiger auf eine `int`-Variable, in die das Ergebnis eingetragen wird
`status` - Feld von `MPI_Status`-Variablen, in dessen Elemente der Status der jeweiligen Operationen eingetragen wird

Die Handles müssen zuvor mit einer der Funktionen `MPI_Bsend_init`, `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Send_init`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend` oder `MPI_Isend` ermittelt worden sein. Die Variable, auf die `flag` zeigt, wird dann und nur dann auf einen von 0 verschiedenen Wert gesetzt, falls mindestens eine der Operationen beendet wurde. In diesem Fall enthät die Variable, auf die `index` zeigt, die Nummer des Elements von `req`, dessen Operation beendet wurde. Ist `*flag` also nicht 0, ist die Operation `req[*index]` beendet worden.

`int` **`MPI_Testsome`**`(int incount, MPI_Request *req, int *outcount, int *indices, MPI_Status *status);`
Testet Handles von Kommunikationsoperationen auf deren Beendigung.
Parameter:
`incount` - Größe des Feldes `req`
`req` - Feld von Handles, die getestet werden sollen
`outcount` - Zeiger auf eine `int`-Variable, in die die Größe der Felder `indices` und `status` eingetragen wird
`indices` - `int`-Feld, in das die Indices der Operationen eingetragen werden, die beendet wurden
`status` - Feld von `MPI_Status`-Variable, in dessen Elemente der Status der beendeten Operationen eingetragen wird

Die Handles müssen zuvor mit einer der Funktionen `MPI_Bsend_init`, `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Send_init`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend` oder `MPI_Isend` ermittelt worden sein.

`int` **`MPI_Wait`**`(MPI_Request *req, MPI_Status *status);`
Wartet auf die Beendigung einer Operation, die durch ein Handle repräsentiert wird. **Parameter:**
`req` - Zeiger auf das Handle
`status` - Zeiger auf eine `MPI_Status`-Variable, in die der Status der Operation eingetragen wird

Das Handle der Operation muß zuvor mit einer der Funktionen `MPI_Bsend_init`, `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Send_init`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend` oder `MPI_Isend` ermittelt worden sein. Diese Funktion blockiert solange, bis die Operation tatsächlich beendet wurde.

`int` **`MPI_Waitall`**`(int count, MPI_Request *req, MPI_Status *status);`
Wartet auf die Beendigung aller Operationen, die von den ¨bergebenen Handles repräsentiert werden.
Parameter:
`count` - Größe der Felder `req` und `status`
`req` - Feld von Handles, die getestet werden sollen
`status` - Feld von `MPI_Status`-Variablen, in dessen Elemente der Status der jeweiligen Operationen eingetragen wird

Die Handles müssen zuvor mit einer der Funktionen `MPI_Bsend_init`, `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Send_init`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend` oder `MPI_Isend` ermittelt worden sein. Diese Funktion blockiert solange, bis alle Operationen, deren Handles übergeben wurden, tatsächlich beendet wurden.

`int` **`MPI_Waitany`**`(int count, MPI_Request *req, int *index, MPI_Status *status);`
Wartet auf die Beendigung einer der Operationen, die von allen ¨bergebenen Handles repräsentiert werden.
Parameter:
`count` - Größe der Felder `req` und `status`
`req` - Feld von Handles, die getestet werden sollen
`index` - Zeiger auf eine `int`-Variable, in die eingetragen wird, welche Operation beendet wurde
`status` - Feld von `MPI_Status`-Variablen, in dessen Elemente der Status der jeweiligen Operationen eingetragen wird

Die Handles müssen zuvor mit einer der Funktionen `MPI_Bsend_init`, `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Send_init`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend` oder `MPI_Isend` ermittelt worden sein. Wird eine Operation beendet, enthät die Variable, auf die `index` zeigt, die Nummer des Elements von `req`, dessen Operation beendet wurde; die Operation `req[*index]` wurde dann beendet. Diese Funktion blockiert solange, bis eine der Operationen, deren Handles übergeben wurden, tatsächlich beendet wurde.

`int` **`MPI_Waitsome`**`(int incount, MPI_Request *req, int *outcount, int *indices, MPI_Status *status);`
Wartet auf die Beendigung mindestens einer der Operationen, die von allen ¨bergebenen Handles repräsentiert werden.
Parameter:
`incount` - Größe des Feldes `req`
`req` - Feld von Handles, die getestet werden sollen
`outcount` - Zeiger auf eine `int`-Variable, in die die Größe der Felder `indices` und `status` eingetragen wird
`indices` - `int`-Feld, in das die Indices der Operationen eingetragen werden, die beendet wurden
`status` - Feld von `MPI_Status`-Variable, in dessen Elemente der Status der beendeten Operationen eingetragen wird

Die Handles müssen zuvor mit einer der Funktionen `MPI_Bsend_init`, `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Send_init`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend` oder `MPI_Isend` ermittelt worden sein. Diese Funktion blockiert solange, bis mindestens eine der Operationen, deren Handles

übergeben wurden, tatsächlich beendet wurde.

`int` **`MPI_Cancel`**`(MPI_Request *req);`
Bricht eine Operation ab.
Parameter: `req` - das Handle der Operation
Das Handle der Operation muß zuvor mit einer der Funktionen `MPI_Bsend_init`, `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Send_init`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend` oder `MPI_Isend` ermittelt worden sein.

`int` **`MPI_Test_cancelled`**`(MPI_Status *status, int *flag);`
Testet, ob eine Operation angebrochen wurde.
Parameter: `status` - Zeiger auf eine `MPI_Status`-Variable, die von einer Test- oder Wartefunktion ausgefüllt wurde
`flag` - Zeiger auf eine `int`-Variable, in die eine 1 geschrieben wird, falls die Operation abgebrochen wurde, und eine 0, falls nicht

`int` **`MPI_Request_free`**`(MPI_Request *req);`
Zerstört ein Handle und gibt die von ihm belegten Ressourcen wieder frei.
Parameter:
`req` - Zeiger auf das Handle

`unsigned int` **`MPI_REQUEST_NULL;`**
Definiert die Konstante, die das Handle angibt, das eigentlich kein Handle ist.

`int` **`MPI_Sendrecv`**`(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status);`
Führt ein gleichzeitiges Senden und Empfangen an.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der versendet werden soll
`sendcount` - Anzahl der Daten, die versendet werden sollen
`sendtype` - Typ der Daten, die versendet werden sollen
`dest` - Nummer (rank) des Prozesses im Kommunikator `comm`, der die Daten erhalten soll
`sendtag` - Kennzeichnung der Sende-Operation
`recvbuf` - Zeiger auf den Speicherbereich, in den die empfangenen Daten eingetragen werden
`recvcount` - Anzahl der Daten, die empfangen werden sollen
`recvtype` - Typ der Daten, die empfangen werden sollen
`source` - Nummer (rank) des Prozesses, der die Daten gesendet hat
`recvtag` - Kennzeichnung der Empfangsoperation
`comm` - Kommunikator, in dem die Operation ablaufen soll
`status` - Zeiger auf eine `MPI_Status`-Variable, die den Status der Empfangsoperation anzeigt

Diese Funktion ist äquivalent zu einer Nacheinanderausführung von `MPI_Send` und `MPI_Recv`.

`int` **`MPI_Sendrecv_replace`**`(void *buffer, int count, MPI_Datatype type, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status);`

Führt ein gleichzeitiges Senden und Empfangen an, wobei Sende-Puffer und Empfangspuffer gleich sind.
Parameter:
`buffer` - Zeiger auf den Speicherbereich, der versendet werden soll und in den anschließend die empfangenen Daten eingetragen werden sollen
`count` - Anzahl der Daten, die versendet und empfangen werden sollen
`type` - Typ der Daten, die versendet und empfangen werden sollen
`dest` - Nummer (rank) des Prozesses im Kommunikator `comm`, der die Daten erhalten soll
`sendtag` - Kennzeichnung der Sende-Operation
`source` - Nummer (rank) des Prozesses, der die Daten gesendet hat
`recvtag` - Kennzeichnung der Empfangsoperation
`comm` - Kommunikator, in dem die Operation ablaufen soll
`status` - Zeiger auf eine `MPI_Status`-Variable, die den Status der Empfangsoperation anzeigt

`int` **`MPI_Buffer_attach`**`(void *buffer, int size);`
Macht einen Speicherbereich MPI als Puffer bekannt.
Parameter:
`buffer` - Zeiger auf den Speicherbereich
`size` - Größe des Speicherbereichs in Bytes

Nachdem ein Speicherbereich MPI als Puffer bekannt gemacht wurde, kann er mit `MPI_Bsend` versendet werden.

`int` **`MPI_Buffer_detach`**`(void *buffer, int *size);`
Gibt einen Puffer wieder frei.
Parameter:
`buffer` - Zeiger auf den Speicherbereich, der den Puffer enthält
`size` - `int`-Variable, in die MPI die Größe des Puffers einträgt

Der Puffer muß vorher mit `MPI_Buffer_attach` zugewiesen worden sein.

Referenz - Verteilung und Sammeln

Hinweis: Einige hier aufgeführten Definitionen sind nicht von allen MPI-Implementationen übernommen worden. Außerdem sind einige Definitionen entweder nur für C oder nur für Fortran77 sinnvoll.
Desweiteren sind einige hier ausgeführten Funktionen Wrapper-Funktionen für andere Funktionen, die hier nicht aufgelistet sind.
`int` **`MPI_Barrier`**`(MPI_Comm comm);`
Diese Funktion synchronisiert alle Prozesse im Kommunikator.
Parameter:
`comm` - Kommunikator, dessen Prozesse synchronisiert werden sollen

Alle Prozesse im Kommunikator `comm` warten beim Aufruf dieser Funktion, bis alle diese Prozesse eben diesen Aufruf erreichen. Alle Prozesse im Kommunikator `comm` müssen diese Funktion aufrufen.

`int` **`MPI_Bcast`**`(void *buffer, int count, MPI_Datatype type, int root, MPI_Comm comm);`
Versendet einen Speicherbereich an alle Prozesse im Kommunikator.
Parameter:
`buffer` - Zeiger auf den Speicherbereich, der versendet werden soll
`count` - Anzahl der Elemente, die versendet werden sollen
`type` - Typ der Elemente, die versendet werden sollen

`root` - Besitzer der Kommunikationsoperation
`comm` - Kommunikator, dessen Prozesse die Daten empfangen sollen

Alle Prozesse im Kommunikator `comm` müssen diese Funktion aufrufen. Der Parameter `root` gibt die Nummer (rank) des Besitzers im Kommunikator `comm` an. Beim Besitzer zeigt `buffer` auf den Speicherbereich, der die Daten enthält, die versendet werden sollen. Bei allen anderen Prozessen in `comm` zeigt `buffer` auf den Speicherbereich, wo die Daten eingetragen werden sollen.

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```
Sammelt Daten von allen Prozessen ein.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der gesendet wird
`sendcount` - Anzahl der Daten, die gesendet werden
`sendtype` - Typ der Daten, die gesendet werden `recvbuf` - Zeiger auf den Speicherbereich, in dem die Ergebnisse gesammelt werden
`recvcount` - Anzahl der erwarteten Ergebnisse
`recvtype` - Typ der Daten, die empfangen werden
`root` - Besitzer dieser Kommunikationsoperation
`comm` - Kommunikator, dessen Prozesse an dieser Operation teilnehmen

Alle Prozesse im Kommunikator `comm` müssen diese Funktion aufrufen.

```
int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root,
MPI_Comm comm);
```
Sammelt Daten von allen Prozessen ein.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der gesendet wird
`sendcount` - Anzahl der Daten, die gesendet werden
`sendtype` - Typ der Daten, die gesendet werden `recvbuf` - Zeiger auf den Speicherbereich, in dem die Ergebnisse gesammelt werden
`recvcounts` - Feld der Anzahlen der erwarteten Ergebnisse
`displs` - `int`-Feld, dessen Einträge angeben, an welche Position in `*recvbuf` die Daten geschrieben werden sollen
`recvtype` - Typ der Daten, die empfangen werden
`root` - Besitzer dieser Kommunikationsoperation
`comm` - Kommunikator, dessen Prozesse an dieser Operation teilnehmen

Der Eintrag `recvcounts[i]` legt fest, wieviele Daten Prozeß *i* im Kommunikator `comm` empfangen soll. Der Eintrag `displs[i]` gibt an, wo Prozeß *i* die empfangenen Daten in `*recvbuf` eintragen soll. Diese Funktion arbeitet genauso wie `MPI_Gather` mit der Ausnahme, daß hier die Positionen festgelegt werden können, an denen die Daten abgelegt werden sollen. Alle Prozesse im Kommunikator `comm` müssen diese Funktion aufrufen.

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);
```
Sammelt Daten von allen Prozessen im Kommunikator und verteilt sie an alle Prozesse im Kommunikator.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der gesendet werden soll
`sendcount` - Anzahl der Einheiten, die versendet werden sollen
`sendtype` - Typ der Daten, die versendet werden sollen
`recvbuf` - Zeiger auf den Speicherbereich, in den die empfangenen Daten eingetragen werden sollen

`recvcount` - Anzahl der Einheiten, die empfangen werden sollen
`recvtype` - Typ der Daten, die empfangen werden sollen
`comm` - Kommunikator, dessen Prozesse an dieser Operation teilnehmen sollen

Diese Funktion sammelt von allen Prozessen im Kommunikator `comm` Daten und macht sie allen Prozessen im Kommunikator `comm` bekannt. Alle Prozesse im Kommunikator müssen diese Funktion aufrufen.

```
int MPI_Allgatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm);
```
Sammelt Daten von allen Prozessen im Kommunikator und verteilt sie an alle Prozesse im Kommunikator.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der gesendet werden soll
`sendcount` - Anzahl der Einheiten, die versendet werden sollen
`sendtype` - Typ der Daten, die versendet werden sollen
`recvbuf` - Zeiger auf den Speicherbereich, in den die empfangenen Daten eingetragen werden sollen
`recvcounts` - `int`-Feld mit den Anzahlen der Einheiten, die empfangen werden sollen
`displs` - `int`-Feld, dessen Einträge angeben, an welche Position in `*recvbuf` die Daten geschrieben werden sollen
`recvtype` - Typ der Daten, die empfangen werden sollen
`comm` - Kommunikator, dessen Prozesse an dieser Operation teilnehmen sollen

Der Eintrag `recvcounts[i]` legt fest, wieviele Daten Prozeß *i* im Kommunikator `comm` empfangen soll. Der Eintrag `displs[i]` gibt an, wo in `*recvbuf` Prozeß *i* die empfangenen Daten eintragen soll. Diese Funktion sammelt von allen Prozessen im Kommunikator `comm` Daten und macht sie allen Prozessen im Kommunikator `comm` bekannt. Alle Prozesse im Kommunikator müssen diese Funktion aufrufen. Der Unterschied zu `MPI_Allgather` ist, daß hier noch die Positionen im Empfangspuffer angegeben werden können.

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype type,
MPI_Op op, int root, MPI_Comm comm);
```
Sammelt Daten von allen Prozessen im Kommunikator ein und verrechnet sie.
Parameter:
`sendbuf` - Zeiger auf einen Speicherbereich, der die Daten enthält, die gesendet und verrechnet werden sollen
`recvbuf` - Zeiger auf einen Speicherbereich, der das Ergebnis der Berechnung enthält
`count` - Anzahl der Daten, die gesendet und verrechnet werden sollen
`type` - Typ der Daten, die versendet und verrechnet werden sollen
`op` - Funktion, mit der die Daten verrechnet werden sollen
`root` - Nummer (rank) des Prozesses im Kommunikator `comm`, der das Ergebnis erhalten soll
`comm` - Kommunikator, in dem die Operation ablaufen soll

Alle Prozesse im Kommunikator `comm` müssen diese Funktion aufrufen. Die Funktion `op` muß den Typ `type` verarbeiten können.

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype type,
MPI_Op op, MPI_Comm comm);
```
Sammelt Daten von allen Prozessen im Kommunikator, verrechnet sie und sendet das Ergebnis an alle Prozesse im Kommunikator.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, dessen Elemente Teil der Berechnung werden sollen
`recvbuf` - Zeiger auf den Speicherbereich, in den das Ergebnis eingetragen werden soll
`count` - Anzahl der Elemente, die verrechnet werden sollen
`type` - Typ der Daten, die verrechnet werden sollen
`op` - Handle auf eine Operation, die angibt, was mit den Daten geschehen soll

comm - Kommunikator, dessen Prozesse an dieser Operation teilnehmen

Alle Prozesse im Kommunikator comm müssen diese Funktion aufrufen. Die Operation op muß auf den Typ type angewandt werden können.

```
int MPI_Reduce_scatter(void *sendbuf, void *recvbuf, int *counts, MPI_Datatype type, MPI_Op op, MPI_Comm comm);
```

Führt eine Reduce-Operation aus und verteilt das Ergebnis an alle Prozesse im Kommunikator.

Parameter:

sendbuf - Zeiger auf den Speicherbereich, der zur Berechnung der Reduce-Operation beitragen soll
recvbuf - Zeiger auf den Speicherbereich, der ein Teil des Ergebnisses enthält
counts - int-Feld, das angibt, wieviele Teile des Ergebnisses ein Prozeß enthalten soll
type - Datentyp der Daten, die zur Berechnung beitragen sollen
op - Operation, mit der die Berechnung durchgeführt werden soll
comm - Kommunikator, dessen Prozesse an dieser Operation teilnehmen

Der Datentyp type muß von der Operation op verarbeitet werden können. Alle Prozesse im Kommunikator comm müssen diese Funktion aufrufen.

```
int MPI_Scan(void *v1, void *v2, int count, MPI_Datatype t, MPI_Op op, MPI_Comm c);
```

Sammelt Daten von einem Teil aller Prozessen im Kommunikator und verrechnet sie.

Parameter:

sendbuf - Zeiger auf den Speicherbereich, dessen Elemente Teil der Berechnung werden sollen
recvbuf - Zeiger auf den Speicherbereich, in den das Ergebnis eingetragen werden soll
count - Anzahl der Elemente, die verrechnet werden sollen
type - Typ der Daten, die verrechnet werden sollen
op - Handle auf eine Operation, die angibt, was mit den Daten geschehen soll
comm - Kommunikator, dessen Prozesse an dieser Operation teilnehmen

Alle Prozesse im Kommunikator comm müssen diese Funktion aufrufen. Die Operation op muß auf den Typ type angewandt werden können.

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

Erzeugt eine neue benutzerdefinierte Operation, die bei reduce-Operationen eingesetzt werden kann.

Parameter:

uf - Zeiger auf eine Funktion vom Prototyp MPI_User_function, die benutzt werden soll
commute - int-Variable, die angibt, ob die definierte Funktion uf kommutativ ist
op - Zeiger auf die benutzerdefinierte Funktion, die erzeugt wird

Benutzerdefinierte Funktionen sollten so geschrieben sein, daß sie kommutativ sind. Dann kann MPI Optimierungen vornehmen.

```
int MPI_Op_free(MPI_Op *op);
```

Zerstört den Verweis auf eine benutzerdefinierte Operation und gibt den durch sie belegten Speicherplatz frei.

Parameter:

op - Zeiger auf die Operation

Mit dieser Funktion können nur benutzerdefinierte Funktionen zerstört werden.

`typedef void **MPI_User_function**(void *in, void *out, int *length, MPI_Datatype *type);`
Definiert einen Callback-Funktionsprototypen, dessen Instanzen (Funktionen von diesem Typ) mittels `MPI_Op_create` in eine benutzerdefinierte Funktionen für reduce-Operationen umgewandelt werden können.
Parameter:
`in` - Zeiger auf einen Speicherbereich, in den MPI vor dem Funktionsaufruf die Werte, die verrechnet werden sollen, einträgt
`out` - Zeiger auf einen Speicherberech, in den das/die Ergebnis(se) der Berechnung von der Funktion eingetragen werden muß
`length` - Zeiger auf einen Speicherbereich, in den MPI die Anzahl der Elemente in `*in` vor dem Funktionsaufruf einträgt und in den die Funktion die Anzahl der Elemente in `*out` einträgt
`type` - Zeiger auf eine Variable vom Typ `MPI_Datatype`, in die MPI den Typ der Elemente in `*in` vor dem Funktionsaufruf einträgt und in den die Funktion die Anzahl der Elemente in `*out` einträgt

`unsigned int **MPI_OP_NULL;**`
Definiert die Konstante, die die Operation angibt, die eigentlich keine ist.

`unsigned int **MPI_MAX;**`
Definiert die Konstante, die die Operation angibt, die das Maximum berechnet.

`unsigned int **MPI_MIN;**`
Definiert die Konstante, die die Operation angibt, die das Minimum berechnet.

`unsigned int **MPI_MAXLOC;**`
Definiert die Konstante, die die Operation angibt, die das Maximum und dessen Position berechnet. Siehe dazu die Datentypen MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_INT_INT, MPI_SHORT_INT und MPI_LONG_DOUBLE_INT für C und MPI_2REAL, MPI_2DOUBLE_PRECISION und MPI_2INTEGER für Fortran77.

`unsigned int **MPI_MINLOC;**`
Definiert die Konstante, die die Operation angibt, die das Minimum und dessen Position berechnet. Siehe dazu die Datentypen MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_INT_INT, MPI_SHORT_INT und MPI_LONG_DOUBLE_INT für C und MPI_2REAL, MPI_2DOUBLE_PRECISION und MPI_2INTEGER für Fortran77.

`unsigned int **MPI_SUM;**`
Definiert die Konstante, die die Operation angibt, die die Summe berechnet.

`unsigned int **MPI_PROD;**`
Definiert die Konstante, die die Operation angibt, die das Produkt berechnet.

`unsigned int **MPI_LAND;**`

Definiert die Konstante, die die Operation angibt, die das logische UND berechnet.

`unsigned int` **`MPI_BAND;`**
Definiert die Konstante, die die Operation angibt, die das bitweise UND berechnet.

`unsigned int` **`MPI_LOR;`**
Definiert die Konstante, die die Operation angibt, die das logische ODER berechnet.

`unsigned int` **`MPI_BOR;`**
Definiert die Konstante, die die Operation angibt, die das bitweise ODER berechnet.

`unsigned int` **`MPI_LXOR;`**
Definiert die Konstante, die die Operation angibt, die das logische ENTWEDER ODER berechnet.

`unsigned int` **`MPI_BXOR;`**
Definiert die Konstante, die die Operation angibt, die das bitweise ENTWEDER ODER berechnet.

`int` **`MPI_Scatter`**`(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);`
Verteilt Daten an alle Prozesse im Kommunikator.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der die Daten enthält, die verteilt werden sollen
`sendcount` - Anzahl der Daten, die verteilt werden sollen
`sendtype` - Datentyp der Daten, die verteilt werden sollen
`recvbuf` - Zeiger auf einen Speicherbereich, in den die Daten, die empfangen werden, eingetragen werden sollen
`recvcount` - Anzahl der Daten, die empfangen werden sollen
`recvtype` - Datentyp der Daten, die empfangen werden sollen
`root` - Nummer (rank) des Prozesses im Kommunikator `comm`, der die Daten verteilt
`comm` - Kommunikator, in dem die Operation ablaufen soll

Alle Prozesse im Kommunikator `comm` müssen diese Funktion aufrufen.

`int` **`MPI_Scatterv`**`(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);`
Verteilt Daten an alle Prozesse im Kommunikator.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der die Daten enthält, die verteilt werden sollen
`sendcounts` - `int`-Feld mit den Anzahlen der Daten, die verteilt werden sollen
`displs` - `int`-Feld, dessen Einträge angeben, an welcher Position in `*sendbuf` die Daten stehen, die jeder Prozeß senden soll
`sendtype` - Datentyp der Daten, die verteilt werden sollen
`recvtype` - Zeiger auf einen Speicherbereich, in den die Daten, die empfangen werden, eingetragen werden sollen
`recvcount` - Anzahl der Daten, die empfangen werden sollen

`recvtype` - Datentyp der Daten, die empfangen werden sollen
`root` - Nummer (rank) des Prozesses im Kommunikator `comm`, der die Daten verteilt
`comm` - Kommunikator, in dem die Operation ablaufen soll

Der Eintrag `sendcounts[i]` legt fest, wieviele Daten Prozeß *i* im Kommunikator `comm` senden soll. Der Eintrag `displs[i]` gibt an, wo in `*sendbuf` die Daten liegen, die Prozeß *i* senden soll. Alle Prozesse im Kommunikator `comm` müssen diese Funktion aufrufen. Diese Funktion arbeitet bis auf die Angabe der Speicherorte für die gesendeten Daten genau wie `MPI_Scatter`.

```
int MPI_Alltoall(void *sendbuf , int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);
```
Sendet Daten von jedem Prozeß an jeden Prozeß.
Parameter:
`sendbuf` - Zeiger auf den Speicherbereich, der gesendet werden soll
`sendcount` - Anzahl der Einheiten, die versendet werden sollen
`sendtype` - Typ der Daten, die versendet werden sollen
`recvbuf` - Speicherbereich, in den die empfangenen Daten eingetragen werden sollen
`recvcount` - Anzahl der Einheiten, die empfangen werden sollen
`recvtype` - Typ der Daten, die empfangen werden sollen
`comm` - Kommunikator, dessen Prozesse an dieser Operation teilnehmen sollen
Alle Prozesse im Kommunikator `comm` müssen diese Funktion aufrufen. Im Gegensatz zu `MPI_Allgather` werden die Daten, die versendet werden, nicht zusammengefaßt.

```
int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *senddispls,
MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *recvdispls,
MPI_Datatype recvtype, MPI_Comm comm);
```
Sendet Daten von jedem Prozeß an jeden Prozeß.
Parameter:
`sendbuf` - Speicherbereich, der gesendet werden soll
`sendcounts` - `int`-Feld mit den Anzahlen der Einheiten, die versendet werden sollen
`senddispls` - `int`-Feld, das die absoluten Positionen das Daten im Sendepuffer angibt
`sendtype` - Typ der Daten, die versendet werden sollen
`recvbuf` - Speicherbereich, in den die empfangenen Daten eingetragen werden sollen
`recvcounts` - `int`-Feld mit den Anzahlen der Einheiten, die empfangen werden sollen
`recvdispls` - `int`-Feld, das die absoluten Positionen im Empfangspuffer angibt, wo die Daten abgelegt werden sollen
`recvtype` - Typ der Daten, die empfangen werden sollen
`comm` - Kommunikator, dessen Prozesse an dieser Operation teilnehmen sollen

Diese Funktion arbeitet genau wie `MPI_Alltoall` mit dem Unterschied, daß hier auch die absoluten Positionen der Daten abgegeben werden können. Alle Prozesse im Kommunikator `comm` müssen diese Funktion aufrufen.

Referenz - Topologien

Hinweis: Einige hier aufgeführten Definitionen sind nicht von allen MPI-Implementationen übernommen worden. Außerdem sind einige Definitionen entweder nur für C oder nur für Fortran77 sinnvoll.
Desweiteren sind einige hier ausgeführten Funktionen Wrapper-Funktionen für andere Funktionen, die hier nicht aufgelistet sind.
```
int MPI_Cart_create(MPI_Comm cOld, int nDims, int *dims, int *periods, int
reorder, MPI_Comm *cNew);
```
Erzeugt ein kartesisches Gitter.
Parameter:

`cOld` - Kommunikator, der in ein kartesisches Gitter umgewandelt werden soll
`nDims` - Anzahl der Dimensionen
`dims` - `int`-Feld, das die Ausdehnung des Gitters in jeder Dimension enthält
`periods` - `int`-Feld, das angibt, ob ein echtes Gitter oder ein Torus erzeugt wird
`reorder` - Indikator, der angibt, ob MPI die Prozesse für das Gitter neu ordnen darf (1) oder nicht darf (0)
`cNew` - Zeiger auf einen Kommunikator, der nach der Ausführung dieser Funktion auf das Gitter zeigt

Ein Wert `dims[i]` gibt die Ausdehnung des Gitters in der *i*-ten Dimension an. Ist `periods[i]` ungleich 0, dann wird der "letzte" Prozeß in der Dimension *i* mit dem "ersten" Prozeß verbunden, so daß statt eines echten Gitters ein Torus entsteht.

`int` **`MPI_Cart_get`**`(MPI_Comm comm, int maxdim, int *dims, int *periods, int *coords);`
Ermittelt Informationen über ein kartesisches Gitter.
Parameter:
`comm` - Kommunikator, der auf das Gitter zeigt
`maxdim` - Grösz lig;e der Felder dims und periods
`dims` - `int`-Feld, in das die Ausdehnungen des Gitters in jeder Dimension eingetragen werden
`periods` - `int`-Feld, in das eingetragen wird, ob `comm` ein Torus oder ein echtes Gitter ist
`coords` - `int`-Feld, in das die Koordinaten des aufrufenden Prozesses eingetragen werden

Ein Wert `periods[i]` wird mit 1 aufgefüllt, falls in der Dimension *i* der "letzte" Prozeß mit dem "ersten" Prozeß verbunden ist, wenn also `comm` ein Torus in dieser Dimension ist. Andernfalls ist `periods[i]` gleich 0.

`int` **`MPI_Cartdim_get`**`(MPI_Comm comm, int *dims);`
Ermittelt die Ausdehnung eines kartesischen Gitters in allen Dimensionen
Parameter:
`comm` - Kommunikator, der auf das Gitter zeigt
`dims` - `int`-Feld, in das die Ausdehnungen in allen Dimensionen eingetragen wird

`int` **`MPI_Cart_map`**`(MPI_Comm comm, int ndims, int *dims, int *periods, int *newrank);`
Projiziert den aufrufenden Prozeß auf ein kartesisches Gitter.
Parameter:
`comm` - Kommunikator, der das Gitter darstellt
`ndims` - Anzahl der Dimensionen von `comm`
`dims` - `int`-Feld, das die Ausdehnung von `comm` in jeder Dimension beinhaltet
`periods` - `int`-Feld, das angibt, ob eine Dimension ein Ring ist (1) oder nicht (0)
`newrank` - Zeiger auf eine `int`-Variable, in die die neue Nummer (rank) des aufrufenden Prozesses eingetragen wird

`int` **`MPI_Cart_rank`**`(MPI_Comm comm, int *coords, int *rank);`
Ermittelt die Nummer (rank) zu einem Prozeß in einem kartesischen Gitter, der durch seine Koordinaten spezifiziert wird.
Parameter:
`comm` - Kommunikator, der auf das Gitter zeigt
`coords` - `int`-Feld, das die Koordinaten enthält
`rank` - Zeiger auf eine `int`-Variable, in die die Nummer eingetragen wird

`int` **`MPI_Cart_coords`**`(MPI_Comm comm, int rank, int maxdim, int *coords);`
Ermittelt die Koordinaten eines Prozesses in einem kartesischen Gitter.
Parameter: `comm` - Kommunikator, der das kartesische Gitter darstellt
`rank` - Nummer des Prozesses im Kommunikator c
`maxdim` - Anzahl der Dimensionen des kartesisches Gitters
`coords` - `int`-Feld, in das die Dimensionen eingetragen werden

Idealerweise sollte `maxdim` sowohl mit der Größe des eindimensionalen Feldes `coords` als auch mit der Anzahl der Dimensionen des kartesischen Gitters `comm` übereinstimmen. Nach der Ausführung dieser Funktion steht in `coords[i]` die Koordinate vom Prozeß `rank` in der Dimension *i*.

`int` **`MPI_Cart_shift`**`(MPI_Comm comm, int dim, int disp, int *source, int *dest);`
Ermittelt die Nummern (ranks) der Nachbarn eines Prozesses in einem kartesischen Gitter.
Parameter:
`comm` - Kommunikator, der auf das Gitter zeigt
`dim` - Dimension, in der die Nachbarn ermittelt werden sollen
`disp` - Abstand, in dem die Nachbarn ermittelt werden sollen
`source` - Zeiger auf eine `int`-Variable, in die die Nummer des Prozesses eingetragen wird, der in die entgegengesetzte Richtung zu `dir` im Abstand `disp` liegt
`dest` - Zeiger auf eine `int`-Variable, in die die Nummer des Prozesses eingetragen wird, der in Richtung `dir` im Abstand `disp` liegt

Der Parameter `dim` gibt die Dimension an, in der die Nachbarn ermittelt werden sollen. Die erste Dimension ist 0, die zweite 1 usw. In jeder Dimension gibt es zwei Nachbarn im Abstand von `disp`. In die Variable, auf die `source` verweist, wird die Nummer des Nachbarn in der "linken"/"oberen" Richtung im Abstand `disp` eingetragen; in die Variable, auf die `dest` verweist, wird die Nummer des Nachbarn in der "rechten"/"unteren"Richtung im Abstand `disp` eingetragen. Eine -1 als `source` bzw. `dest` bedeutet, daß der Knoten in der angegebenen Distanz `disp` keinen linken bzw. rechten Nachbarn hat (Beispielsweise am Rand eines Gitters).

`int` **`MPI_Cart_sub`**`(MPI_Comm cOld, int *dims, MPI_Comm *cNew);`
Erstellt aus einem kartesisches Gitter weitere Gitter mit jeweils weniger Dimensionen.
Parameter:
`cOld` - Kommunikator, der auf das alte Gitter zeigt
`dims` - `int`-Feld, das die Dimensionen der alten Gitters enthält, die in das neue Gitter übernommen werden sollen
`cNew` - Zeiger auf den Kommunikator, welcher nach der Ausführung der Funktion auf das neue Gitter zeigt

`int` **`MPI_Graph_create`**`(MPI_Comm cOld, int nnodes, int *index, int *edges, int reorder, MPI_Comm *cNew);`
Erzeugt eine neue Graph-Topologie.
Parameter:
`cOld` - der Kommunikator, aus dem der Graph erzeugt werden soll
`nnodes` - Anzahl der Knoten, die der Graph enthält
`index` - `int`-Feld, das die Durchnummerierung der Knoten angibt
`edges` - `int`-Feld, das die Kanten des Graphen spezifiziert
`reorder` - `int`-Variable, die angibt, ob MPI die Prozesse beim Anlegen der Topologie umordnen darf (1) oder nicht (0)
`cNew` - Zeiger auf einen Kommunikator, der auf den Graphen zeigt

`int` **`MPI_Graph_get`**`(MPI_Comm comm, int maxindex, int maxedges, int *index, int *edges);`
Ermittelt Daten eines Graphen.
Parameter:
`comm` - Kommunikator, der auf den Graphen zeigt
`maxindex` - Größe des Feldes index
`maxedges` - Größe des Feldes edges
`index` - `int`-Feld, in das die Indices der Knoten eingetragen werden
`edges` - `int`-Feld, in das die Kantenbeziehungen im Graphen `comm` eingetragen werden

`int` **`MPI_Graphdims_get`**`(MPI_Comm comm, int *nnodes, int *nedges);`
Ermittelt die Anzahl der Knoten und Kanten eines Graphen.
Parameter:
`comm` - Kommunikator, der auf den Graphen zeigt
`nnodes` - Zeiger auf eine `int`-Variable, in die die Anzahl der Knoten von `comm` eingetragen wird
`nedges` - Zeiger auf eine `int`-Variable, in die die Anzahl der Kanten von `comm` eingetragen wird

`int` **`MPI_Graph_map`**`(MPI_Comm comm, int nnodes, int *index, int *edges, int *newrank);`
Projiziert den aufrufenden Prozeß auf eine Graph-Topologie.
Parameter:
`comm` - Kommunikator, der den Graphen darstellt
`nnodes` - Anzahl der Knoten von `comm`
`index` - `int`-Feld, das die Knotennummerierung angibt
`edges` - `int`-Feld, das die Kantenbeziehungen angibt
`newrank` - Zeiger auf eine `int`-Variable, in die die neue Nummer (rank) des aufrufenden Prozesses eingetragen wird

`int` **`MPI_Graph_neighbors`**`(MPI_Comm comm, int rank, int maxneighbors, int *neighbors);`
Ermittelt die Nachbarn eines Knotens in einem Graphen.
Parameter:
`comm` - Kommunikator, der auf den Graphen zeigt
`rank` - Nummer des Knotens (nicht der Index!) des Knotens im Graphen `comm`
`maxneighbors` - Größe des Feldes `neighbors`
`neighbors` - Feld, in das die Nummern (ranks) der Nachbarn des Knotens `rank` eingetragen werden

`int` **`MPI_Graph_neighbors_count`**`(MPI_Comm comm, int rank, int *nneighbors);`
Ermittelt die Anzahl der Nachbarn eines Knotens in einem Graphen.
Parameter:
`comm` - Kommunikator, der auf den Graphen zeigt
`rank` - Nummer des Knotens (nicht der Index!) des Knotens im Graphen `comm`
`nneighbors` - Zeiger auf eine `int`-Variable, in die die Anzahl der Nachbarn von `rank` eingetragen werden

`int` **`MPI_Dims_create`**`(int n, int dim, int *dims);`
Diese Hilfsfunktion versucht, eine gewisse Anzahl (von Prozessen) so in einem kartesischen Gitter, daß das Gitter in jeder Dimension möglichst die gleiche Ausdehnung hat (also möglichst quadratisch, kubisch, ... ist).
Parameter:

`n` - Anzahl (der Prozesse)
`dim` - Anzahl der Dimensionen
`dims` - `int`-Feld, in das MPI einträgt, wie groß die einzelnen Dimensionen sind

Die erste Dimension beginnt beim Index 0, d.h. `dims[0]` enthält die Größe der ersten Dimension. Es ist zu beachten, daß diese Funktion kein kartesisches Gitter erzeugt.

`int` **`MPI_Topo_test`**`(MPI_Comm comm, int *result);`
Ermittelt den Typ einer Topologie.
Parameter:
`comm` - Kommunikator, der getestet werden soll
`result` - Zeiger auf eine `int`-Variable, in die das Ergebnis eingetragen wird

Das Ergebnis ist einer der Werte `MPI_UNDEFINED`, `MPI_GRAPH`, und `MPI_CART`.

`unsigned int` **`MPI_IDENT;`**
Definiert die Konstante, die angibt, daß zwei verglichende Gruppen oder Kommunikatoren identisch sind. Sie umfassen den gleichen Kontext (die gleiche Anzahl von Prozessoren in der gleichen Anordnung) und dieselbe Gruppe.

`unsigned int` **`MPI_CONGRUENT;`**
Definiert die Konstante, die angibt, daß zwei verglichende Kommunikatoren kongruent sind, also die gleiche Gruppe haben. Dieser Rückgabewert existiert nur für `MPI_Comm_compare`.

`unsigned int` **`MPI_SIMILAR;`**
Definiert die Konstante, die angibt, daß zwei verglichende Gruppen oder Kommunikatoren ähnlich sind. Sie haben die gleiche Anzahl von Prozessen, jedoch nicht die gleiche Anordnung.

`unsigned int` **`MPI_UNEQUAL;`**
Definiert die Konstante, die angibt, daß zwei verglichende Gruppen oder Kommunikatoren nicht gleich sind. Sie haben weder die gleiche Anordnung von Prozessen noch die gleiche Anzahl von Prozessen.

`unsigned int` **`MPI_GRAPH;`**
Definiert die Konstante, die angibt, daß eine mit `MPI_Topo_test` getestete Topologie ein Graph ist.

`unsigned int` **`MPI_CART;`**
Definiert die Konstante, die angibt, daß eine mit `MPI_Topo_test` getestete Topologie ein kartesisches Gitter ist.

Referenz - Typen

Hinweis: Einige hier aufgeführten Definitionen sind nicht von allen MPI-Implementationen übernommen worden. Außerdem sind einige Definitionen entweder nur für C oder nur für Fortran77 sinnvoll.

Desweiteren sind einige hier ausgeführten Funktionen Wrapper-Funktionen für andere Funktionen, die hier nicht aufgelistet sind.

`int` **`MPI_Type_contiguous`**`(int count, MPI_Datatype tOld, MPI_Datatype *tNew);`

Erstellt einen neuen Datentyp, der einer Reihung entspricht.

Parameter:

`count` - Anzahl der Elemente der Reihung

`tOld` - Datentyp der Elemente, aus denen die Reihung bestehen soll

`tNew` - Zeiger auf den neuen Datentypen

`int` **`MPI_Type_hindexed`**`(int count, int *lengths, MPI_Aint *displs, MPI_Datatype tOld, MPI_Datatype *tNew);`

Erstellt eine Reihung als neuen Datentyp.

Parameter:

`count` - Anzahl der Elemente der Reihung

`lengths` - `int`-Feld, das die L&aum;nge der einzelnen Elemente angibt

`displs` - `MPI_Aint`-Feld, dessen Einträge die Positionen der Elemente der Reihung angeben

`tOld` - Datentyp der Elemente der Reihung

`tNew` - Zeiger auf den neuen Typ, der erzeugt wird

`int` **`MPI_Type_indexed`**`(int count, int *lengths, int *displs, MPI_Datatype tOld, MPI_Datatype *tNew);`

Erstellt eine Reihung als neuen Datentyp.

Parameter:

`count` - Anzahl der Elemente der Reihung

`lengths` - `int`-Feld, das die L&aum;nge der einzelnen Elemente angibt

`displs` - `MPI_Aint`-Feld, dessen Einträge die Positionen der Elemente der Reihung angeben

`tOld` - Datentyp der Elemente der Reihung

`tNew` - Zeiger auf den neuen Typ, der erzeugt wird

`int` **`MPI_Type_hvector`**`(int count, int length, MPI_Aint stride, MPI_Datatype tOld, MPI_Datatype *tNew);`

Erstellt einen Vektor als neuen Datentyp.

Parameter:

`count` - Anzahl der Einträge des Vektors

`length` - Länge eines Eintrags des Vektors

`stride` - Abstand vom Anfang einer Komponente des Vektors zum Anfang der nächsten Komponente

`tOld` - Typ der Einträge des Vektors

`tNew` - Zeiger auf den neuen Typ, der erzeugt wird

`int` **`MPI_Type_vector`**`(int count, int length, int stride, MPI_Datatype tOld, MPI_Datatype *tNew);`

Erstellt einen Vektor als neuen Datentyp.

Parameter:

`count` - Anzahl der Einträge des Vektors

`length` - Länge eines Eintrags des Vektors

`stride` - Abstand vom Anfang einer Komponente des Vektors zum Anfang der nächsten Komponente

`tOld` - Typ der Einträge des Vektors

`tNew` - Zeiger auf den neuen Typ, der erzeugt wird

`int` **`MPI_Type_struct`**`(int count, int *lengths, MPI_Aint *displs, MPI_Datatype *types, MPI_Datatype *tNew);`
Erstellt eine Struktur als neuen Datentypen.
Parameter:
`count` - Anzahl der Elemente der Struktur
`lengths` - `int`-Feld, das die Längen der Elemente der Struktur angibt
`displs` - `MPI_Aint`-Feld, dessen Einträge die Positionen der Elemente der Struktur angeben
`types` - Feld, das die einzelnen Typen der Struktur enthält
`tNew` - Zeiger auf den neuen Typ, der erzeugt wird

`int` **`MPI_Type_commit`**`(MPI_Datatype *type);`
Macht MPI einen neuen Datentyp bekannt.
Parameter:
`type` - der Datentyp

Ein Datentyp kann mit MPI erst dann verwendet werden, wenn dieser MPI mit `MPI_commit` bekannt gemacht wurde.

`int` **`MPI_Type_free`**`(MPI_Datatype *type);`
Zerstört einen Datentypen und gibt die von ihm verwendeten Ressourcen frei.
Parameter:
`type` - der Datentyp

Mit dieser Funktion können nur benutzerdefinierte Typen zerstört werden.

`int` **`MPI_Get_count`**`(MPI_Status *status, MPI_Datatype type, int *count);`
Ermittelt die Anzahl der Top-Level-Elemente einer Datenstruktur.
Parameter:
`status` - Zeiger auf eine `MPI_Status`-Variable, die von der Empfangsoperation gesetzt wurde, mit der die Datenstruktur empfangen wurde
`type` - der Datentyp, den die Datenstruktur hat
`count` - Zeiger auf eine `int`-Variable, in die Anzahl eingetragen wird

Die Top-Level-Elemente einer Datenstuktur sind die Elemente, die **unmittelbar** a ihrem Aufbau beteiligt sind.

`int` **`MPI_Get_elements`**`(MPI_Status *status, MPI_Datatype type, int *count);`
Ermittelt die Anzahl der Elemente einer Datenstruktur.
Parameter:
`status` - Zeiger auf eine `MPI_Status`-Variable, die von der Empfangsoperation gesetzt wurde, mit der die Datenstruktur empfangen wurde
`type` - der Datentyp, den die Datenstruktur hat
`count` - Zeiger auf eine `int`-Variable, in die Anzahl eingetragen wird

Im Gegensatz zur Funktion `MPI_Get_count` ermittelt diese Funktion die Anzahl aller am Aufbau eines Datentyps beteiligten Elemente.

`int` **`MPI_Type_extent`**`(MPI_Datatype type, MPI_Aint *extent);`
Liefert den Umfang eines Datentyps. Der Umfang eines Datentyps ist definiert durch: Obergrenze minus Untergrenze, aufgerundet auf das nächste Vielfache der Wortbreite der Maschine.

Parameter:
`type` - der Datentyp
`extent` - Zeiger auf eine Variable vom Typ `MPI_Aint`, in die der Umfang des Typs type eingetragen wird

`int` **`MPI_Type_size`**`(MPI_Datatype type, int *size);`
Ermittelt die Größe eines Datentyps. Die Größ eines Datentyps ist die Anzahl der von ihm eingebommenen Bytes. Das Äquivalent in C ist der `sizeof`-Operator.
Parameter:
`type` - der Datentyp
`size` - Zeiger auf eine `int`-Variable, in die die Größe des Typs type eingetragen wird

`int` **`MPI_Type_ub`**`(MPI_Datatype type, MPI_Aint *ub);`
Ermittelt die Obergrenze eines Datentyps. Die Obergrenze eines Datentyps ist die Position der letzten Komponente plus dieGröße dieser Komponente in Bytes.
Parameter:
`type` - der Datentyp
`ub` - Zeiger auf eine Variable vom Typ `MPI_Aint`, in die die Obergrenze eingetragen wird

`int` **`MPI_Type_lb`**`(MPI_Datatype type, MPI_Aint *lb);`
Ermittelt die Untergrenze eines Datentyps. Die Untergrenze eines Datentyps ist die Position der ersten Komponente.
Parameter:
`type` - der Datentyp
`lb` - Zeiger auf eine Variable vom Typ `MPI_Aint`, in die die Untergrenze eingetragen wird

`int` **`MPI_Address`**`(void *v, MPI_Aint *address);`
Ermittelt zu einer Adresse die zugehörige Adresse im Typ `MPI_Aint`.
Parameter:
`v` - Zeiger auf den Speicherbereich, dessen Adresse ermittelt werden soll
`address` - eine Variable vom Typ `MPI_Aint`, in die die Adresse eingetragen wird

Einige Funktionen, die einen benutzerdefinierten Datentyp erstellen, benötigen Adressen im Typ `MPI_Aint`. Mit dieser Funktion kann eine normale Adresse in eine solche umgewandelt werden.

`#define` **`MPI_BOTTOM`** `((MPI_Aint)0)`
Definiert das untere Ende eines benutzerdefinierten Datentyps als Adresse 0 relativ zum Beginn des Datentyps.

`unsigned int` **`MPI_DATATYPE_NULL;`**
Definiert die Konstante, die den Datentyp angibt, der eigentlich keiner ist.

`unsigned int` **`MPI_CHAR;`**
Definiert die Konstante, die den Datentyp `char` angibt (nur für C).

`unsigned int` **`MPI_BYTE;`**
Definiert die Konstante, die den Datentyp `unsigned char` angibt (nur für C).

`unsigned int` **`MPI_SHORT;`**
Definiert die Konstante, die den Datentyp `short` angibt (nur für C).

`unsigned int` **`MPI_INT;`**
Definiert die Konstante, die den Datentyp `int` angibt (nur für C).

`unsigned int` **`MPI_LONG;`**
Definiert die Konstante, die den Datentyp `long` angibt (nur für C).

`unsigned int` **`MPI_UNSIGNED_CHAR;`**
Definiert die Konstante, die den Datentyp `unsigned char` angibt (nur für C).

`unsigned int` **`MPI_UNSIGNED_SHORT;`**
Definiert die Konstante, die den Datentyp `unsigned short` angibt (nur für C).

`unsigned int` **`MPI_UNSIGNED;`**
Definiert die Konstante, die den Datentyp `unsigned int` angibt (nur für C).

`unsigned int` **`MPI_UNSIGNED_LONG;`**
Definiert die Konstante, die den Datentyp `unsigned long` angibt (nur für C).

`unsigned int` **`MPI_FLOAT;`**
Definiert die Konstante, die den Datentyp `float` angibt (nur für C).

`unsigned int` **`MPI_DOUBLE;`**
Definiert die Konstante, die den Datentyp `double` angibt (nur für C).

`unsigned int` **`MPI_LONG_DOUBLE;`**
Definiert die Konstante, die den Datentyp `long double` angibt (nur für C).

`unsigned int` **`MPI_LONG_LONG;`**
Definiert die Konstante, die den Datentyp `long long` angibt (nur für C).

`unsigned int` **`MPI_FLOAT_INT;`**
Definiert die Konstante, die den Datentyp `struct {float; int}` angibt (nur für C). Dieser Datentyp wird benutzt für die Redizierungsoperationen `MPI_MINLOC` und `MPI_MAXLOC`. In der ersten Komponente steht das Minimum/Maximum, in der zweiten Komponente steht die Nummer des Prozesses, der das Minimum/Maximum beisteuert.

`unsigned int` **`MPI_DOUBLE_INT;`**
Definiert die Konstante, die den Datentyp `struct {double; int}` angibt (nur für C). Dieser Datentyp wird benutzt für die Redizierungsoperationen `MPI_MINLOC` und `MPI_MAXLOC`. In der ersten Komponente steht das Minimum/Maximum, in der zweiten Komponente steht die Nummer des Prozesses, der das Minimum/Maximum beisteuert.

`unsigned int` **`MPI_LONG_INT;`**
Definiert die Konstante, die den Datentyp `struct {long; int}` angibt (nur für C). Dieser Datentyp wird benutzt für die Redizierungsoperationen `MPI_MINLOC` und `MPI_MAXLOC`. In der ersten Komponente steht das Minimum/Maximum, in der zweiten Komponente steht die Nummer des Prozesses, der das Minimum/Maximum beisteuert.

`unsigned int` **`MPI_2INT;`**
Definiert die Konstante, die den Datentyp `struct {int; int}` angibt (nur für C). Dieser Datentyp wird benutzt für die Redizierungsoperationen `MPI_MINLOC` und `MPI_MAXLOC`. In der ersten Komponente steht das Minimum/Maximum, in der zweiten Komponente steht die Nummer des Prozesses, der das Minimum/Maximum beisteuert.

`unsigned int` **`MPI_SHORT_INT;`**
Definiert die Konstante, die den Datentyp `struct {short; int}` angibt (nur für C). Dieser Datentyp wird benutzt für die Redizierungsoperationen `MPI_MINLOC` und `MPI_MAXLOC`. In der ersten Komponente steht das Minimum/Maximum, in der zweiten Komponente steht die Nummer des Prozesses, der das Minimum/Maximum beisteuert.

`unsigned int` **`MPI_LONG_DOUBLE_INT;`**
Definiert die Konstante, die den Datentyp `struct {long double; int}` angibt (nur für C). Dieser Datentyp wird benutzt für die Redizierungsoperationen `MPI_MINLOC` und `MPI_MAXLOC`. In der ersten Komponente steht das Minimum/Maximum, in der zweiten Komponente steht die Nummer des Prozesses, der das Minimum/Maximum beisteuert.

`#define` **`MPI_LONG_LONG_INT`** `MPI_LONG_LONG`
Definiert einen Alias für die Konstante `MPI_LONG_LONG`

`unsigned int` **`MPI_INTEGER;`**
Definiert die Konstante, die den Datentyp `INTEGER` angibt (nur für Fortran77).

`unsigned int` **`MPI_REAL;`**
Definiert die Konstante, die den Datentyp `REAL` angibt (nur für Fortran77).

`unsigned int` **`MPI_DOUBLE_PRECISION;`**
Definiert die Konstante, die den Datentyp `DOUBLE PRECISION` angibt (nur für Fortran77).

`unsigned int` **`MPI_COMPLEX;`**
Definiert die Konstante, die den Datentyp `COMPLEX` angibt (nur für Fortran77).

`unsigned int` **`MPI_DOUBLE_COMPLEX;`**
Definiert die Konstante, die den Datentyp `complex*16` oder `complex*32` angibt (nur für Fortran77).

`unsigned int` **`MPI_LOGICAL;`**
Definiert die Konstante, die den Datentyp `LOGICAL` angibt (nur für Fortran77).

`unsigned int` **`MPI_CHARACTER;`**
Definiert die Konstante, die den Datentyp `CHARACTER` angibt (nur für Fortran77).

`unsigned int` **`MPI_INTEGER1;`**
Definiert die Konstante, die den Datentyp `integer*1` angibt (nur für Fortran77).

`unsigned int` **`MPI_INTEGER2;`**
Definiert die Konstante, die den Datentyp `integer*2` angibt (nur für Fortran77).

`unsigned int` **`MPI_INTEGER4;`**
Definiert die Konstante, die den Datentyp `integer*4` angibt (nur für Fortran77).

`unsigned int` **`MPI_INTEGER8;`**
Definiert die Konstante, die den Datentyp `integer*8` angibt (nur für Fortran77).

`unsigned int` **`MPI_REAL4;`**
Definiert die Konstante, die den Datentyp `real*4` angibt (nur für Fortran77).

`unsigned int` **`MPI_REAL8;`**
Definiert die Konstante, die den Datentyp `real*8` angibt (nur für Fortran77).

`unsigned int` **`MPI_REAL16;`**
Definiert die Konstante, die den Datentyp `real*16` angibt (nur für Fortran77).

`unsigned int` **`MPI_2REAL;`**
Definiert die Konstante, die den Datentyp `REAL, REAL` angibt (nur für C). Dieser Datentyp wird benutzt für die Redizierungsoperationsen `MPI_MINLOC` und `MPI_MAXLOC`. In der ersten Komponente steht das Minimum/Maximum, in der zweiten Komponente steht die Nummer des Prozesses, der das Minimum/Maximum beisteuert.

`unsigned int` **`MPI_2DOUBLE_PRECISION;`**
Definiert die Konstante, die den Datentyp `DOUBLE PRECISION, DOUBLE PRECISION` angibt (nur für Fortran77). Dieser Datentyp wird benutzt für die Redizierungsoperationsen `MPI_MINLOC` und `MPI_MAXLOC`. In der ersten Komponente steht das Minimum/Maximum, in der zweiten Komponente steht die Nummer des Prozesses, der das Minimum/Maximum beisteuert.

`unsigned int` **`MPI_2INTEGER;`**
Definiert die Konstante, die den Datentyp `INTEGER, INTEGER` angibt (nur für Fortran77). Dieser Datentyp wird benutzt für die Redizierungsoperationsen `MPI_MINLOC` und `MPI_MAXLOC`. In der ersten Komponente steht das Minimum/Maximum, in der zweiten Komponente steht die Nummer des Prozesses, der das Minimum/Maximum beisteuert.

`unsigned int` **`MPI_PACKED;`**
Definiert die Konstante, die den Datentyp MPI_PACKED angibt. Dieser Typ ist nur MPI bekannt. Variablen dieses Typs werden mit `MPI_Pack`
erzeugt.

`unsigned int` **`MPI_UB;`**
Definiert die Konstante, die beim Aufruf von Funktionen, die Typen betreffen, das letzte Element eines Typs angeben.

`unsigned int` **`MPI_LB;`**
Definiert die Konstante, die beim Aufruf von Funktionen, die Typen betreffen, das erste Element eines Typs angeben.

Referenz - Kommunikatoren und Gruppen

Hinweis: Einige hier aufgeführten Definitionen sind nicht von allen MPI-Implementationen übernommen worden. Außerdem sind einige Definitionen entweder nur für C oder nur für Fortran77 sinnvoll.
Desweiteren sind einige hier ausgeführten Funktionen Wrapper-Funktionen für andere Funktionen, die hier nicht aufgelistet sind.
`int` **`MPI_Comm_create`**`(MPI_Comm cOld, MPI_Group group, MPI_Comm *cNew);`
Erzeugt aus einem Kommunikator einen neuen Kommunikator.
Parameter:

`cOld` - Kommunikator, aus dem der neue Kommunikator erzeugt werden soll
`group` - Gruppe, aus die der neue Kommunikator erzeugt werden soll
`cNew` - Zeiger auf einen Kommunikator, der nach dem Ausführen dieser Funktion auf den neuen Kommunikator zeigt

Der neue Kommunikator wird aus den Prozessen erzeugt, die im Kommunikator `cOld` sind. Die Gruppe `group` gibt dann an, welche Prozesse in den neuen Kommunikator `nNew` kommen und welche nicht. Ein Prozeß kommt genau dann in `nNew`, wenn er in `group` ist.

`int` **`MPI_Comm_free`**`(MPI_Comm *comm);`
Löscht einen Kommunikator.
Parameter:
`comm` - Zeiger auf den Kommunikator, der gelöscht werden soll

Mit dieser Funktion können nur benutzerdefinierte Kommunikatoren zerstört werden.

`int` **`MPI_Comm_compare`**`(MPI_Comm comm1, MPI_Comm comm2, int *result);`
Vergleicht zwei Kommunikatoren.
Parameter:
`comm1` - der eine Kommunikator
`comm2` - der andere Kommunikator
`result` - Zeiger auf eine `int`-Variable, in die das Ergebnis des Vergleichs eingetragen werden soll

Das Resultat ist einer Konstanten `MPI_IDENT`, `MPI_CONGRUENT`, `MPI_SIMILAR` und `MPI_UNEQUAL`.

`int` **`MPI_Comm_dup`**`(MPI_Comm cOld, MPI_Comm *cNew);`
Dupliziert einen Kommunikator.
Parameter:
`cOld` - der alte Kommunikator
`cNew` - Zeiger auf den neuen Kommunikator

`int` **`MPI_Comm_split`**`(MPI_Comm cOld, int color, int key, MPI_Comm *nNew);`
Spaltet einen Kommunikator auf.
Parameter:
`cOld` - der alte Kommunikator
`color` - eine Farbe, die dem aufrufenden Prozeß zugewiesen wird
`key` - die neue Nummer (rank), die dem aufrufenden Prozeß im neuen Kommunikator zugewiesen wird
`nNew` - Zeiger auf den neuen Kommunikator

Der alte Kommunikator `cOld` wird so aufgespalten, daß alle die Prozesse, die die gleiche Farbe haben, zu einem Kommunikator zusammengefaßt werden und dort die Nummer `key` erhalten. `cNew` repräsentiert nach dem Aufruf dieser Funktion den Kommunikator, dem der aufrufende Prozeß zugeordnet wurde aufgrund seiner Farbe. Diese Funktionen müssen alle Prozesse im Kommunikator aufrufen.

`int` **`MPI_Comm_rank`**`(MPI_Comm comm, int *rank);`
Ermittelt die Nummer (rank) des aufrufenden Prozesses im Kommunikator.
Parameter:
`comm` - der Kommunikator, in dem die Nummer ermittelt werden soll

`rank` - Zeiger auf eine `int`-Variable, in die die Nummer des aufrufenden Prozesses im Kommunikator `comm` eingetragen wird

Ein Prozeß kann durchaus zu mehreren Kommunikatoren gehöhren. Deswegen muß der Kommunikator angegeben werden. Der erste Prozeß hat die Nummer 0, der zweite die Nummer 1 usw.

`int` **`MPI_Comm_size`**`(MPI_Comm c, int *size);`
Liefert die Anzahl der Prozesse eines Kommunikators.
Parameter:
`comm` - Kommunikator, dessen Gröszlig;e ermittelt werden soll
`size` - `int`-Variable, in die die Anzahl eingetragen wird

`int` **`MPI_Intercomm_create`**`(MPI_Comm cLocal, int masterLocal, MPI_Comm cPeer, int masterPeer, int tag, MPI_Comm *cNew);`
Erzeugt einen neuen Interkommunikator.
Parameter:
`cLocal` - lokaler Kommunikator, der zum Interkommunikator verbunden werden soll
`masterLocal` - Nummer des Prozesses im Kommunikator `cLocal`, der im Interkommunikator mit rank=0 auftreten soll
`cPeer` - Kommunikator, der mit cLocal zum Interkommunikator verbunden werden soll
`masterPeer` - Nummer des Prozesses im Kommunikator `cPeer`, der im Interkommunikator mit rank=0 auftreten soll
`tag` - Kennzeichnung des Interkommunikators
`cNew` - Zeiger auf den neuen Interkommunikator

Die beiden Kommunikatoren `cLocal` und `cPeer` bleiben erhalten, ebenso die Prozeßnummerierung. Die Semantik von Prozeßnummern in Interkommunikatoren hängt vom Kontext (Standort des Aufrufs) der verwendeten Operationen ab, siehe dazu Kapitel 7.4.

`int` **`MPI_Intercomm_merge`**`(MPI_Comm cInter, int high, MPI_Comm *cIntra);`
Wandelt einen Interkommunikator in einen normalen Kommunikator um.
Parameter:
`cInter` - der Interkommunikator
`high` - Wozu soll das gut sein?
`cIntra` - Zeiger auf einen Kommunikator, der nach dem Aufruf dieser Funktion auf den normalen Kommunikator zeigt

`int` **`MPI_Comm_test_inter`**`(MPI_Comm comm, int *inter);`
Testet, ob ein Kommunikator ein Interkommunikator ist.
Parameter:
`comm` - Interkommunikator, der getestet werden soll
`inter` - Zeiger auf eine `int`-Variable, die auf einen Wert ungleich 0 gesetzt wird, falls `comm` ein Interkommunikator ist, und andernfalls auf 0 gesetzt wird

`int` **`MPI_Comm_remote_size`**`(MPI_Comm comm, int *size);`
Liefert die Anzahl der Prozesse eines Interkommunikators.
Parameter:
`comm` - Interkommunikator, dessen Gröszlig;e ermittelt werden soll

`size` - Zeiger auf eine `int`-Variable, in die die Anzahl eingetragen wird

`int` **`MPI_Comm_group`**`(MPI_Comm ccomm, MPI_Group *group);`
Liefert die einem Kommunikator zugeordnete Gruppe.
Parameter:
`comm` - Kommunikator, dessen Gruppe ermittelt werden soll
`group` - Zeiger auf eine Gruppe, die nach der Ausführung der Funktion auf die `comm` zugeordnete Gruppe zeigt

`int` **`MPI_Comm_remote_group`**`(MPI_Comm comm, MPI_Group *group);`
Liefert die einem Interkommunikator zugeordnete Gruppe.
Parameter:
`comm` - Interkommunikator, dessen Gruppe ermittelt werden soll
`group` - Gruppe, die nach der Ausführung der Funktion auf die `comm` zugeordnete Gruppe zeigt

`int` **`MPI_Group_free`**`(MPI_Group *group);`
Zerstört eine Gruppe und gibt die von ihr belegten Ressourcen frei.
Parameter:
`group` - die Gruppe

Mit dieser Funktion können nur benutzerdefinierte Gruppen zerstört werden.

`int` **`MPI_Group_excl`**`(MPI_Group gOld, int nranks, int *ranks, MPI_Group *gNew);`
Erzeugt eine neue Gruppe, indem Prozesse aus einer Gruppe entfernt werden.
Parameter:
`gOld` - Gruppe, deren Prozesse behandelt werden sollen
`nranks` - Gröszlig;e des Feldes `ranks`
`ranks` - `int`-Feld, das die Prozessnummern angibt, die aus der Gruppe `gOld` entfernt werden sollen
`gNew` - Zeiger auf die neue Gruppe

`int` **`MPI_Group_incl`**`(MPI_Group gOld, int nranks, int *ranks, MPI_Group *gNew);`
Erzeugt eine neue Gruppe, indem Prozesse zu einer einer Gruppe hinzugefügt werden.
Parameter:
`gOld` - Gruppe, deren Prozesse behandelt werden sollen
`nranks` - Gröszlig;e des Feldes `ranks`
`ranks` - `int`-Feld, das die Prozeßnummern angibt, die zu der Gruppe `gOld` hinzugefügt werden sollen
`gNew` - Zeiger auf die neue Gruppe

Die Prozesse, die zu `gOld` hinzugefügt werden sollen, müssen im Kommunikator, aus dem `gOld` erzeugt worden ist, enthalten sein.

`int` **`MPI_Group_range_excl`**`(MPI_Group gOld, int nranges, int ranges[][3], MPI_Group *gNew);`
Erzeugt eine neue Gruppe, indem aus einer bestehenden Gruppe Bereiche von Prozessen entfernt werden.
Parameter:
`gOld` - Gruppe, aus der die neue Gruppe erzeugt wird
`nranges` - Größe des Feldes `ranges[]`
`ranges[][3]` - `int`-Feld von Tripeln, in denen die erste Nummer, die letzte Nummer und ein Abstand stehen

gNew - Zeiger auf die neue Gruppe

Kommt zum Beispiel das Tripel {4, 10, 3} im Feld ranges vor, werden die Prozesse 4, 7 und 10 des Kommunikators, zu dem der aufrufende Prozeß gehört, entfernt.

```
int MPI_Group_range_incl(MPI_Group gOld, int nranges, int ranges[][3],
MPI_Group *gNew);
```

Erzeugt eine neue Gruppe, indem zu einer bestehenden Gruppe neue Bereiche von Prozessen hinzugefügt werden.
Parameter:
gOld - Gruppe, aus der die neue Gruppe erzeugt wird
nranges - Größe des Feldes ranges[]
ranges[][3] - int-Feld von Tripeln, in denen die erste Nummer, die letzte Nummer und ein Abstand stehen
gNew - Zeiger auf die neue Gruppe

Kommt zum Beispiel das Tripel {5, 12, 2} im Feld ranges vor, werden die Prozesse 5, 7, 9 und 11 des Kommunikators, zu dem der aufrufende Prozeß gehört, hinzugefügt.

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group
*groupRes);
```

Erzeugt eine neue Gruppe, deren Menge der Prozesse die Differenzmenge der Mengen von Prozessen zweier Gruppen ist.
Parameter:
group1 - die eine Gruppe
group2 - die andere Gruppe
groupRes - Zeiger auf die neue Gruppe

Die Gruppe groupRes enthält alle Prozesse, die in Gruppe group1, aber nicht in Gruppe group2 enthalten sind.

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group
*groupRes);
```

Erzeugt eine neue Gruppe, deren Menge der Prozesse die Schnittmenge der Mengen von Prozessen zweier Gruppen ist.
Parameter:
group1 - die eine Gruppe
group2 - die andere Gruppe
groupRes - Zeiger auf die neue Gruppe

Die Gruppe groupRes enthält alle Prozesse, die in Gruppe group1 und in Gruppe group2 enthalten sind.

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *groupRes);
```

Erzeugt eine neue Gruppe, deren Menge der Prozesse die Vereinigungsmenge der Mengen von Prozessen zweier Gruppen ist.
Parameter:
group1 - die eine Gruppe
group2 - die andere Gruppe
groupRes - Zeiger auf die neue Gruppe

Die Gruppe groupRes enthält alle Prozesse, die in Gruppe group1 oder in Gruppe group2 oder in beiden Gruppen enthalten sind.

`int` **`MPI_Group_compare`**`(MPI_Group group1, MPI_Group group2, int *result);`
Vergleicht zwei Gruppen.
Parameter:
`group1` - die eine Gruppe
`group2` - die andere Gruppe
`result` - Zeiger auf eine `int`-Variable, in die das Ergebnis eingetragen wird

Das Resultat ist einer Konstanten `MPI_IDENT`, `MPI_SIMILAR` und `MPI_UNEQUAL`.

`int` **`MPI_Group_rank`**`(MPI_Group group, int *rank);`
Ermittelt die Nummer (rank) eines Prozesses in einer Gruppe.
Parameter:
`group` - die Gruppe
`rank` - Zeiger auf eine `int`-Variable, in die die Nummer eingetragen wird

`int` **`MPI_Group_size`**`(MPI_Group group, int *size);`
Ermittelt die Anzahl der Prozesse in einer Gruppe.
Parameter:
`group` - die Gruppe
`size` - Zeiger auf eine `int`-Variable, in die die Anzahl der Prozesse von `group` eingetragen werden

`int` **`MPI_Group_translate_ranks`**`(MPI_Group g1, int nranks, int *ranks1, MPI_Group g2, int *ranks2);`
Projiziert die Nummern (ranks) von Prozessen einer Gruppe auf eine andere Gruppe.
Parameter:
`g1` - die Gruppe, deren Prozeßnummern projiziert werden sollen
`nranks` - Größe der Felder `ranks1` und `ranks2`
`ranks1` - `int`-Feld, in dem die Prozeßnummern, gültig innerhalb von `g1`, stehen
`g2` - die Gruppe, auf die projiziert werden soll
`ranks2` - `int`-Feld, in welches die Prozeßnummern eingetragen werden, die die Prozesse, die in `ranks1` vorkommen, erhalten würden, wenn sie in `g2` vorkommen würden

`unsigned int` **`MPI_COMM_NULL;`**
Definiert die Konstante, die den Kommunikator angibt, der eigentlich kein Kommunikator ist. Prozesse, die beim Erzeugen von Kommunikatoren nicht in einen Kommunikator eingefügt werden, aber an dieser Operation teilnehmen, erhalten diesen Wert statt einen Verweis auf den neuen Kommunikator.

`unsigned int` **`MPI_COMM_WORLD;`**
Definiert die Konstante, die den Kommunikator angibt, der alle Prozesse beinhaltet.

`unsigned int` **`MPI_COMM_SELF;`**
Definiert die Konstante, die den Kommunikator angibt, der nur den Prozeß enthält, der eine Funktion aufruft.

`unsigned int` **`MPI_GROUP_NULL;`**

Definiert die Konstante, die die Gruppe angibt, die eigentlich keine Gruppe ist.

`unsigned int` **`MPI_GROUP_EMPTY;`**
Definiert die Konstante, die die Gruppe angibt, die keinen Prozeß enth&aumml;lt.

Referenz - Fehlerbehandlung

Hinweis: Einige hier aufgeführten Definitionen sind nicht von allen MPI-Implementationen übernommen worden. Außerdem sind einige Definitionen entweder nur für C oder nur für Fortran77 sinnvoll.
Desweiteren sind einige hier ausgeführten Funktionen Wrapper-Funktionen für andere Funktionen, die hier nicht aufgelistet sind.
`int` **`MPI_Errhandler_create`**`(MPI_Handler_function *hf, MPI_Errhandler *eh);`
Erzeugt aus einer Funktion einen Error-Handler.
Parameter:
`hf` - Zeiger auf die Funktion, aus der ein Error-Handler erzeugt werden soll
`eh` - Zeiger auf den Error-Handler, der erzeugt wird

Die Funktion, auf die `hf` zeigt, muß eine Instanz des Prototypen `MPI_Handler_function` sein.

`int` **`MPI_Errhandler_free`**`(MPI_Errhandler *eh);`
Zerstört einen Error-Handler.
Parameter:
`eh` - Zeiger auf den Error-Handler, der zerstört werden soll.

Wird ein Error-Handler zerstört, wird in allen Kommunikatoren, in denen dieser Error-Handler benutzt wird, der Standard-Error-Handler wieder eingesetzt. Mit dieser Funktion können nur benutzerdefinierte Error-Handler zerstört werden.

`int` **`MPI_Errhandler_set`**`(MPI_Comm comm, MPI_Errhandler eh);`
Setzt den Error-Handler für einen Kommunikator.
Parameter:
`comm` - Kommunikator, dessen Error-Handler gesetzt werden soll
`eh` - Error-Handler, der für den Kommunikator `comm` gesetzt werden soll

`int` **`MPI_Errhandler_get`**`(MPI_Comm comm, MPI_Errhandler *eh);`
Ermittelt den Error-Handler eines Kommunikators.
Parameter:
`comm` - Kommunikator, dessen Error-Handler ermittelt werden soll
`eh` - Zeiger auf den Error-Handler, der ermittelt wird

`int` **`MPI_Error_class`**`(int errcode, int *errclass);`
Ermittelt zu einem Fehlerwert die dazugehöhrige Fehlerklasse.
Parameter:
`errcode` - der Fehlerwert
`errclass` - Zeiger auf eine `int`-Variable, in die die Fehlerklasse eingetragen wird

Eine Aufzählung aller Fehlerklassen findet sich hier.

`int` **`MPI_Error_string`**`(int errcode, char *errstring, int *resLength);`
Ermittelt zu einem Fehlerwert die dazugehöhrige Fehlermeldung im Klartext.
Parameter:
`errcode` - der Fehlerwert
`errclass` - Zeiger auf eine `char`-Variable, in die die Fehlermeldung eingetragen wird
`resLength` - Zeiger auf eine `int`-Variable, in die die Länge der Fehlermeldung eingetragen wird

Die Fehlermeldung variiert von Implementierung zu Implementierung. Der Klartext findet sich nach dem Aufruf dieser Funktion im Feld `errstring[0]` bis `errstring[resLength-1]`.

`unsigned int` **`MPI_MAX_ERROR_STRING;`**
Definiert die maximale Länge des `char`-Feldes, das von `MPI_Error_string` zurückgegeben werden kann.

`typedef void` **`MPI_Handler_function`**`(MPI_Comm *comm, int *errcode, ...);`
Definiert einen Callback-Funktionsprototypen, dessen Instanzen (Funktionen von diesem Typ) mit `MPI_Errhandler_create` in einen Error-Handler umgewandelt werden k&oumm;nnen.
Parameter:
`comm` - Zeiger auf einen Kommunikator, der auf den Kommunikator zeigt, in dem der Fehler aufgetreten ist
`errcode` - Zeiger auf eine `int`-Variable, in die der Fehlerwert eingetragen wird

Weitere Argumente sind von der Implementation anhängig.

`unsigned int` **`MPI_ERRHANDLER_NULL;`**
Definiert die Konstante, die den Error-Handler angibt, der keine Fehler bearbeitet.

`unsigned int` **`MPI_ERRORS_ARE_FATAL;`**
Definiert die Konstante, die den Error-Handler angibt, der alle Fehler mit einem Abbruch des Programms im Kontext des jeweiligen Kommunikators moniert.

`unsigned int` **`MPI_ERRORS_RETURN;`**
Definiert die Konstante, die den Error-Handler angibt, der es zul&aum;ßt, daß die R&ouuml;ckgabewerte der Funktionen als Fehlerwerte verarbeitet werden können.

`unsigned int` **`MPI_SUCCESS;`**
Definiert den Fehlerwert, der den Erfolg einer Funktion anzeigt.

`unsigned int` **`MPI_ERR_BUFFER;`**
Definiert den Fehlerwert, der einen Fehler bei Puffern anzeigt. Möglicherweise wurde versucht, einen größeren Puffer anzusprechen, als ursprünglich vereinbart wurde. Bei einigen Funktionen müssen die Argumente bezüglich Puffer in allen beteiligten Prozessen übereinstimmen.

`unsigned int` **`MPI_ERR_COUNT;`**
Definiert den Fehlerwert, der einen Fehler bei der Anzahl von Pufferelementen anzeigt. Bei einigen Funktionen müssen die Argumente bezüglich Anzahlen von Elementen in allen beteiligten Prozessen übereinstimmen.

`unsigned int` **`MPI_ERR_TYPE;`**
Definiert den Fehlerwert, der einen Fehler bei dem angegebenen Typ anzeigt. Bei einigen Funktionen müssen die Argumente bezüglich Typen in allen beteiligten Prozessen übereinstimmen.

`unsigned int` **`MPI_ERR_TAG;`**
Definiert den Fehlerwert, der einen Fehler bei der Kennzeichnung von Operationen anzeigt. Bei einigen Funktionen müssen die Argumente bezüglich Kennzeichnungen in allen beteiligten Prozessen übereinstimmen.

`unsigned int` **`MPI_ERR_COMM;`**
Definiert den Fehlerwert, der einen Fehler bei dem verwendeten Kommunikator anzeigt. Bei einigen Funktionen müssen die Argumente bezüglich Kommunikatoren in allen beteiligten Prozessen übereinstimmen.

`unsigned int` **`MPI_ERR_RANK;`**
Definiert den Fehlerwert, der einen Fehler bei dem angegebenen rank anzeigt. Möglicherweise existiert der angegebene rank nicht im angegebenen Kommunikator.

`unsigned int` **`MPI_ERR_REQUEST;`**
Definiert den Fehlerwert, der einen Fehler beim angegebenen Handle anzeigt. Möglicherweise zeigt das Handle noch auf nichts oder es wird im falschen Kontext verwendet. Vielleicht soll die Beendigung einer Operation getestet oder darauf gewartet werden, obwohl die Operation schon beendet oder abgebrochen wurde.

`unsigned int` **`MPI_ERR_ROOT;`**
Definiert den Fehlerwert, der einen Fehler bei dem angegebenen Besitzer anzeigt. Der Besitzer einer Funktion muß im angegebenen Kommunikator existieren. Bei einigen Funktionen müssen die Argumente bezüglich eines Besitzers in allen beteiligten Prozessen übereinstimmen.

`unsigned int` **`MPI_ERR_GROUP;`**
Definiert den Fehlerwert, der einen Fehler bei der angegebenen Gruppe anzeigt. Bei einigen Funktionen müssen die Argumente bezüglich Gruppen in allen beteiligten Prozessen übereinstimmen.

`unsigned int` **`MPI_ERR_OP;`**
Definiert den Fehlerwert, der einen Fehler bei der verwendeten Operation anzeigt. Bei einigen Funktionen müssen die Argumente bezüglich Operationen in allen beteiligten Prozessen übereinstimmen.

`unsigned int` **`MPI_ERR_TOPOLOGY;`**

Definiert den Fehlerwert, der einen Fehler bei einer Topologie angibt. Dieser Fehler tritt auf, falls man versucht, die Art der Topologie eines Kommunikators festzustellen, dem keine Topologie zugeordnet wurde.

`unsigned int` **`MPI_ERR_DIMS;`**
Definiert den Fehlerwert, der einen Fehler bei den Dimensionen eines kartesischen Gitters angibt.

`unsigned int` **`MPI_ERR_ARG;`**
Definiert der Fehlerwert, der einen Fehler bei einem Funktionsargument angibt.

`unsigned int` **`MPI_ERR_UNKNOWN;`**
Definiert den Fehlerwert, der anzeigt, daß ein unbekannter Fehler in MPI aufgetreten ist.

`unsigned int` **`MPI_ERR_TRUNCATE;`**
Definiert den Fehlerwert, der anzeigt, daß ein Puffer beim Empfang abgeschnitten werden mußte.

`unsigned int` **`MPI_ERR_INTERN;`**
Definiert den Fehlerwert, der anzeigt, daß ein interner Fehler in MPI aufgetreten ist.

`unsigned int` **`MPI_ERR_IN_STATUS;`**
Zeigt an, daß der Fehlerwert in der `MPI_Status`-Variable zu finden ist.

`unsigned int` **`MPI_ERR_PENDING;`**
Definiert den Fehlerwert, der anzeigt, daß die Beendigung einer Operation noch aussteht.

`unsigned int` **`MPI_ERR_LASTCODE;`**
Definiert den größten Fehlerwert in MPI. Dieser Wert stellt keinen Fehler dar, sondern kann dazu benutzt werden, um abzufragen, ob ein Rückgabewert ein Fehlerwert ist. Das ist bei allen Funktionen außer `MPI_Wtick` und `MPI_Wtime` genau dann der Fall, wenn der Rückgabewert zwischen 0 (=`MPI_SUCCESS`) inklusive und `MPI_ERR_LASTCODE` exklusive liegt.

`unsigned int` **`MPI_ERR_OTHER;`**
Definiert den Fehlerwert, der anzeigt, daß ein Fehler aufgetreten ist, der in keine der oben genannten Kategorien fällt.

`unsigned int` **`MPI_STATUS_SIZE;`**
Gibt unter Fortran77 die Größe des Feldes MPI_Status an.

Referenz - Sonstiges

Hinweis: Einige hier aufgeführten Definitionen sind nicht von allen MPI-Implementationen übernommen worden. Außerdem sind einige Definitionen entweder nur für C oder nur für Fortran77 sinnvoll.
Desweiteren sind einige hier ausgeführten Funktionen Wrapper-Funktionen für andere Funktionen, die hier nicht aufgelistet sind.

`double` **`MPI_Wtick`**`(void);`
Ermittelt die Anzahl der Sekunden, die einem "System-Tick" entsprechen.
Parameter:
keine

`double` **`MPI_Wtime`**`(void);`
Ermittelt die Anzahl der Sekunden, die seit dem letzten Aufruf dieser Funktion vergangen sind.
Parameter:
keine

`int` **`MPI_Probe`**`(int source, int tag, MPI_Comm comm, MPI_Status *status);`
Prüft blockierend, ob eine Nachricht von einem Prozeß mit einer bestimmten Kennzeichnung anliegt.
Parameter:
`source` - Nummer des Prozesses, der geprüft werden soll
`tag` - Kennzeichnung der Operation
`comm` - Kommunikator, in dem die Operation ablaufen soll
`status` - Zeiger auf eine `MPI_Status`-Variable, in die der Status der Operation eingetragen wird

Diese Funktion simuliert eine blockierende Empfangsoperation `MPI_Recv`. Es werden jedoch keine (sichtbaren) Daten übertragen. In die Variable, auf die `flag` zeigt, wird eine 0 eingetragen, falls der Prozeß mit der Nummer `source` keine Nachricht mit der Kennzeichnung `tag` gesendet hat, ansonsten ein von 0 verschiedener Wert.

`int` **`MPI_Iprobe`**`(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);`
Prüft nichtblockierend, ob eine Nachricht von einem Prozeß mit einer bestimmten Kennzeichnung anliegt.
Parameter:
`source` - Nummer des Prozesses, dessen Erreichbarkeit überprüft werden soll
`tag` - Kennzeichnung der Operation
`comm` - Kommunikator, in dem die Operation ablaufen soll
`flag` - Zeiger auf eine `int`-Variable, in die das Ergebnis eingetrage wird
`status` - Zeiger auf eine `MPI_Status`-Variable, in die der Status der Operation eingetragen wird

Diese Funktion simuliert eine blockierende Empfangsoperation `MPI_Irecv`. Es werden jedoch keine (sichtbaren) Daten übertragen. In die Variable, auf die `flag` zeigt, wird eine 0 eingetragen, falls der Prozeß mit der Nummer `source` keine Nachricht mit der Kennzeichnung `tag` gesendet hat, ansonsten ein von 0 verschiedener Wert.

`int` **`MPI_Pcontrol`**`(int level, ...);`
Veranlaßt MPI zur Selbstüberwachung.
Parameter:
`level` - Stufe der Überwachung

Weitere Parameter sind von der Implementierung von MPI abhängig. Die nullte Stufe schaltet die Überwachung aus. Die erste Stufe der Überwachung besteht darin, daß MPI mittels `MPI_Wtime` bei jedem Funktionsaufruf die gemessene Zeit ermittelt. Die zweite Stufe erzeugt Log-Dateien, die die Kommunikation aufzeichnen, und die

dritte Stufe benutzt MPE, um die Kommunikation graphisch zu veranschaulichen.

`int` **`MPI_Attr_delete`**`(MPI_Comm comm, int key);`
Diese Funktion löscht den Wert eines Schlüssels eines Kommunikator.
Parameter:
`comm` - der Kommunikator
`key` - der Schlüssel, der gelöscht werden soll

`int` **`MPI_Attr_put`**`(MPI_Comm comm comm, int key, void *value);`
Gibt einem Schlüssele eines Kommunikators einen Wert.
Parameter:
`comm` - der Kommunikator
`key` - der Schlüssel
`value` - Zeiger auf den Wert

`int` **`MPI_Attr_get`**`(MPI_Comm comm, int key, void *value, int *found);`
Ermittelt ein Attributwert eines Schlüssels eines Kommunikators.
Parameter:
`comm` - der Kommunikator
`key` - der Schlüssel
`value` - Zeiger auf einen Speicherbereich, in den der Attributwert eingetragen wird, sofern `comm` den Schlüssel `key` hat
`found` - Zeiger auf eine `int`-Variable, in die eine 1 eingetragen wird, falls `comm` den Schlüssel `key` hat, und 0 sonst

`int` **`MPI_Keyval_create`**`(MPI_Copy_function *cf, MPI_Delete_function *df, int *key, void *extra);`
Erzeugt einen neuen Schlüssel für einen Kommunikator.
Parameter:
`cf` - Zeiger auf eine Funktion, die aufgerufen wird, wenn der Schlüssel kopiert wird
`df` - Zeiger auf eine Funktion, die aufgerufen wird, wenn der Schlüssel gelöscht wird
`key` - Zeiger auf den Schlüssel, der erzeugt wird
`extra` - Zeige auf Extradaten für diesen Schlüssel

`int` **`MPI_Keyval_free`**`(int *key);`
Löscht einen Schlüssel.
Parameter:
`key` - Zeiger auf den Schlüssel

`typedef int` **`MPI_Copy_function`**`(MPI_Comm, int, void *, void *, void *, int *);`
Definiert einen Callback-Funktionsprototypen, dessen Instanzen benutzt werden, um Schlüssel zu kopierren

`typedef int` **`MPI_Delete_function`**`(MPI_Comm, int, void *, void *);`
Definiert einen Callback-Funktionsprototypen, dessen Instanzen benutzt werden, um Schlüssel zu löschen

`MPI_Copy_function` **`MPI_NULL_COPY_FN;`**
Definiert eine Kopierfunktion, die nichts macht. Diese Funktion wird bereits von MPI implementiert.

`MPI_Copy_function` **`MPI_DUP_FN;`**
Definiert eine Kopierfunktion, die eins zu eins kopiert. Diese Funktion wird bereits von MPI implementiert.

`MPI_Delete_function` **`MPI_NULL_DELETE_FN;`**
Definiert eine Löschfunktion, die nichts macht. Diese Funktion wird bereits von MPI implementiert.

`unsigned int` **`MPI_KEYVAL_INVALID;`**
Definiert den Schlüssel, der einen ungültigen Schlüssel angibt.

`unsigned int` **`MPI_TAG_UB;`**
Definiert den Schlüssel, dessen Wert die Obergrenze für Kennzeichnungen von Operationen ist. Die Untergrenze ist immer 0.

`unsigned int` **`MPI_HOST;`**
Definiert den Schlüssel, dessen Wert angibt, welcher Prozeß in `MPI_COMM_WORLD` der Host ist (sofern es einen solchen gibt). Gibt es keinen Host, ist der zugehörige Wert `MPI_PROC_NULL`.

`unsigned int` **`MPI_IO;`**
Definiert den Schlüssel, dessen Wert angibt, welcher Prozeß Eingaben und Ausgaben tätigen kann. Wenn jeder Prozeß dies kann, dann ist der zugehörige Wert `MPI_ANY_SOURCE`, wenn kein Prozeß dies kann, ist der Wert `MPI_PROC_NULL`.

`unsigned int` **`MPI_WTIME_IS_GLOBAL;`**
Dieser Schlüssel hat den Wert 1, wenn die Funktion `MPI_Wtime` global synchronisiert ist.

`int` **`MPI_Pack`**`(const void *in, int incount, MPI_Datatype type, void *out, int outsize, int *position, MPI_Comm comm);`
Packt einen Speicherbereich in einen Puffer.
Parameter:
`in` - Zeiger auf einen Speicherbereich, der die zu packenden Daten enthält
`incount` - Anzahl der Daten, die gepackt werden sollen
`type` - Typ der Daten, die gepackt werden sollen
`out` - Zeiger auf den Speicherbereich, in dem der Puffer gehalten wird
`outsize` - Größe des Puffers
`position` - Zeiger auf eine `int`-Variable, die die Position angibt, an der die zu packenden Daten in den Puffer eingefügt werden
`comm` - der Kommunikator, in dem die Funktion aufgerufen wird

Der Zeiger position zeigt nach dem erfolgreichen Aufruf dieser Funktion auf die Position, an der das nächste Element in den Puffer eingefügt werden kann. Dieser Wert (also diese Stelle im Speicher) kann also für den nä‚chsten Aufruf dieser Funktion verwendet werden. Der Wert von outsize kann mit der Funktion MPI_Pack_size ermittelt werden.

int **MPI_Unpack**(const void *in, int insize, int *position, void *out, int outcount, MPI_Datatype type, MPI_Comm comm);
Liest einen Speicherbereich, der mit MPI_Pack gepackt wurde, aus.
Parameter:
in - Zeiger auf einen Speicherbereich, der die zu entpackenden Daten enthält
insize - Größe;e des Puffers
position - Zeiger auf eine int-Variable, die die Position angibt, an der die zu entpackenden Daten ausgelesen werden sollen
out - Zeiger auf den Speicherbereich, in dem die entpackten Daten eingetragen werden
outcount - Anzahl der Daten, die entpackt werden sollen
type - Typ der Daten, die entpackt werden sollen
comm - der Kommunikator, in dem die Funktion aufgerufen wird

Der Zeiger position wird beim Aufruf dieser Funktion aktualisiert. Nach dem Aufruf dieser Funktion zeigt er auf das nächste Element, das entpackt werden kann. Somit kann der Wert für den nächsten Aufruf dieser Funktion benutzt werden. Der Wert von insize kann mit folgendem Code-Fragment ermittelt werden :

```
MPI_Recv(in, count, MPI_PACKED, source, tag, comm, &status);
MPI_Get_count(&status, MPI_PACKED, &insize);
```

int **MPI_Pack_size**(int incount, MPI_Datatype type, MPI_Comm comm, int *size);
Ermittelt die Größe eines gepackten Puffers.
Parameter:
incount - Anzahl der Elemente, die der Puffer aufnehmen soll
type - Typ der Elemente, die der Puffer aufnehmen soll
comm - Kommunikator, in dem die Funktion aufgerufen wird
size - Zeiger auf eine int-Variable, in die die Größe des Puffer eingetragen wird

Beim Funktionsaufruf wird nicht der Puffer selbst angegeben, sondern dessen Spezifikationen (Anzahl und Typ der Elemente). Deshalb ist es auch mö‚glich, diese Funktion anzuwenden, ohne daß wirklich ein Puffer existiert, z.B., um die vorraussichtliche Größe eines Puffers zu ermitteln. Es kann dann mit buffer=malloc((unsigned) *size) der Puffer dynamisch reserviert werden.

int **MPI_Get_processor_name**(char *name, int *length);
Ermittelt den Namen des Prozessors, auf dem MPI gerade ausgeführt wird.
Parameter:
name - char-Feld, in das der Prozessorname eingetragen wird
length - Zeiger auf eine int-Variable, in die die Länge des Prozessornamen eingetragen wird

Nach dem Aufruf dieser Funktion steht der Prozessorname im Feld c[0] bis c[length-1] zur Verfügung.

unsigned int **MPI_MAX_PROCESSOR_NAME;**
Definiert die maximale Länge des char-Feldes, das von MPI_Get_processor_name zurückgegeben werden kann.

int **MPI_Get_version**(int *major, int *minor);

Ermittelt die verwendete Version von MPI.
Parameter:
`major` - Zeiger auf eine `int`-Variable, in die die Hauptversion eingetragen wird
`minor` - Zeiger auf eine `int`-Variable, in die die Nebenversion eingetragen wird

Die tatsächliche Version von MPI ist major.minor.

`#define` **`MPI_VERSION`** `1`
Definiert die MPI-Hauptversion.

`#define` **`MPI_SUBVERSION`** `2`
Definiert die MPI-Unterversion.

`unsigned int` **`MPI_UNDEFINED;`**
Definiert die Konstante, die angibt daß ein Rückgabewert undefiniert ist. Dieser Wert wird auch zurückgegeben, wenn ein mit `MPI_Topo_test` getesteter Komunikator weder Graph noch kartesisches Gitter ist.

`unsigned int` **`MPI_PROC_NULL;`**
Definiert die Konstante, die einen Dummy-Empfänger oder Dummy-Sender angibt. Diese Konstante ist auch der Wert des Schlüssels `MPI_HOST`, falls es keinen ausgezeichneten Host-Prozeß in `MPI_COMM_WORLD` gibt, bzw. Wert des Schlüssels `MPI_IO`, falls es keinen Prozeß gibt, der Eingaben und Ausgaben tätigen kann.

Referenz - Verwendete Typen

Hinweis: Einige hier aufgeführten Definitionen sind nicht von allen MPI-Implementationen übernommen worden. Außerdem sind einige Definitionen entweder nur für C oder nur für Fortran77 sinnvoll.
Desweiteren sind einige hier ausgeführten Funktionen Wrapper-Funktionen für andere Funktionen, die hier nicht aufgelistet sind.
`typedef long` **`MPI_Aint;`**
Definiert einen Alias für den Typ `MPI_Aint` (nur für C). In Fortran gibt es diesen Typ nicht, dort ist er durch `INTEGER` zu ersetzen.

`typedef unsigned int` **`MPI_Request;`**
Definiert einen Alias für den Typ `MPI_Request` (nur für C). In Fortran gibt es diesen Typ nicht, dort ist er durch `INTEGER` zu ersetzen.

`typedef unsigned int` **`MPI_Group;`**
Definiert einen Alias für den Typ `MPI_Group` (nur für C). In Fortran gibt es diesen Typ nicht, dort ist er durch `INTEGER` zu ersetzen.

`typedef unsigned int` **`MPI_Comm;`**

Definiert einen Alias für den Typ `MPI_Comm` (nur für C). In Fortran gibt es diesen Typ nicht, dort ist er durch `INTEGER` zu ersetzen.

`typedef unsigned int` **`MPI_Errhandler;`**
Definiert einen Alias für den Typ `MPI_Errhandler` (nur für C). In Fortran gibt es diesen Typ nicht, dort ist er durch `INTEGER` zu ersetzen.

`typedef unsigned int` **`MPI_Op;`**
Definiert einen Alias für den Typ `MPI_Op` (nur für C). In Fortran gibt es diesen Typ nicht, dort ist er durch `INTEGER` zu ersetzen.

`typedef unsigned int` **`MPI_Datatype;`**
Definiert einen Alias für den Typ `MPI_Datatype` (nur für C). In Fortran gibt es diesen Typ nicht, dort ist er durch `INTEGER` zu ersetzen.

```
typedef struct {
 int MPI_SOURCE;
 int MPI_TAG;
 int MPI_ERROR;
 int size;
 int reserved[2];
} MPI_Status;
```
Definiert eine Strukturvariable (nur für C) oder eine Feldvariable (nur für Fortran77), deren Instanzen bei allen Empfangsoperationen, Testoperationen und Warteoperationen als `MPI_Status`-Variablen eingesetzt werden können.