Überblick über MPI

Michael Schröder

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

1 Credits

- Der MPI Standard ist erhältlich über
  `http://www.mcs.anl.gov/mpi/`

- Einige Teile und Ideen sind dem Kurs
  "Introduction to Parallel Computing with MPI"
  © MAN T&EC entnommen.
  `http://unix.hensa.ac.uk/parallel/`

2 MPI Grundlagen

- Entwickelt vom MPI Komitee:
  - ▲ Über 40 Universitäten.
  - ▲ Hardware und Software Hersteller.
  - ▲ Große MP Anwender (z.B.: Forschungseinrichtungen).
- Beginn der Entwicklung: 29. April 1992.
- Mailingliste & Treffen alle 6 Wochen.
- Veröffentlichung des MPI 1.0 Standards am 5. Mai 1994.
- MPI 1.1 Standard veröffentlicht am 12. Juni 1995.
  Kleinere Korrekturen und Anmerkungen.

3 Ziele

- Lauffähig in heterogener Umgebung.
- Einfaches Interface für C und Fortran.
- Effiziente Kommunikation: möglichst wenig Kopieren von Daten und paralleler Ablauf Kommunikation & Berechnungen.
- Verbergung des Kommunikationsprotokolls und Bereitstellung einer sicheren Datenübertragung.
- Kann als Basis für numerische Bibliotheken dienen.

4 Stammbaum

- MPI übernahm das Beste aus
  - Intels nX/2
  - Express
  - PVM
  - p4 (USA - Argone)
  - Chimp (Edinburgh)
  - IBM TJ Watson Research center
  - PICL

5 Mängel

- Aus Zeitgründen wurden einige wichtige Dinge aus MPI-1 weggelassen:
  - Prozeßmanagement (keine *spawn* Funktionen)!
  - Shared Memory Operationen.
  - Erzeugung und Verwaltung von Threads.
  - Parallele I/O.
  - Debugging Hilfen.
- Viele dieser Mängel werden mit MPI-2 behoben sein.

6 Implementierungen

- Gedacht als “Proof of Concept”.
- 3 Hauptimplementierungen:
  - ▲ Argonne National Labs & MSU library: mpich
    Setzt auf PVM oder p4 auf.
  - ▲ Edinburghs CHIMP Implementierung
    Benuzt CHIMP zur Kommunikation.
  - ▲ Ohio Supercomputer Center LAM Implementierung
    LAM steht für Local Area Multicomputer und ist eine Programmierumgebung für parallele Maschinen.

7 Prozeßstart

- MPI-1 definiert keine Möglichkeit, Prozesse zu starten.
- Daher ist es bei den meisten Implementierungen möglich, anzugeben, wieviele Prozesse gestartet werden sollen.
- Diese Angabe geschieht allerdings auf der Kommandozeile.
- Prozesse können nicht (portabel) dynamisch nachgestartet werden.
- MPI-2 soll diese Einschränkung beseitigen (`MPI_Spawn`).
- Im Moment ist nur das SPMD Model portabel.

8 Das SPMD Model

- Beispiel: Ein einfaches Animationsprogramm:

Animations Prozeß

```
for (time = 0; ; time += step) {
 process_scene(time);
 send_graphics_data();
}
```

Display Prozeß

```
for(;;) {
 receive_data();
 display_data();
}
```

- Dies wird häufig durch zwei Programme realisiert.
- MPI bietet aber keine Möglichkeit, mit der zwei Programme kommunizieren können.

9 cont.

- Daher benutzen MPI Programme meist das SPMD Parallelitätsmodel.
- SPMD steht für Single Program Multiple Data
- Der Anwender schreibt nur ein Programm, das intern in die gewünschten Funktionalitäten verzweigt.
- Für das Beispiel könnte das folgendermaßen aussehen:

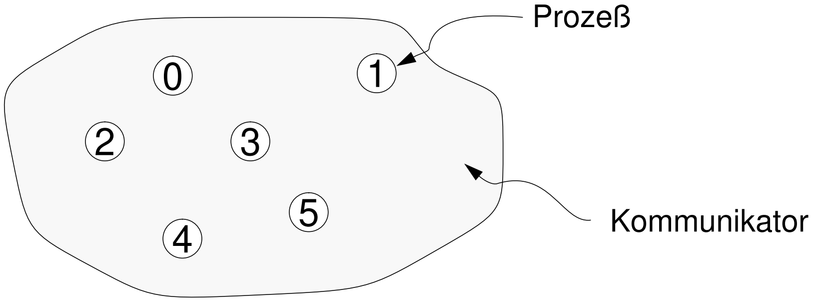
```
if (running_on_first_processor)
      animation_process();
else
      display_process();
```

10 MPI Schnittstelle

- MPI Funktionen starten grundsätzlich mit `MPI_`.
- Jede MPI Funktion liefert einen Fehlerstatus zurück. In C geschieht dies über den Rückgabewert (ein `int`), in Fortran über einen zusätzlichen Fehlerparameter.
- MPI Fortran Funktionen bestehen nur aus Großbuchstaben, in C wird der Anfangsbuchstabe des ersten Wortes großgeschrieben.
- Die MPI Bibliothek versteckt interne Implementierungsdetails über sogenannte *Handles*. Diese sind in Fortran vom Typ `INTEGER`, in C existiert für jeden Handle ein `typedef`.
- Fortran Arrays starten bei 1, C Arrays bei 0.

11 Kommunikatoren

- In MPI können Nachrichten nur zu Prozessen verschickt werden, die sich im gleichen Kommunikator (*communicator*) befinden.
- Ein Kommunikator ist eine Menge von Prozessen, die sich untereinander kennen (sollten).



- Ein Prozeß kann mehreren Kommunikatoren angehören.

12 cont.

- Jeder Prozeß besitzt eine dem Kommunikator zugeordnete Nummer, der sogenannte Rang (*rank*).
- Das Tupel (Kommunikator, Rang) adressiert eindeutig einen Prozeß.
- Gehört ein Prozeß mehreren Kommunikatoren an, können die jeweiligen Ränge unterschiedlich sein.
- Wenn ein Programm `MPI_Init()` aufruft, erzeugt MPI einen Standardkommunikator, dem alle Prozesse angehören.
- Dieser Kommunikator wird mit `MPI_COMM_WORLD` bezeichnet.

13 Wozu Kommunikatoren?

- Message Passing Programmierung ist vielen Anwendern zu schwer.
- Deshalb wollen sie Bibliotheken, die ihnen die komplizierte Arbeit abnehmen.
- Diese Bibliotheken sollen ihre interne Funktion nach außen hin verbergen. Falls die Bibliothek Daten verschickt, soll der Benutzer nichts davon merken.
- In MPI läßt sich dieses Problem leicht lösen, indem die Bibliothek einen eigenen Kommunikator erzeugt.

14 MPI Programmierung

- Die erste MPI Funktion, die ein Programm aufruft, muß `MPI_Init(&argc, &argv);` sein.
- Dies erlaubt der MPI Implementierung, alle notwendigen Initialisierungen (z.B. Prozeßsynchronisation) durchzuführen.
- In Fortran wird nur der Fehlerparameter übergeben: `MPI_INIT(IERR)`
- Alle MPI Konstanten sind im Headerfile `mpi.h` gespeichert (in Fortran `mpif.h`). Daher sollte jedes MPI Programm mit `#include <mpi.h>` beginnen.

15 Beenden von MPI

- Am Ende jedes MPI Programms sollte `MPI_Finalize();` aufgerufen werden.
- Dies ermöglicht der Bibliothek, noch nicht abgeholte Nachrichten und andere dynamische Daten freizugeben.
- Alternativ dazu können auch mit `MPI_Abort(MPI_Comm comm, int errcode);` alle Prozesse eines Kommunikators angehalten werden.
- Wenn als Kommunikator `MPI_COMM_WORLD` verwendet wird, terminiert das ganze Program.

16 Fehlerbehandlung

- Jeder MPI Aufruf liefert einen Integer zurück, der gleich `MPI_SUCCESS` ist, wenn der Aufruf erfolgreich bearbeitet werden konnte.

- Fehler führen normalerweise zum Programmabbruch, aber es ist möglich, einen *Errorhandler* in die Kommunikatoren einzutragen.

- MPI besitzt zwei vordefinierte Errorhandler:
  - ▲ `MPI_ERRORS_ARE_FATAL` (default Handler)
  - ▲ `MPI_ERRORS_RETURN`

17 Abfrage der Prozeßinformationen

- Nachdem durch das Laufzeitsystem eine Anzahl von Prozessen gestartet wurde, ist es für das SPMD Programmiermodel notwendig, den Rang eines Prozesses (also sozusagen die PID) abzufragen.

- MPI stellt dazu zwei Funktionen zur Verfügung:
  - ▲ `MPI_Comm_rank(MPI_Comm comm, int *rank);`
    setzt `rank` auf den Rang des Prozesses innerhalb des Kommunikators `comm`.
  - ▲ `MPI_Comm_size(MPI_Comm comm, int *size);`
    setzt `size` auf die Anzahl der Prozesse innerhalb des Kommunikators `comm`.

18 MPI Beispiel 1

- Dieses Beispiel gibtfür jeden gestarteten Prozeß den Rang innerhalb `MPI_COMM_WORLD` aus:

```
#include <mpi.h>

int main(int argc, char **argv) {
    int r, s;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &s);
    MPI_Comm_rank(MPI_COMM_WORLD, &r);
    printf("Process %d of %d\n", r + 1, s);
    MPI_Finalize();
    exit(0);
}
```


19 cont.

- Compilieren erfolgt auf der Convex SPP1600 durch:

```
$ mpicc b1.c -o b1
```

- Beim Aufruf kann mit “`-np`” die Anzahl der Prozessoren festgelegt werden:

```
$ ./b1 -np 4
Process 4 of 4
Process 1 of 4
Process 3 of 4
Process 2 of 4
```

- Die Verteilung auf mehrere Rechner geschieht über `mpirun`.

20 MPI Nachrichten

- Die Kommunikation zwischen Prozessen geschieht über das Senden und Empfangen von Nachrichten.
- In MPI besteht jede Nachricht aus einer Anzahl von Elementen, die alle vom selben MPI-Typ sind.

integer	integer	integer	integer	integer

- Der Typ muß sowohl beim Senden als auch beim Empfangen einer Nachricht angegeben werden.
- Die Anzahl der Elemente wird beim Empfang zurückgeliefert.

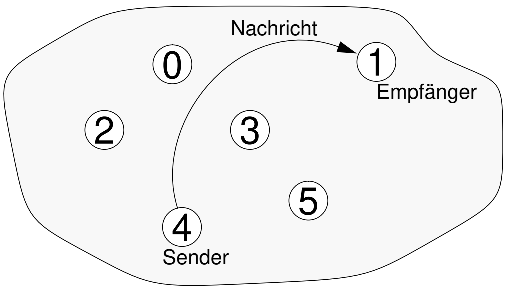
21 MPI Standardtypen

- MPI unterstützt folgende C Datentypen:

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

22 Punkt zu Punkt Kommunikation

- Punkt zu Punkt Kommunikation ist der grundlegende Mechanismus, in MPI Nachrichten auszutauschen.
- Die Nachricht wird innerhalb eines Kommunikators verschickt. Der Zielprozeß wird über seinen Rang im Kommunikator adressiert.



23 Senden einer Nachricht

- Der Grundlegende Befehl zum Senden einer Nachricht ist:

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

- Dieser Befehl verschickt `count` Elemente vom MPI-Typ `datatype` an den Prozeß mit Rang `dest` im Kommunikator `comm`.
- Die Nachricht kann durch den `tag` Parameter einen zusätzlichen Typ erhalten. Beim Empfangen können so spezielle Nachrichten ausgewählt werden.
- Nach dem Aufruf von `MPI_Send` ist die Nachricht erst einmal unterwegs. Der Empfang kann also erst später stattfinden.

24 Empfangen einer Nachricht

- Der Grundlegende Befehl zum Empfangen ist

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

- Es werden maximal `count` Elemente vom Typ `datatype` nach `buf` geschrieben. Mit `source` (Rang im Kommunikator `comm`) und `tag` kann der Sender eingeschränkt werden. Hier ist auch `MPI_ANY_SOURCE` und `MPI_ANY_TAG` möglich.
- Falls Wildcards verwendet wurden, kann über `status` der Sender abgefragt werden. Weiterhin enthält `status` auch die Anzahl der empfangenen Elemente.

25 Beispiel einer Kommunikation

```
MPI_Comm_rank(MPI_COMM_WORLD, &r);
if (r == 0) {
    MPI_Send("Hello, world\n", 14, MPI_CHAR,
             1, 0, MPI_COMM_WORLD);
} else {
    char buf[256];
    MPI_Status status;
    MPI_Recv(buf, sizeof(buf), MPI_CHAR,
             0, 0, MPI_COMM_WORLD, &status);
    printf("Received %s\n", buf);
}
```

26 Auswertung des Statusobjekts

- Der Status ist eine C Struktur, mit folgenden “öffentlichen” Elementen:
  - `status.MPI_SOURCE`
  - `status.MPI_TAG`
  - `status.MPI_ERROR`
- Die Anzahl der empfangenen Elemente kann mit

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

abgefragt werden (dies ist meist nicht nötig).

27 Kommunikationsarten

- MPI bietet vier verschiedene Kommunikationsarten an:
  - Synchron (`MPI_Ssend`)
    Die Funktion kehrt zurück, wenn der Empfänger mit dem Empfang begonnen hat.
  - Gepuffert (`MPI_Bsend`)
    Die Funktion kehrt zurück, nachdem die Daten in einen lokalen Puffer übertragen wurden.
  - Standard (`MPI_Send`)
    Entweder synchron oder gepuffert.
  - Ready (`MPI_Rsend`)
    Diese Variante darf nur verwendet werden, wenn feststeht, daß der Empfänger die zugehörige Empfangsfunktion aufgerufen hat (dies spart Kommunikationsoverhead).

28 Semantik der Kommunikation

- Ordnungserhaltend
  Nachrichten können sich nicht überholen.

- Fortschreitend
  Wurden zwei passende `MPI_Send` und `MPI_Receive` Aufrufe getätigt, so kehrt einer der beiden Aufrufe garantiert zurück.

- nicht Fair
  MPI garantiert keine Fairness.

- beschränkter Nachrichtenpuffer
  `MPI_Bsend` bricht mit einem Fehler ab, wenn die Nachricht nicht im Puffer gespeichert werden kann.

29 Bereitstellen des Nachrichtenpuffers

- MPI bietet die Möglichkeit, den Nachrichtenpuffer zu setzen:
  `MPI_Buffer_attach(void *buffer, int size)`

  ▲ Gepuffert wird auf der Senderseite.

  ▲ Es kann nur ein Puffer bereitgestellt werden.

- Der Puffer kann mit folgendem Aufruf entfernt werden:
  `MPI_Buffer_detach(void *buffer, int *size)`

- Falls kein Puffer gesetzt wurde, liefert jeder `MPI_Bsend` Aufruf einen Fehler zurück.

30 Nicht-blockierende Operationen

- Die bisher vorgestellten Sende/Empfangsoperationen waren *blockierend*, daß heißt, der Aufruf kehrt erst bei Erfolg oder Fehler zurück.
- Dies kann zu Performanceeinbußen führen, da in der Zeit, in der auf das Ende der Operation gewartet werden muß, gerechnet werden könnte.
- Ein Ausweg ist die Verwendung eines Nachrichtenpuffers, aber hierbei muß die Nachricht einmal mehr kopiert werden.
- MPI bietet deshalb Operationen an, die sich nicht blockieren. Der Aufrufer erhält einen *handle* auf die Operation zurück, den er später auf Erfolg der Operation überprüfen kann.

31 cont.

- Die nicht-blockierenden Operationen zeichnen sich durch ein ‘`I`’ im Namen (für *Immediate*) und einen zusätzlichen Parameter aus (beim Empfang entfällt natürlich der Statusparameter). Der Sendeaufruf sieht demnach so aus:
```
MPI_Isend(void *buf, int count,
          MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm,
          MPI_Request *request)
```
- In `request` wird das Aufrufobjekt zurückgeliefert.
- Analog gibt es `MPI_Issend`, `MPI_Ibsend`, `MPI_Irsend`, und `MPI_Irecv`.

32 Benutzung des Aufrufobjektes

- Warten auf das Ende des Aufrufes:

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

- `status` ist ein Zeiger auf ein MPI Statusobjekt, wie es schon beim Nachrichtenempfang beschrieben wurde.
- `request` wird durch den Aufruf auf `MPI_REQUEST_NULL` gesetzt.
- Soll nicht auf das Ende des Aufrufs gewartet werden, kann dies dem System mit
  `MPI_Request_free(MPI_Request *request)`
  mitgeteilt werden.

33 cont.

- Soll nur überprüft werden, ob ein Aufruf geendet hat, aber nicht auf das Ende gewartet werden, so kann dies mit
  `MPI_Test(request, flag, status)`
  geschehen.
- `flag` gibt an, ob der Aufruf beendet wurde und das `status` Objekt gefüllt wurde.
- `MPI_Wait` und `MPI_Test` gibt es auch in Varianten, die mehrere Aufrufobjekte gleichzeitig testen. Hierbei gibt es drei Typen, die `...any`, `...all` und `...some` heißen.
- `any` liefert ein Ergebnis zurück, `all` alle und `some` soviel wie möglich.

34 Abbruch eines laufenden Aufrufs

- Mit
  `MPI_Cancel(MPI_Request *request)`
  wird dem System mitgeteilt, daß der Aufruf abgebrochen werden soll.

- Da nicht garantiert ist, daß der Aufruf abgebrochen wird, muß immer noch `MPI_Wait` oder `MPI_Test` oder `MPI_Request_free` aufgerufen werden.

- Bei einem abgebrochenen Befehl liefert

```
MPI_Test_cancelled(MPI_Status *status,
                   int *flag)
```

  in `flag` `true` zurück.

35 Test auf Empfangbarkeit

- MPI bietet die Möglichkeit, abzufragen, ob eine Nachricht empfangen werden könnte. Das Status Objekt, welches beim Empfang entstehen würde, kann schon vorab abgefragt werden. Dies ist nützlich, wenn Speicherplatz für den Empfang bereitgestellt werden muß.

```
MPI_Iprobe(int source, int tag, MPI_Comm comm,
           int *flag, MPI_Status *status)
```

- in `flag` wird `true` zurückgeliefert, wenn der Empfang möglich wäre.

- Es gibt den Aufruf auch in blockierender Form: `MPI_Probe`. Hier wird solange gewartet, bis der Empfang durchgeführt werden kann. Der `flag` Parameter entfällt natürlich.

36 Kombination von Senden + Empfang

- Da es bei zyklischen Sende/Empfangsaufrufen leicht zu Verklemmungen kommen kann, bietet MPI zwei kombinierte Sende- und Empfangsaufrufe an:

```
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)
```

- Bei `MPI_Sendreceive_replace` entfällt `recvbuf`, `recvcount` und `recvtype`.

37 Abgeleitete Datentypen

- So wie es in C Sinn macht, zusammengehörige Dinge zu einer Struktur zusammenzufassen, macht es Sinn, sie als eine Einheit zu verschicken und zu empfangen.
- MPI bietet daher die Möglichkeit, aus vorhandenen Datentypen einen neuen abzuleiten. Dieser kann dann überall verwendet werden, wo in MPI ein Datentyp verlangt wird.
- Beim Empfang muß der gleiche Datentyp angegeben werden, deshalb müssen die Datentypen überall gleich zusammengesetzt sein!

38 cont.

- Die einfachste Art, einen neuen Datentyp zu bilden, ist

```
MPI_Type_contiguous(int count,
                    MPI_Datatype oldtyp,
                    MPI_Datatype *newtype)
```

- Der neue Datentyp besteht aus `count` alten Datentypen.

- Nachdem ein neuer Datentyp erzeugt wurde, muß er zur Verwendung freigegeben werden. Dies geschieht mit:

```
MPI_Type_commit(MPI_Datatype *datatype)
```

MPI kann dadurch z.B. eine interne Repräsentation erzeugen.

- Wird ein Datentyp nicht mehr benötigt, kann er mit

```
MPI_Type_free(MPI_Datatype *datatype)
```

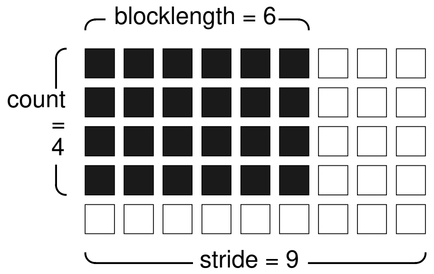
freigegeben werden.

39 cont.

- Die nächste Komplexitätsstufe ist

```
MPI_Type_vector(int count, int blocklegth,
                int stride,
                MPI_Datatype oldtype,
                MPI_Datatype *newtype)
```

Mit diesem Aufruf können einzelne Blöcke aus einer Matrix extrahiert werden:



40 cont.

- Den `MPI_Type_vector` Aufruf gibt es auch als `MPI_Type_hvector` (h steht für heterogen).
- Er ist identisch zu `MPI_Type_vector` mit einer Ausnahme: `stride` gibt den Abstand zum nächsten Block in Bytes an.
- Das vorhergegangene Beispiel läßt sich daher auch folgendermaßen schreiben:

```
MPI_Type_continous(6, MPI_DOUBLE, &vectyp);
MPI_Type_hvector(4, 1, 9 * sizeof(double),
                 vectyp, &blocktyp);
```

- `MPI_Type_hvector` ist zum Beispiel nötig, wenn aus einer dreidimensionalen Matrix ein Block herausgeschnitten werden soll.

41 cont.

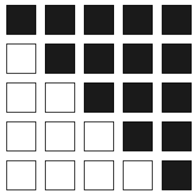
- Die nächste Erweiterung ist `MPI_Type_indexed`. Hier setzt sich der neue Datentype wieder aus mehreren Blöcken zusammen, aber für jeden Block kann die Länge und der Offset spezifiziert werden:

```
MPI_Type_indexed(int count,
                 int *array_of_blocklengths,
                 int *array_of_displacements,
                 MPI_Datatype oldtype,
                 MPI_Datatype *newtype)
```

- `array_of_blocklengths` und `array_of_displacements` sind Felder mit jeweils `count` Elementen. Wenn `MPI_Type_hindexed` verwendet wird, sind die Displacements Offsets in Bytes.

42 cont.

- Beispiel: Extraktion einer oberen Dreiecksmatrix:



```
int blockl[5] = {5, 4, 3, 2, 1};
int disp[5]   = {0, 5+1, 2*5+2, 3*5+3,
                 4*5+4};

MPI_Type_indexed(5, blockl, disp,
                 MPI_DOUBLE, &newtype);
```


43 cont.

- Der allgemeinste Fall ist `MPI_Type_struct`. Hier kann jeder Block von einem Anderen Typ sein:

```
MPI_Type_struct(int count,
                int *array_of_blocklengths,
                MPI_Aint *array_of_dispments,
                MPI_Datatype *array_of_types,
                MPI_Datatype *newtype)
```

- Die Displacements sind hier Offsets in Bytes, da kein gemeinsamer Datentyp zugrundeliegt.

- Mit dieser Funktion ist es möglich, einen Datentyp für eine normale C Struktur zu erstellen. Das Problem ist nur die Berechnung der Displacements.

44 Strukturen

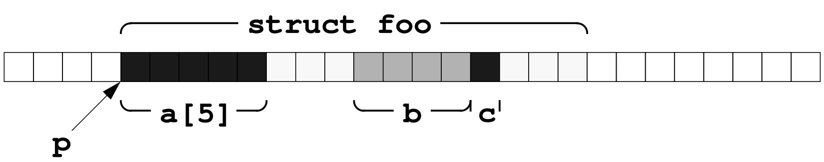
- Viele Prozessoren besitzen sogenannte *alignment restrictions*. So muß z.B. ein “int” auf einer durch 32 teilbaren Adresse liegen, ein “double” auf einer durch 64 teilbaren usw. Greift man auf eine Adresse mit falschen Alignment zu, so resultiert daraus ein *Bus Error*.
- Wird in C eine Struktur deklariert, muß der Compiler die Alignment Restrictions berücksichtigen. Daraus resultiert, daß in der Struktur Löcher sind (sogenanntes *padding*) und die Struktur selbst nur an bestimmten Adressen plaziert werden darf.

45 cont.

- Beispiel für Padding:

```
struct foo {
    char a[5];
    int b;
    char c;
} p;
```

- Die Struktur sieht folgendermaßen im Speicher aus:



46 cont.

- Da FORTRAN keinen Adreßoperator besitzt und der ANSI-C '&' Operator Pointer und keine absoluten Adressen zurückliefert, stellt MPI eine Funktion `MPI_Address(void *location, MPI_Aint *address)` zu Verfügung. Damit kann man die Displacements für das vorige Beispiel folgendermaßen ausrechnen:

```
MPI_Aint displ[3];
struct foo dummy;
MPI_Address(dummy.a, displ);
MPI_Address(&dummy.b, displ + 1);
MPI_Address(&dummy.c, displ + 2);
for (i = 2; i >= 0; i--)
    displ[i] -= displ[0];
```


47 cont.

- Der neue Datentyp kann dann folgendermaßen definiert werden:

```
MPI_Datatype foo_type;
MPI_Datatype type[3] = {MPI_CHAR, MPI_INT,
                        MPI_CHAR};
int blockl[3] = {5, 1, 1};
MPI_Type_struct(3, blockl, displ, type,
                &foo_type);
```

- Man kann die Größe eines MPI Datentyps über `MPI_Type_extent(MPI_Datatype *datatype, MPI_Aint *size)` abfragen. Dies entspricht dem C `sizeof()` Operator.

48 cont.

- ANSI-C spezifiziert ein `offsetof(type, member)` Makro (in `<stddef.h>`), das hervorragend zur Berechnung der Displacements eingesetzt werden kann:

```
MPI_Aint displ[3] = {
     (MPI_Aint)offsetof(struct foo, a[0]),
     (MPI_Aint)offsetof(struct foo, b),
     (MPI_Aint)offsetof(struct foo, c)
};
```

- Hierbei “berechnet” der Compiler die Displacements. Das Macro ist normalerweise folgendermaßen definiert:

```
#define offsetof(s, m) \
                    (size_t)(&(((s *)0)->m))
```


49 Kollektive Kommunikation

- Im Gegensatz zur Punkt zu Punkt Kommunikation sind bei der kollektiven Kommunikation alle Prozesse eines Kommunikators beteiligt.
- Jeder dieser Prozesse muß den gleichen Funktionsaufruf tätigen.
- Folgende Arten kollektiver Aufrufe sind möglich:
  - ▲ Synchronisation durch eine Barriere.
  - ▲ Broadcast einer Nachricht zu allen Prozessen.
  - ▲ Scatter/Gather und Abarten.
  - ▲ Globale Anwendung von Funktionen (sum/max/min/...). Hierbei kann man auch eigene Funktionen definieren.

50 Barrierensynchronisation

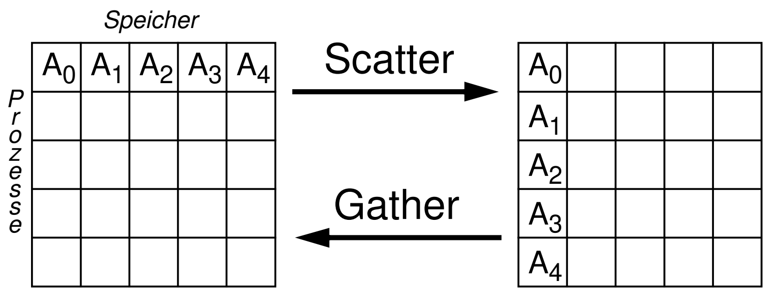
- MPI bietet eine Barriere als Synchronisationsmechanismus an, mit deren Hilfe sich alle Prozesse eines Kommunikators synchronisieren können.
- Die Syntax ist:
  `MPI_Barrier(MPI_Comm comm)`
- Der Befehl muß von allen Prozessen eines Kommunikators aufgerufen werden.

51 Broadcast

- Mit Hilfe von
  ```
  MPI_Bcast(void *buffer, int count,
            MPI_Datatype datatype,
            int root, MPI_Comm comm)
  ```
  wird die Nachricht vom Knoten mit Rang `root` zu allen anderen Prozessen des Kommunikators `comm` geschickt.
- Wie bei allen anderen kollektiven Operationen gibt es keine nichtblockierende (“`I`”) Form!

52 Gather/Scatter

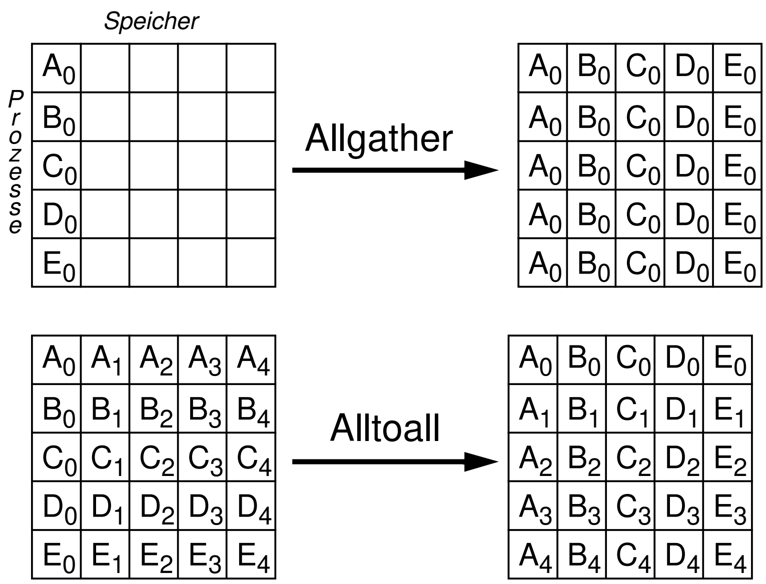
- MPI bietet folgende Gather/Scatter Grundoperationen an:



- Ein Scatter Aufruf verteilt die Daten eines Prozesses auf alle anderen eines Kommunikators.
- Ein Gather Aufruf sammelt die Daten aller Prozesse ein und speichert sie auf einem Zielprozeß.

53 Weitere Gatherabarten

- Zusätzlich zu den Scatter/Gatherfunktionen existieren noch zwei weitere Aufrufe:



54 Scatter/Gather Syntax

- ```
  MPI_Gather(void *sendbuf, int sendcount,
             MPI_Datatype sendtype,
             void *recvbuf, int recvcount,
             MPI_Datatype recvtype,
             int root, MPI_Comm comm)
  ```
- `MPI_Scatter` verwendet die gleichen Argumente.
- `MPI_Allgather` und `MPI_Alltoall` verwenden die gleichen Argumente, allerdings fehlt bei ihnen das `root` Argument, welches den Ziel/Quellprozeß beschreibt.
- `sendcount` und `recvcount` beschreiben jeweils einen einzelnen Datenbereich, bei `MPI_Scatter` wird z.B. ein Bereich von size(`comm`) x `sendcount` verschickt.

55 Globale Operationen

- MPI bietet die Möglichkeit an, eine Operation auf alle Prozesse eines Kommunikators anzuwenden.
- Es gibt 12 vordefinierte Operationen:

Name	Wirkung	Name	Wirkung
MPI_MAX	Maximum	MPI_MIN	Minimum
MPI_SUM	Summe	MPI_PROD	Produkt
MPI_LAND	Logisches AND	MPI_BAND	Bitweises AND
MPI_LOR	Logisches OR	MPI_BOR	Bitweises OR
MPI_LXOR	Logisches XOR	MPI_BXOR	Bitweises XOR
MPI_MAXLOC	Maximum und dessen Position	MPI_MINLOC	Minimum und dessen Position

- Die Operationen müssen assoziativ sein.
  ```

56 cont.

- Syntax des Aufrufes:

```
MPI_Reduce(void *sendbuf, void *recvbuf,
           int count,
           MPI_Datatype datatype,
           MPI_Op op,
           int root, MPI_Comm comm)
```

`root` ist der Rang des Prozesses, auf dem das Ergebnis gespeichert werden soll. Alle anderen Prozesse ignorieren das `recvbuf` Argument.

- Ist `count > 1`, wird die Operation parallel auf alle Elemente angewandt. Der Aufruf berechnet dann also `count` Ergebnisse.

57 cont.

- Es gibt eine Variante, bei dem das Ergebnis auf allen Prozessen hinterlegt wird: `MPI_Allreduce` hat die gleichen Aufrufparameter bis auf den `root` Parameter, der hier entfällt.

- Für die MAX/MINLOC Variante stellt MPI weitere vordefinierte Datentypen zur Verfügung, die aus einer Kombination eines `float/double/int`... mit einem `int` bestehen.
Das Ergebnis des MAXLOC Aufrufes ist genau das Paar, bei dem das Maximum auftrat. Analoges gilt für MINLOC.

- Über `MPI_Op_create` und `MPI_Op_free` lassen sich eigene Funktionen definieren.

58 Fehlerbehandlung

- Wie schon erwähnt bricht MPI beim Auftreten eines Fehlers automatisch ab. Dieses Verhalten läßt sich durch Setzen eines *Errorhandlers* ändern.
- Man setzt einen Errorhandler durch

```
MPI_Errhandler_set(MPI_Comm comm,
                   MPI_Errhandler errhandler)
```

  Ein existierender Handler kann mit

```
MPI_Errhandler_get(MPI_Comm comm,
                   MPI_Errhandler *errhandler)
```

  abgefragt werden.

59 cont.

- Mit dem Aufruf von

```
MPI_Errhandler_create(
             MPI_Handler_function *function,
             MPI_Errhandler *errhandler)
```

  wird eine eigene Funktion als Errorhandler angemeldet.
- `MPI_Handler_function` ist folgendermaßen definiert:

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

- Ein Errorhandler kann mit

```
MPI_Errhandler_free(
                   MPI_Errhandler *errhandler)
```

  freigegeben werden.

60 cont.

- Ein Fehlercode kann durch
```
MPI_Error_string(int errcode, char *string,
                 int *resultlen)
```
`string` muß dabei mindestens `MPI_MAX_ERROR_STRING` Zeichen fassen können. In `resultlen` wird die Länge der Fehlermeldung zurückgeliefert.

- Durch die Funktion
```
MPI_Error_class(int errorcode,
                int *errorclass)
```
ist es möglich, die Fehlercodes auf vordefinierte Klassen von Fehlern abzubilden. Diese sind im `mpi.h` Headerfile definiert.

61 Ausblick: Kommunikatoren

- Ein Kommunikator besteht eigentlich auf folgenden Komponenten:
  - ▲ Ein Kontext für die Nachrichtenübertragung.
  - ▲ Eine Gruppe von Prozessen.
  - ▲ Eine virtuelle Topologie.
  - ▲ Ein Cache von Attributen.
- Zusätzlich zu den *intra*-Kommunikatoren, die für den Nachrichtentransport innerhalb einer Gruppe von Prozessen verantwortlich sind, gibt es noch *inter*-Kommunikatoren, die den Nachrichtenaustausch zwischen verschiedenen Gruppen ermöglichen.

62 Gruppenmanagement

- MPI bietet folgende Funktionen, aus zwei Gruppen eine neue zu erzeugen:
  - ▲ `MPI_Group_union`
  - ▲ `MPI_Group_intersection`
  - ▲ `MPI_Group_difference`
- Bildung einer neuen Gruppe durch Einschränken auf einzelne Prozesse:
  - ▲ `MPI_Group_incl`
  - ▲ `MPI_Group_excl`
  - ▲ `MPI_Group_range_incl`
  - ▲ `MPI_Group_range_excl`

63 cont.

- Abfragen einer Gruppe:
  - ▲ `MPI_Group_size`
  - ▲ `MPI_Group_rank`
- Vergleich zweier Gruppen:
  - ▲ `MPI_Group_compare`
  - ▲ `MPI_Group_translate_ranks`
- Freigabe einer Gruppe:
  - ▲ `MPI_Group_free`

64 Gruppen und Kommunikatoren

- Abfragen eines Kommunikators:
  - `MPI_Comm_group`
  - `MPI_Comm_compare`
- Erzeugung eines Neuen Kommunikators
  - `MPI_Comm_dup`
  - `MPI_Comm_create`
  - `MPI_Comm_split`
- Freigabe eines Kommunikators
  - `MPI_Comm_free`

65 Caching

- MPI bietet Funktionen an, mit denen ein Wert in einem Kommunikator gespeichert werden kann.
- Der gespeicherte Wert ist prozeßlokal!
- Der Typ des Wertes ist “`void *`”.
- Diese Funktionalität wurde für die kollektive Kommunikationsroutinen eingebaut und als Interface bereitgestellt.
- Dies ist nützlich für die Programmierer numerischer Bibliotheken.

66 cont.

- Funktion zum Reservieren/Freigeben eines Speicherplatzes für den Wert:
  - ▲ `MPI_Keyval_create`
  - ▲ `MPI_Keyval_free`
- Schreiben/Lesen/Löschen des Wertes:
  - ▲ `MPI_Attr_put`
  - ▲ `MPI_Attr_get`
  - ▲ `MPI_Attr_delete`
- Beim Reservieren können Callbackfunktionen für das Kopieren und Löschen des Wertes angegeben werden.

67 Virtuelle Topologien

- Um die Adressierung einzelner Prozesse innerhalb von MPI zu erleichtern, bietet MPI die Möglichkeit, Topologien mit den Prozessen zu verknüpfen.
- MPI arbeitet mit kartesischen Topologien (d.h. Gitter) und beliebige Graphtopologien.
- Die Abbildung von den Topologien zu den Prozeßrängen sollte von den MPI Implementieren so gestaltet sein, daß die vorhandene Hardware berücksichtigt wird.

68 cont.

- Beispiel: Kartesische Koordinaten:
- Mittels

```
MPI_Cart_create(MPI_Comm comm_old,
                int ndims, int *dims,
                int *periods, int reorder,
                MPI_Comm *comm_cart)
```

kann ein Gitter der Dimension `ndims` mit `dims[i]` Prozessoren in Richtung `i` erzeugt werden.

- Später kann dann mittels

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

der Rang eines Prozesses im Gitter abgefragt werden.

69 Ausblick MPI-2

- Wurde zuerst auf der *Supercomputing '96* vorgestellt.
- Das MPI Komitee wird 3-mal im Frühling 1997 tagen.
- Der neue Standard soll im Mai 1997 veröffentlicht werden.
- MPI-2 Erweiterungen:

▲ dynamische Prozeßerzeugung,

▲ Parallele IO,

▲ Einseitige Kommunikation,

▲ C++ und Fortran-90 Bindings,

▲ Real-time extensions,

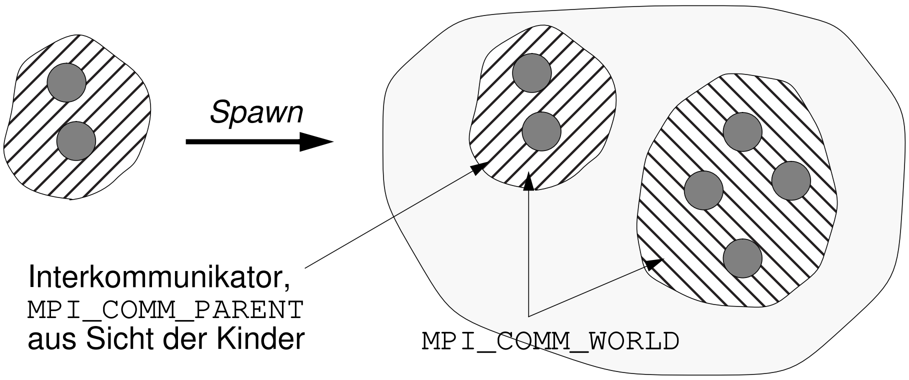
▲ und ein paar andere Kleinigkeiten.

70 Dynamische Prozeßerzeugung

- MPI-1 bietet keine Möglichkeit, Prozesse nachzustarten oder zu beenden.
- PVM Benutzer erwarten etwas wie `pvm_spawn`.
- Das Model muß weit genug ausgelegt sein, um mit Workstation Clustern, Hochleistungsparallelrechnern, Batchjob Schedulern usw. zurechtzukommen.
- MPI soll unabhängig vom Betriebssystem spezifiziert sein.
- MPI-1 Kompatibilität muß gewahrt bleiben.

71 cont.

- MPI-2 bildet die Eltern-Kinder Beziehung auf Kommunikatoren ab. Die Eltern können über einen *Interkommunikator* mit den Kindern Nachrichten austauschen.



- Über den MPI-1 Aufruf `MPI_Intercomm_merge` kann ein einzelner Kommunikator erzeugt werden.

72 cont

- Der Spawn Aufruf hat folgende Syntax:

```
MPI_Spawn(char *command, char **argv,
          int maxprocs,
          int root, MPI_Comm comm,
          MPI_Comm *intercomm)
```

  Von Prozeß `root` des Kommunikators `comm` aus werden `maxprocs` Prozesse durch das Kommando `command` gestartet.

- Es gibt zusätzlich dazu die nicht blockierende Version `MPI_Ispawn`, eine Version, bei der man mehrere Kommandos angeben kann (`MPI_Spawn_multiple`), und die Möglichkeit, nicht kommunizierende Prozesse zu starten `MPI_Spawn_independent`.

73 cont.

- Ein Prozeß kann sich mit

```
Mpi_Comm_disconnect(MPI_Comm comm)
```

  aus einen Kommunikator ausklinken.

- Signale sind an einzelne Prozesse sendbar:

```
MPI_Signal(MPI_Group group, int rank,
           int signal)
```

- Der Ablauf eines Prozesses läßt sich über

```
MPI_Process_monitor(MPI_Group group, int rank,
                    int event,
                    MPI_Request *request)
```

  überwachen (im Moment existiert als `event` nur `MPI_PROCESS_DIED`).

74 cont.

- Zwei verschiedene MPI-2 Prozesse können miteinander Verbindung aufnehmen. Dies geschieht über ein *Socket* ähnliches Interface:
  - MPI_Port_open
  - MPI_Accept
  - MPI_Connect
- Ports können einem *Portmapper* bekanntgegeben werden:
  - MPI_Name_publish
  - MPI_Name_get

75 Parallele IO

- UNIX ähnlich Schnittstelle.
- Gleichzeitiger Zugriff von allen Prozessen (`MPI_Read_all`).
- Beliebige Datenpartitionierung möglich, jeder Prozeß hat die Sicht auf seinen speziellen Teil des Files.
- Unterstützung von Scatter/Gather Zugriffen.
- Lese/Schreiboptimierungen durch die Angabe von *Hints* möglich.
- Die Lese/Schreibaufrufe gibt es alle auch in der nicht-blockierenden Version (`MPI_Iread`).