Einführung in die Message-Passing Bibliothek

MPI

A) Einführung - Referenzen -

- Dank für Anregungen:
  - M. Schröder
  - Training and Education Centre at Edinburgh Parallel Computing Centre (EPCC-TEC), University of Edinburgh, United Kingdom. (http://www.lrz-muenchen.de/services/software/parallel/mpi/epcc-course/)
- MPI Standards:

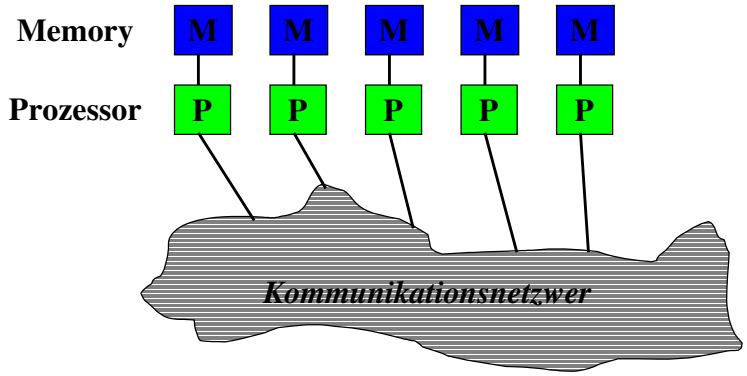
  http://www.mcs.anl.gov/mpi

A) Einführung - Konventionen -

- Auswahl der wichtigsten MPI-Routinen
  (Vollständiger Überblick: siehe Referenz)
- Diskussion der Aufrufe in C und FORTRAN
- Hardwareunabhängige Beschreibung
- Diskussion aus Sicht des Programmierers
- Prozess = Prozessor

A) Einführung - MP Programmierparadigma (1) -

Das Message-Passing (MP) Programmierparadigma (schematisch):



A) Einführung
- MP Programmierparadigma (2) -

- Spezifikation des MP Programmierparadigmas:

  Jeder Prozessor eines MP Programmes führt ein Unterprogramm mit folgenden Eigenschaften aus:

  1) Das Unterprogramm ist in einer sequentiellen Programmiersprache verfaßt.

  2) Alle Variablen sind lokal (privat).

  3) Die Kommunikation mit anderen Prozessoren geschieht via spezieller Subroutinen

- Einschränkung: MP oft nur in Verbindung mit SPMD Modell möglich.

A) Einführung
- Das SPMD Modell -

- *Single Program, Multiple Data*
- Das gleiche Programm läuft auf allen Prozessoren.

  ⇒ Einschränkung des allgemeinen MP Modells

- Allgemeines MP Modell kann emuliert werden:

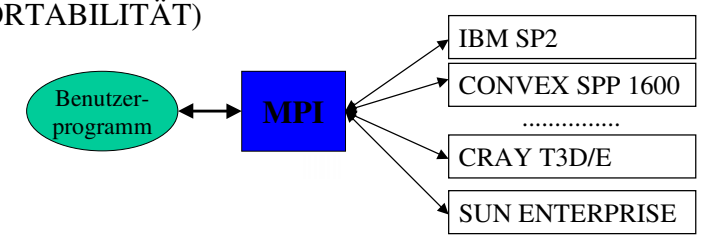
```
main(){
  if (process is master){
      master( /*Arguments */);}
  else{
      slave( /*Arguments */);}
}
```


A) Einführung
- MPI Forum -

- *Message-Passing Interface (MPI)* Forum*:* Definition eines MP Standards: **MPI**
- Beginn: April 1992
- Teilnehmer (ca. 60 weltweit)
  - Nutzer
  - Forschungseinrichtungen
  - Hersteller von Hard- und Software
- Aktuelle Standards:
  - MPI 1.0: 5. Mai 1994
  - MPI 1.1: 12. Juni 1995
  - MPI 1.2: 18. Juli 1997
  - MPI 2.0: 18. Juli 1997

A) Einführung
- MPI : Ziele -

- Architektur- und Herstellerunabhängiger Sourcecode (PORTABILITÄT)



- Einfaches Interface für FORTRAN und C
- Bereitstellung sicherer und einfacher Datenübertragung
- Basis für parallele numerische Bibliotheken
- Einsatz auch in heterogenen Umgebungen

A) Einführung
- Mängel des MPI-1 Standards -

- Im MPI-1 Standard fehlen:
  - Dynamische Prozeßverwaltung
  - *Shared memory* Operationen
  - Erzeugung und Verwaltung von *Threads*
  - Paralleler IO
  - *Debugging* Hilfen
- MPI-2 behebt viele dieser Mängel
- Aber: Z. Zt. existieren nur wenige MPI-2 Implementationen

B) MPI Programme: Grundlagen
- MPI Schnittstelle (1) -

- *Header* File:
  - C: #include <mpi.h>
  - FORTRAN: include `mpif.h´
- Aufrufsyntax:
  - C: error = MPI_Xxxx(parameter,.....);
    [bzw. MPI_Xxxx(parameter,.....);]
  - FORTRAN: call MPI_XXXX(parameter,...,ierror)
- Arraykonventionen:
  - C: Arrays starten bei 0
  - FORTRAN: Arrays starten bei 1

B) MPI Programme: Grundlagen
- MPI Schnittstelle (2) -

- Initialisieren des MPI Prozesses:
  - C: MPI_Init(&argc, &argv);
  - FORTRAN: call MPI_INIT(ierror)

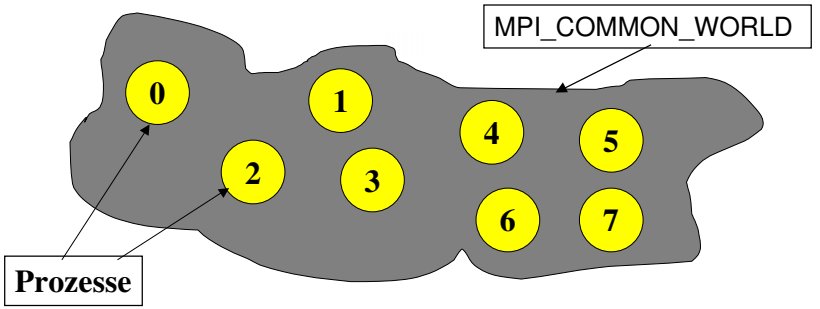
Immer am Beginn des Programmes!

- Terminieren des MPI Prozesses:
  - C: MPI_Finalize();
  - FORTRAN: call MPI_FINALIZE(ierror)

Immer am Ende des Programmes!

B) MPI Programme: Grundlagen
- Kommunikator MPI_COMMON_WORLD -

- Kommunikation zwischen MPI Prozessen erfolgt über *Kommunikatoren*.
- Der Kommunikator MPI_COMMON_WORL umfaßt alle MPI Prozesse:



B) MPI Programme: Grundlagen - Handles -

- Kommunikatoren (z. B. MPI_COMM_WORLD) sind als sog. *Handles* implementiert
- MPI kontrolliert mit *Handles* seine internen Datenstrukturen
- Benutzer kann auf *Handles* zugreifen:
  - C: Definition über spezielle typedef
  - FORTRAN: INTEGER
- Definition von MPI_COMM_WORLD in mpi.h bzw. mpif.h

B) MPI Programme: Grundlagen - Rang eines Prozesses -

- Eindeutige Identifikation des *Ranges* eines Prozesses (innerhalb des Kommunikators MPI_COMM_WORLD):
  - C:

```
int rank;
..
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

  - FORTRAN:

```
INTEGER RANK , IERROR
..
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERROR)
```

- Rank = 0,1,2,.....

B) MPI Programme: Grundlagen - Anzahl der Prozesse (size) -

- Bestimmung der Anzahl der beteiligten Prozesse (innerhalb des Kommunikators MPI_COMM_WORLD):
  - C:

```
int size;
..
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

  - FORTRAN:

```
INTEGER SIZE, IERROR
..
CALL MPI_COMM_RANK(MPI_COMM_WORLD,SIZE, IERROR)
```


B) MPI Programme: Grundlagen - Minimales MPI Programm -

```
#include ´mpi.h´
main(argc, argv)
int argc;
char **argv[];
{
      int rank , size;
      MPI_Init( &argc , &argv );

      MPI_Comm_size( MPI_COMM_WORLD , &size );
      MPI_Comm_rank( MPI_COMM_WORLD , &rank );

      MPI_Finalize();
}
```

B) MPI Programme: Grundlagen - Compilieren und Starten -

- Compilieren und Starten von MPI Programmen sind vom jeweiligen Rechner abhängig
- Für die CONVEX SPP1600 am RRZE gilt:

Compilieren (von test.c):

```
$/usr/convex/bin/mpicc testc -o testx
```

Starten (auf 8 Prozessoren):

```
$./testx -np 8
```


C) MPI Programme: Kommunikation - Nachrichten -

- Kommunikation zwischen Prozessen:
  Senden und Empfangen von Nachrichten
- Nachricht:
  Anzahl von Elemente des gleichen MPI-Datentyps
- Gleicher Datentyp bei Senden und Empfangen !
- MPI-Datentypen:
  - Vordefinierte Datentypen
  - Abgeleitete Datentypen
- Abgeleitete Datentypen werden aus den vordefinierten Typen aufgebaut

C) MPI Programme: Kommunikation - Vordefinierte Datentypen in C -

- MPI unterstützt folgende C Datentypen

MPI Datentyp	*C Datentyp*
MPI_CHAR	signed char
MPI_SHORT	singed short
MPI_INT	singed int
MPI_LONG	singed 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	

C) MPI Programme: Kommunikation -Vordefinierte Datentypen in FORTRAN-

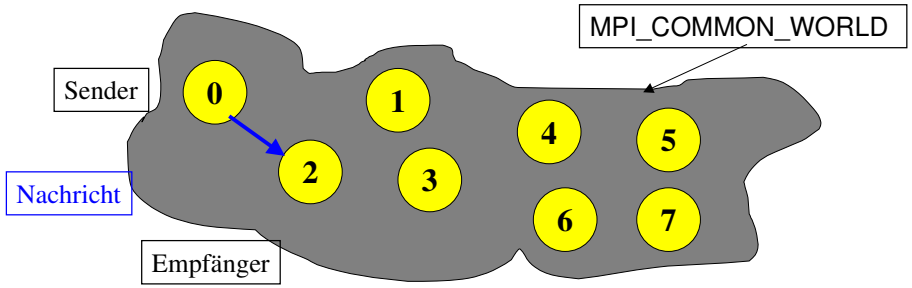
- MPI unterstützt folgende FORTRAN Datentypen

MPI Datentyp	*C Datentyp*
MPI_CHAR	CHARACTER(1)
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_BYTE	
MPI_PACKED	

- Hinweis: Vorsicht mit REAL / DOUBLE_PRECISION bei CRAY Maschinen

C) MPI Programme: Kommunikation - Point to Point Kommunikation -

- Kommunikation zwischen zwei Prozessen



- Kommunikation nur innerhalb des selben Kommunikators
- Identifizierung des Empfängers anhand seines Ranges im Kommunikator

C) MPI Programme: Kommunikation - Senden / Empfangen: Standard (1) -

- Senden einer Nachricht:

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

- Sende count Elemente des MPI-Datentyps datatype an den Prozeß mit Rang dest im Kommunikator comm !
- Typisierung der Nachricht durch tag Parameter
- [FORTRAN:

```
<Type> sendbuf
INTEGER count, datatype, dest , tag, comm, ierror
call MPI_Send(sendbuf, count, datatype, dest , tag, comm, ierror); ]
```


C) MPI Programme: Kommunikation - Senden / Empfangen: Standard (2) -

- Empfangen einer Nachricht:

```
MPI_Recv(void *recvbuf, int count,
         MPI_Datatype datatype,
         int source , int tag ,MPI_Comm comm,
         MPI_Status *recvstatus);
```

- Empfange count Elemente des MPI-Datentyps datatype vom Prozeß mit Rang source im Kommunikator comm und schreibe sie nach recvbuf
- *Wildcards*: Empfange Nachrichten mit beliebigem Senderrang [Sendertag]:

```
source = MPI_ANY_SOURCE
[ tag = MPI_ANY_TAG ]
```


C) MPI Programme: Kommunikation - Senden / Empfangen: Standard (3) -

Statusobjekt (recvstatus)

- Detaillierte Informationen über eine abgeschlossene MPI_Recv Operation.
- MPI_Status recvstatus;
- C-Struktur mit öffentlichen Elementen:
  - recvstatus.MPI_SOURCE
  - recvstatus.MPI_TAG
  - recvstatus.MPI_ERROR
- [FORTRAN: INTEGER recvstatus(MPI_STATUS_SIZE)

```
source    = recvstatus(MPI_SOURCE)
tag       = recvstatus(MPI_TAG)
ierror    = recvstatus(MPI_ERROR)]
```

C) MPI Programme: Kommunikation - Blockierende Kommunikation (1) -

- *Blockierender Aufruf :*

  Modifikation von sendbuf / recvbuf nach erfolgreicher Terminierung des Aufrufs möglich.
- MPI_Recv: Stoppen des Programmablaufs bis recvbuf vollständig empfangen wurde.
- MPI_Send:Gemäß MPI-Standard:

  „*Synchronous send*“ **oder** „*Buffered send*“
- *Synchronous send*: Send / Recv arbeiten synchron
- *Buffered send:* 1) sendbuf wird in lokalen Sendpuffer kopiert

  2) MPI_Send terminiert erfolgreich

  3) MPI_Recv erhält Nachricht aus Sendpuffer

C) MPI Programme: Kommunikation - Blockierende Kommunikation (2) -

Sendmodi : Übersicht

- Standard send MPI_Send();
- *Synchronous send* MPI_Ssend();
- *Buffered send* MPI_Bsend();

[Sender muß Puffer zugeweisen und wieder entfernen:

MPI_Buffer_attach(void *buffer , int size);

MPI_Buffer_dettach(void *buffer , int size);]

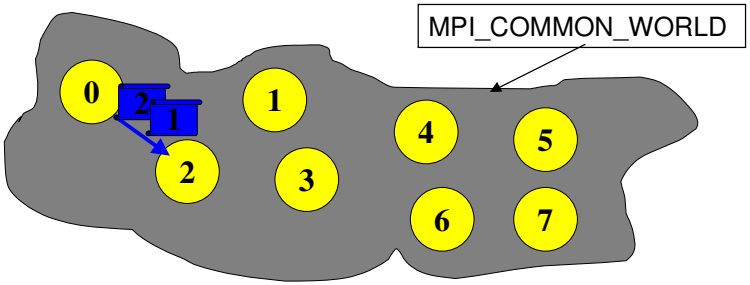
- *Ready send* MPI_Rsend();

  Aufruf terminiert immer!

C) MPI Programme: Kommunikation - Blockierende Kommunikation (3) -

Kommunikation ist Ordnungserhaltend

Nachrichten können sich nicht überholen!



(Gilt auch für nicht-synchrone Send Operationen)

C) MPI Programme: Kommunikation - Blockierende Kommunikation (4) -

Probleme blockierender Aufrufe

- Keine Überlappung von Kommunikation und Rechnung

  ⇒ Zum Teil erhebliche Perfomanceeinbußen
- Puffern der Nachrichten (MPI_Bsend):

  ⇒ Zusätzliche Kopieroperationen und u. U. erheblicher zusätzlicher Speicheraufwand.
- *Deadlock* Situationen.

Nicht-blockierende Aufrufe

- Ablauf:

  + Send/Recv Operation wird nur initialisiert

  + Aufruf terminiert sofort und liefert *Request-handle* zurück

  + /* WORK */

  + Via *Request-handle* kann die initialisierte Operation jederzeit auf erfolgreichen Abschluß geprüft werden.

Syntax

- C:

```
MPI_Isend(void *sendbuf , int count , MPI_Datatype datatype,
      int dest , int tag , MPI_Comm comm ,
      MPI_Request *reques )
```

- FORTRAN:

```
MPI_Isend(sendbuf , count , datatype,
      dest , tag , comm ,
      request)
```

- *Request-handle:* MPI_Request request;
- Analog: MPI_Issend, MPI_Ibsend, MPI_Irsend, MPI_Irecv.

Abschluß einer nicht-blockierenden Kommunikation

- Warten, bis Operation abgeschlossen ist:

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

- Test, ob Operation abgeschlossen ist:

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

- Aufrufparameter:
  - MPI_Request request; /* *Request-handle* */
  - MPI_Status status; /* Statusinformation (siehe MPI_Recv) */
  - int flag; /* Test erfolgreich ? */

Beispiel

```
MPI_Request request;
MPI_Status status;
int rank , dest  , elem;  /* Rang des Senders, Empfägers , Anz. Elem. */
int sendbuf [100000];
....
If(rank == 0){
    dest = 2;
    elem = 100000;
    MPI_Isend( &sendbuf , elem , MPI_INT ,  dest , tag ,
          MPI_COMM_WORLD, &request);
}
  /* Berechnungen: Hier darf sendbuf nicht verändert werden  */
If(rank == 0){
    MPI_Wait( &request , &status);   /* Warte bis MPI_Isend abgeschl. */
    sendbuf[0] = 1;}                  /* und verändere erst dann sendbuf */
.....
```

Varianten von MPI_Wait und MPI_Test

- Gleichzeitiger Test mehrerer *Request-handles* mit:

```
MPI_Waitany();      /* bzw. Testany(); */
MPI_Waitall();      /* bzw. Testall(); */
MPI_Waitsome();     /* bzw. Testsome(); */
```

- any bezieht sich auf ein, all auf alle und some auf soviel wie möglich requests!

Abbruch eines laufenden Aufrufs

- Aufforderung, den Aufruf mit dem *Request-handle* request abzubrechen:

```
MPI_Cancel(MPI_Request *request );
```

- Anschließend sollte der zugehörige *Request-handle* wieder freigegeben werden:

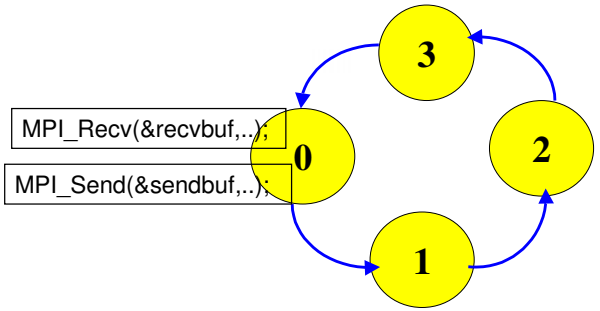
```
MPI_Request_Free(MPI_Request *request );
```

- Test, ob Aufruf mit dem *Request-handle* request abgebrochen wurde:

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


Problemstellung

- Zyklische Send/Recv Aufrufe ⟺ *Deadlock*



Syntax

- C: 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)

Vereinfachung: MPI_Sendrecv_replace

- sendbuf wird abgeschickt und durch recvbu ersetzt!
- recvbuf, recvcount, recvtype entfallen im Aufruf !

D) Kollektive Kommunikation - Überblick -

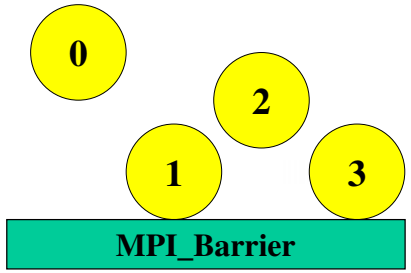
- Kollektive Kommunikation:

  Nachrichtenaustausch zwischen allen Prozessen eines Kommunikators

- Gleicher Funktionsaufruf aller Prozesse eines Kommunikators
- Nur blockierende Aufrufe möglich !
- Arten kollektiver Kommunikation
  - Synchronisation
  - Broadcast
  - Gather / Scatter
  - Aufruf kollektiv operierender vordefinierter **und** benutzerdefinierter Funktionen (z.B. max)

D) Kollektive Kommunikation - Synchronisation -

Synchronisation aller Prozesse eines Kommunikators

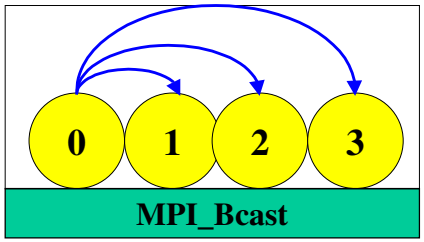


- Syntax:

  C: MPI_Barrier(MPI_Comm comm)

D) Kollektive Kommunikation - Broadcast -

Nachricht an alle Prozesse eines Kommunikators



- Syntax:

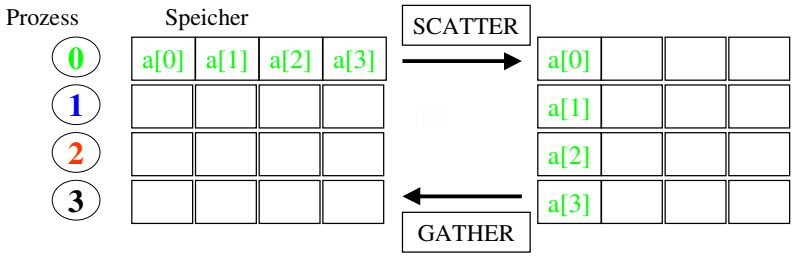
  C: MPI_Bcast(void*buffer, int count,
  MPI_Datatype, int root, MPI_Comm comm)

  (Beispiel: root = 0)

D) Kollektive Kommunikation - Gather / Scatter (1) -

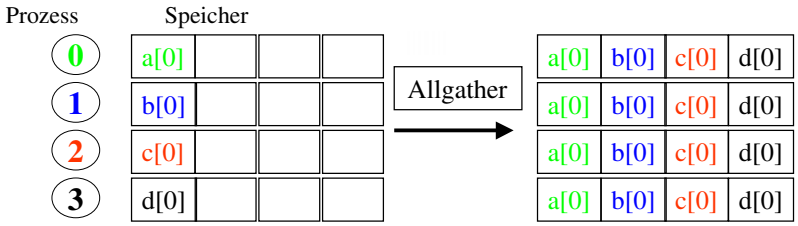
Gather / Scatter Grundoperationen

zwischen allen Prozesse eines Kommunikators

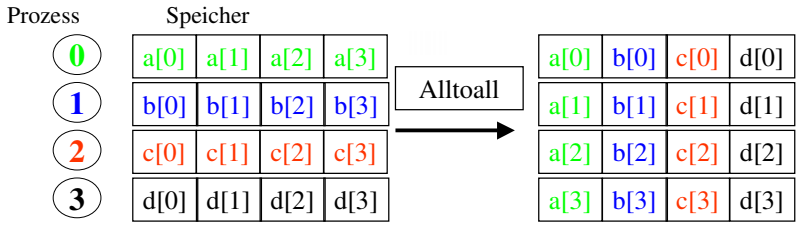


- Scatter: *Verteilung der Daten des root Prozesses auf alle Prozesse*
- Gather: Sammeln *der Daten aller Prozesse auf dem root Prozess*

Erweiterungen der Gather / Scatter Operationen
zwischen allen Prozesse eines Kommunikators:
ALLGATHER



Erweiterungen der Gather / Scatter Operationen
zwischen allen Prozesse eines Kommunikators:
ALLTOALL



Gather / Scatter Syntax

- C: MPI_Gather(void *sendbuf, int sendcount,
  MPI_Datatype sendtype,
  void *recvbuf, int recvcount,
  MPI_Datatype recvtype,
  int root, MPI_Comm comm)
- Analog: Aufruf von MPI_Scatter
- Ohne root, ansonsten analog: MPI_Allgather und MPI_Alltoall

- Globale Operation:

  Eine Operation wird auf alle Prozesse eines Kommunikators angewandt.

- Beispiel: Globales Maximum einer Variablen, die auf jedem Prozess einen unterschiedlichen Wert haben kann.
- MPI stellt 12 vordefinierte Operationen bereit
- Definition benutzereigener Operationen mit MPI_Op_create und MPI_Op_free möglich.

E) Globale Operationen - Vordefinierte Operationen -

Name	Operation	Name	Operation
MPI_SUM	**Summe**	**MPI_PROD**	**Produkt**
MPI_MAX	**Maximum**	**MPI_MIN**	**Minimum**
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+ Possition**	**MPI_MINLOC**	**Minimum+ Possition**

E) Globale Operationen - Varianten -

Ergebnis nur auf root Prozess

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

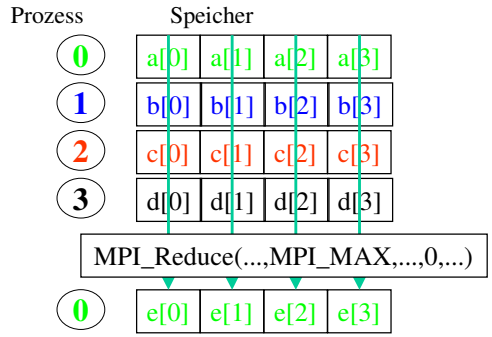
- Ergebnis in recvbuf auf root. Alle anderen Prozesse ignorieren recvbuf.
- Für count > 1: Parallele Anwendung auf alle count Elemente

Ergebnis auf allen Prozessen

- MPI_Allreduce: root Parameter entfällt!

E) Globale Operationen - Beispiel -

Berechne e(i) = max{ a[i], b[i], c[i], d[i] }
jeweils für i=0,1,2,3



F) Abgeleitete Datentypen - Übersicht -

- Abgeleiteter Datentyp :

  Definition einer „Struktur“ von Daten auf die in Kommunikationsoperationen verwendet werden kann.

- Ähnlich einer Struktur in C
- Verwendung: Alle MPI-Aufrufe, die einen Datentyp benötigen.
- Nur Austausch von Nachrichten gleichen Datentyp

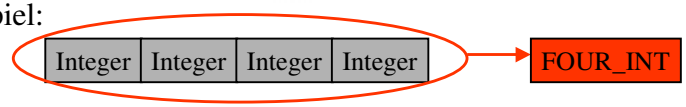
➡ Datentypen auf allen MPI Prozessen gleich setzen !

F) Abgeleitete Datentypen - Zusammenhängende Daten -

- Zusammenfassen von count Elementen eines festen Datentyps zu einem neuen Datentyp:

```
MPI_Type_contiguous(  int coun ,
                      MPI_Datatype oldtype,
                      MPI_Datatype *newtype)
```

- Beispiel:




```
MPI_Type_contiguous(  4, MPI_Int, MPI_Datatype *FOUR_INT);
```


F) Abgeleitete Datentypen - Initialisierung / Freigabe -

- Nach der Definition des neuen Datentyps muß er zusätzlich erlaubt werden:

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

- Freigabe den neues Datentyp:

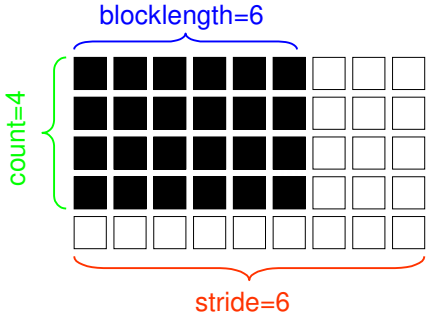
```
MPI_Type_free(MPI_Datatype *datatype)
```

- Beispiel:

```
MPI_Datatype Four_int;
.....
MPI_Type_contiguous(4, MPI_Int,  &Four_int );
MPI_Type_commit( &Four_int );
....
MPI_Type_free( &Four_int );
```


F) Abgeleitete Datentypen - Vektorstrukturen (1) -

- Beispiel: Block aus Matrix extrahieren




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


F) Abgeleitete Datentypen - Vektorstrukturen (2) -

Erweiterungen der MPI_Type_vector Operation

- Angabe des Strides (stride) in Bytes:

```
MPI_Type_hvector(...)
```

- Variable Blocklänge (blocklength) und Stride (stride) :

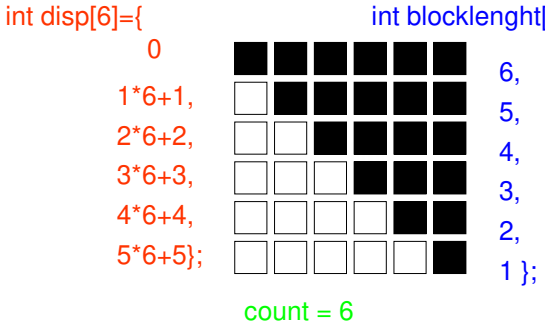
```
MPI_Type_indexed(  int count,
                   int *array_of_blocklength,
                   int *array_of_displacements,
                   MPI_Datatype oldtype,
                   MPI_Datatype *newtype)
```

  array_of_blocklength und array_of_displacements Felder der Länge count.

- MPI_Type_hindexed: Displacements in Bytes.

F) Abgeleitete Datentypen - Vektorstrukturen (3) -

- Beispiel: Extrahieren der oberen Dreiecksmatrix



MPI_Type_indexed(count , &blocklength , &disp , MPI_DOUBLE , &**newtype**);

F) Abgeleitete Datentypen - Allgemeine Strukturen (1) -

Erweiterung auf allgemeine Datenstrukturen

- Blöcke unterschiedlicher Datentyps und Länge zu einer Struktur zusammenfassen:

```
MPI_Type_struct(    int count,
                    int *array_of_blocklengths,
                    MPI_Aint *array_of_displments,
                    MPI_Datatypes *array_of_types,
                    MPI_Datatype *newtypes )
```

- Displacements: Offsets in Bytes, da kein gemeinsamer Datentyp zugrunde liegt.

F) Abgeleitete Datentypen - Allgemeine Strukturen (2) -

Problem: Berechnung der Displacements

- Hardewareabhängige *alignment restriction*:
  Beispiel: int (double) auf durch 32 (64) teilbaren Adressen.
  Zugriff mit falschem *alignment* ➡ *Bus Error*

- *Padding* in C Strukturen:
  Compiler berücksichtigt *alignment restriction* dadurch, daß er Löcher in die Struktur einfügt und die Struktur selbst nur an bestimmten Adressen plaziert.

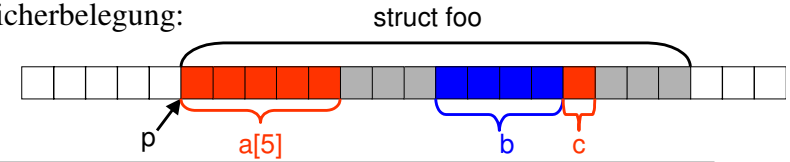
F) Abgeleitete Datentypen - Allgemeine Strukturen (3) -

Beispiel: *Padding*

- Definition einer Struktur

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

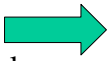
- Speicherbelegung:



F) Abgeleitete Datentypen - Allgemeine Strukturen (4) -

Bestimmen der Displacements via MPI (1)

- Bestimmung absoluter Adressen weder in FORTRAN noch in ANSI-C möglich!



- Berechnung der Displacements via

  MPI_Address(void *location, MPI_Aint *address)

- Beispiel foo:

```
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];
```


F) Abgeleitete Datentypen - Allgemeine Strukturen (5) -

Bestimmen der Displacements via MPI (2)

- Definition des neuen Datentyps:

```
MPI_Datatype foo_type;
MPI_Datatype type[3] = {MPI_CHAR , MPI_INT, MPI_CHAR}
int blockl[3] = { 5 , 1 , 1 };
.....
/* Berechnung der displ: siehe Folie -1 *

MPI_Type_struct( 3 , &blockl , &displ , &type , &foo_type );
```


F) Abgeleitete Datentypen - Allgemeine Strukturen (6) -

Bestimmen der Displacements via ANSI-C Makro

- ANSI-C spezifiziert Makro offsettof(type, member) in <stddef.h>.

- Berechnung der displ mit offsettof:

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


G) Benutzerdef. Kommunikatoren - Übersicht -

- Bisher: Kommunikator MPI_COMM_WORLD
  - Vordefinierter Kommunikator
  - Umfaßt **alle** initialisierten MPI Prozesse
  - Globale Operationen über diesen Kommunikator

- Operationen auf einer fest definierten Teilmenge von MPI Prozessen:
  - Definiere neuen Kommunikator für diese Teilmenge der MPI Prozesse (=Prozessgruppe).
  - Über neuen Kommunikator: Globale Operationen auf dieser Teilmenge der MPI Prozesse.

G) Benutzerdef. Kommunikatoren - Erstellen eines Kommunikators (1) -

Syntax der grundlegenden Aufrufe

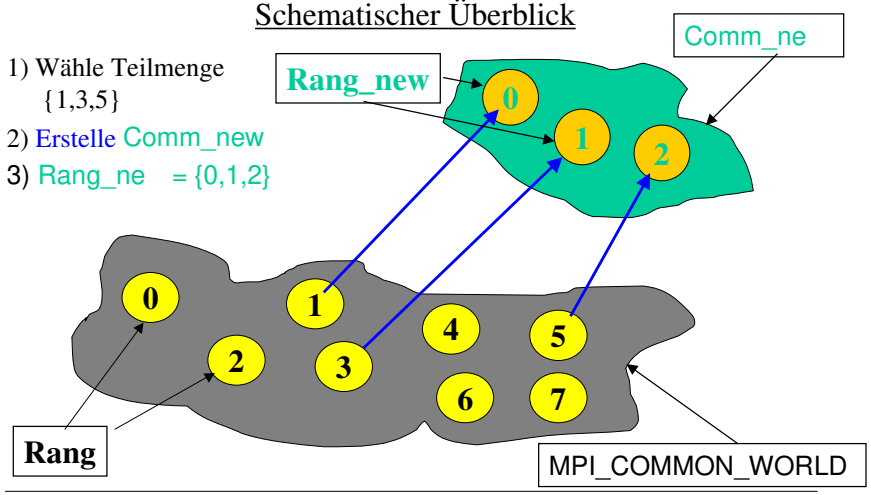
- Bestimme Prozessgruppe eine Kommunikator :
  MPI_Comm_group(MPI_Comm comm , MPI_Group *group);

- Bilde aus bestehender Gruppe (group) eine neue Prozessgruppe die n Prozesse (array_rank) enthält:
  MPI_Group_incl(MPI_Group group , int n , int *array_ranks, MPI_Group *newgroup);

- Erstelle neuen Kommunikator für die Gruppe group:
  MPI_Comm_create(MPI_Comm comm , MPI_Group group , MPI_Comm *newcomm);

G) Benutzerdef. Kommunikatoren - Erstellen eines Kommunikators (2) -

Schematischer Überblick



G) Benutzerdef. Kommunikatoren - Erstellen eines Kommunikators (3) -

Programmablauf

```
MPI_Comm Comm_ne   ;          /* Neuer Kommunikator */
MPI_Group world , Group_ne  ;  /* Neue Gruppe */
int ranks[3] = {1,3,5};       /* Prozesse, die */
                              /* Comm_ne   angehören*/
.....
MPI_Comm_group( MPI_COMM_WORLD , &world);

/* Bilde neue Gruppe aus den Prozessen in ranks */
MPI_Group_incl( world , 3 , &ranks , &Group_new );

/* Erstelle neuen Kommunikator auf Group_new  */
MPI_Comm_create( MPI_COMM_WORLD , Group_ne   ,&Comm_new);
```


G) Benutzerdef. Kommunikatoren - Zusammenfassung -

- Kommunikatoren ermöglichen die Erstellung paralleler Bibliotheken

- Bisher: *intra* Kommunikatoren
  Nachrichtentransport innerhalb einer Gruppe von Prozessen

- Erweiterung: *inter* Kommunikatoren
  Nachrichtentransport zwischen Prozessorgruppen

H) Virtuelle Topologien - Übersicht -

- Verknüpfung von Prozessen mit Topologien
  Beispiel: Anordnung der Prozeese als Ring
- MPI verwendet kartesische Topologien (Gitter) und beliebige Graphtopologien
- Abbildung zwischen Topologien und Prozeßrängen:
  Berücksichtigung der Hardewarespezifikationen bei der MPI Implementierung.

H) Virtuelle Topologien - Erstellen einer Prozessortopologie -

Kartesische Koordinaten

- Gitter der Dimension ndims mit dims[i] Prozessen in i Richtung:

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

periods[i] = true / false falls periodische / offene Randbedingungen in i Richtung.
reorder = true erlaubt beliebige Zuordnung zwischen Prozessen und Gitterpunkten

- Bestimmung des Prozeßranges via Koordinaten:
  MPI_Cart_rank(MPI_Comm comm , int *coords , int *rank);

H) Virtuelle Topologien - Beispiel -

1D Prozeßring

```
MPI_Comm ring_comm;
int ndims , dims[3] , periods[3] , koordinate[3] , reorder;
int other_id;
...
ndims=1;
dims[0]=numprocs;
periods[0]=1;
reorder=1;
MPI_Cart_create( MPI_COMM_WORLD , ndims , &dims[0] ,
    &periods[0] , reorder , &ring_comm );

koordinate[0]= myid + numprocs + 1;
MPI_Cart_rank( ring_comm , &koordinate[0] , &other_id);
```


I) Fehlerabfragen - Übersicht (1) -

- Jeder MPI Aufrruf liefert einen errcode zurück:
  errcode = MPI_Xxxx(...);
- Erfolgreiche Terminierung:
  errcode = MPI_Success

- Tritt jedoch ein Fehler auf bricht MPI sofort ab.
- Fehleranalyse Abbruch verhindern:
  MPI_Errhandler_set(MPI_Comm , MPI_Errhandler errhandler);
  mit errhandler = MPI_ERRORS_RETURN
- Test des Wertes des errhandlers:
  MPI_Errhandler_get(MPI_Comm , MPI_Errhandler errhandler);

I) Fehlerabfragen - Übersicht (2) -

- Ausgabe eines Fehlercodes:

  MPI_Err_string(int errcode , char *string , int *resultlen);

  – string muß min. MPI_MAX_ERROR_STRING Zeichen fassen

  – Länge der Meldung: resultlen
- Abbildung von Fehlercodes auf Fehlerklassen:

  MPI_Err_class(int errcode , int *errclass);

  – Definition der errclass im mpi.h File.

- Kollektiver Abbruch aller Prozesse:

  MPI_Abort(MPI_Comm comm , int errcode);

I) Fehlerabfragen - Benutzerdefinierter Abbruch -

- Benutzerdefinierte Funktion als errhandler:

  MPI_Errhandler_create(MPI_Handler_function *function,
  MPI_Errhandler errhandler);

- Definition von MPI_Handler_function :

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

- Freigeben des Handles:

  MPI_Errhandler_free(MPI_Errhandler errhandler);

J) Zeitmessung - MPI_Wtime -

- Messung der *Wallclock Time* eines Prozesses:

  double MPI_Wtime(void)

  Der Aufruf liefert die Zeit seit Start des MPI Prozesses in Sekunden.
- Timing von Programmteilen: Differenz zwischen zwei Aufrufen
- Vorsicht: Falls mehrere Prozesse auf einem Prozessor !

- Bestimmung der Auflösung des Timers:

  double MPI_Wtick(void)

K) MPI-2 Der neue Standard

Ein kurzer Überblick

K) MPI-2
- Überblick -

- Vorstellung von MPI-2 auf der *Supercomputing ´96*
- Veröffentlichung des Standards: 18. Juli 1997

- MPI-2 bleibt mit den MPI-1 Standards kompatibel
- Wichtige Erweiterungen:
  - Dynamische Prozeßerzeugung
  - Paralleler IO
  - Einseitige Kommunikation
  - C++ / FORTRAN90 Bindings

K) MPI-2
- Dynamische Prozeßerzeugung -

- Spawn Aufruf:
  MPI_Comm_Spawn(...);

- Verbindung zweier verschiedener MPI-2 Prozesse via Interfaces (MPI_Open_port, MPI_Close_port,...)

- Portmapper verwaltet Ports (MPI_Publish_name , MPI_Unpublish_name , MPI_Lookup_name)

K) MPI-2
- Paralleler IO -

- UNIX ähnliche Schnittstelle
- Gleichzeitiger Zugriff aller Prozesse auf ein File (MPI_File_read_all)
- Beliebige Datenpartitionierung möglich.
- Scatter / Gather Zugriffe
- Read / Write Zugriff auch nicht-blockierend möglich (MPI_File_iread)

K) MPI-2
- Einseitige Kommunikation -

- *Remote Memory Access* (RMA) möglich
- Ein Prozeß kann „direkt“ in den Speicher eines anderen Prozesses greifen.
- Zugriff auf den benötigten Speicherbereich muß erlaubt sein. (*Windows* (!))
- Zugriffsarten: MPI_Get, MPI_Put, MPI_Accumulate
- Synchronisation möglich

K) MPI-2
- Zusammenfassung -

- Implementierung des Standards bereitet bisher Problem
- MPI-2 am Siemens Fujitsu VPP700 des Leibniz Rechenzentrums verfügbar
- Einführung in MPI-2:

LRZ Muenchen, Mittwoch 21. April 1999 , 8:30 - 17:30 Uhr

http://www.lrz-muenchen.de/services/compute/aktuell/ali0179/

MPI Programme: Grundlagen
- Übung: Hell World-

- Übungsaufgabe:
  - Erstellen Sie ein MPI Programm, das „**hello world**" sowie den Rang des zugehörigen Prozesses und die Gesamtzahl der Prozesse ausgibt.
  - Compilieren Sie das Programm.
  - Führen Sie Läufe mit verschiedenen Anzahlen von Prozessoren durch.

MPI Programme: Grundlagen
- Übung: Kommunikation-

- Übungsaufgabe:
  - Erstellen Sie ein MPI Programm, in dem jeder Prozess seinen Rang an den Prozeß mit Rang+1 schickt.
  - Verwenden Sie die Send/Recv Aufrufe
  - Erweitern Sie das Programm so, daß der Rang wieder den empfangenen Wert wieder weiter schickt und von jedem Prozeß jeder Rang einmal empfangen und wieder verschickt wurde
  - Verwenden Sie nun MPI_Sendrecv