# Einführung in MPI

## Eine Vorlesung mit Beispielen

Luis Kornblueh

April 17, 2015

- Im Laufe dieser Einführung werden wir einfache MPI parallele Programme schreiben und darüber hinaus einige Feinheiten von MPI kennen lernen.
- Zu jedem der Themen gibt es einfache Beispiele mit Lösungen.
- Die Themen dieser Vorlesung sollen den Einstieg in die grundlegenden Konzepte von MPI erleichtern.

# Inhalt

- Was ist MPI?
- Wie lasse ich ein MPI Program laufen?
- Wie sieht ein einfaches MPI Programm aus?
- Die grundlegenden MPI Funktionen.
- Weitere MPI Funktionen und deren Eigenschaften.

# Was ist MPI?

MPI ist eine Bibliothek von Unterprogrammen und Funktionen zur Handhabung der Kommunikation und Synchronization von Programmen auf parallelen Computern.

- MPI zielt auf Systeme mit verteiltem Speicher.
- MPI ist portierbar.
- MPI Programme folgen im Allgemeinen dem *single program multiple data* (SPMD) Programmiermodell.

# Wie lasse ich ein MPI Programm laufen?

Angenommen wir haben ein MPI Programm `my_mpi_program.f90` in dem MPI initialisiert und benutzt wird:

- Kompiliert wird mit `mpif90` (einem Wrapper für Compiler und MPI Bibliotheken):

```
mpif90 -o my_mpi_program my_mpi_program.f90
```

- Das Programm können wir mit

```
mpiexec -np m ./my_mpi_program
```

  auf `m` Prozessoren laufen lassen.

# Wie läuft ein Programm parallel?

MPI started identische Kopien des Programms `my_mpi_program` auf jedem der angeforderten m Prozessoren.

Ist m = 4, dann laufen die vier identischen Programme auf jeweils einem von vier Prozessoren gleichzeitig!

```
my_mpi_program ...
```

```
my_mpi_program ...
```

```
my_mpi_program ...
```

```
my_mpi_program ...
```

# Beispiel 1: Hello World

```
program example1

  use mpi

  implicit none

  integer :: ierr, irank, isize, irc

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, irank, ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, isize, ierr)

  write(*,'(a,i2,a,i2)')                    &
       &  'Hello, world, I am ', irank, &
       &  ' of ', isize

  call MPI_Finalize(irc)

end program example1
```

# Übung 1: Laufenlassen

1. Kopiere Beispiel 1 von `~m214089/mpi/example1.f90` in dein Übungsdirectory.
2. Übersetze und lasse das Beispiel auf vier Prozessoren laufen.

Ausgabe des Bespiel 1:

```
mpif90 -o example1 example1.f90
mpiexec -np 4 ./example1
```

```
Hello, world, I am  0 of  4
Hello, world, I am  1 of  4
Hello, world, I am  3 of  4
Hello, world, I am  2 of  4
```

# Zuordnung von Arbeit: die kurze Antwort

Wie erreiche ich es, dass jeder Prozessor unterschiedliche Arbeit verrichtet, obwohl doch das selbe Programm läuft?

- MPI weist jedem Prozessor einen *rank* (ein Integer) zu.
- Das Unterprogramm `MPI_Comm_rank` liefert den *rank* (welcher Prozessor) zurück.
- Das Unterprogramm `MPI_Com-_size` liefert den *size* (die Anzahl aller von `mpiexec` zugewiesenen Prozessoren) zurück.

Diese zwei Paramter *rank* und *size* kann man zur Verteilung der Arbeit auf verschieden Prozessoren nutzen.

```
if (irank == 0) then
  ! first working set
end if
if (irank == isize-1) then
  ! last working set
end if
```

# Zuordnung von Arbeit: die lange Antwort

Wenn MPI initialisiert wird, wird ein *communicator* erstellt, der aus einer Gruppe von Prozessoren besteht. Dieser *communicator* ist

`MPI_COMM_WORLD`

Die Anzahl von `m` Prozessoren dieses *communicator* wird durch

```
mpiexec -np m ./my_mpi_program
```

bestimmt. Jeder Prozessor kann diese Anzahl mit dem Unterprogramm

`MPI_Comm_size`

abfragen.

Jeder Prozess in der Gruppe `MPI_COMM_WORLD` bekommt einen *rank* zugewiesen; einem Integer mit einem Wert zwichen 0 und `m`-1. Durch Aufruf des Unterprogramms

`MPI_Comm_rank`

# Benötigte Unterprogrammaufrufe

Welche Unterprogramme muss man mindestens in einem MPI Programm aufrufen?

Das absolute Minimums ist

- `use mpi`
- `call MPI_Init(ierr)`
- `call MPI_Finalize(ierr)`

Kein MPI Programm läuft ohne diese Zeilen!

Die Vorlage eines MPI Pogramms (*template*) findet ihr in `~m214089/mpi/template.f90`.

# Template eines MPI Programms

```
program template

  use mpi

  implicit none

  integer :: ierr, irank, isize, irc

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, irank, ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, isize, ierr)

  call MPI_Finalize(irc)

end program template
```

# Übung 2

Kopiere `template.f90` und erweitere das Programm, so dass es für $x = 5$ auf allen Prozessen

- $y = x^2$ auf Prozess 0,
- $y = x^3$ auf Prozess 1 und
- $y = x^4$ auf Prozess 2

berechnet und den *rank* sowie die Werte von x und y ausgibt.

# Lösung von Übung 2

```
! ...
! define new variables used
real :: x, y
! ...
! set the value of x on all ranks
x = 5.0
! calculate the value of y on each rank
if (irank == 0) then
  y = x**2
else if (irank == 1) then
  y = x**3
else if (irank == 2) then
  y = x**4
endif
! write all required information
write(*,'(a,i3,2(a,f7.2))') 'rank ', irank, ' x = ', x, ' y = ', y
! ...
```

Gibt es noch andere Lösungen (diese liegt unter ~m214089/mpi/exercise2_I.f90)?

# Übung 2: Laufenlassen

```
mpif90 -o exercise2 exercise2.f90
mpiexec -np 3 ./exercise2
```

```
rank    0 x =     5.00 y =    25.00
rank    1 x =     5.00 y =   125.00
rank    2 x =     5.00 y =   625.00
```

# Kommunikation

MPI ist für die Handhabung von Programmen auf Computern mit verteiltem Speicher entworfen worden. Auf Daten, die auf einem Prozessor vorliegen, kann von einem anderen Prozessor durch MPI Funktionen zugegriffen werden.

MPI hat sehr viele Kommunikationsfunktionen, allerdings reichen in der Regel eine handvoll von ihnen aus.

Ein minimaler Satz von Routinen besteht aus

- `MPI_Init`
- `MPI_Comm_size`
- `MPI_Comm_rank`
- `MPI_Send`
- `MPI_Recv`
- `MPI_Finalize`

# Punkt-zu-Punkt Kommunikation

- `MPI_Send` und `MPI_Recv` werden für die Punkt-zu-Punkt Kommunikation benutzt.
- Die beiden Routinen sind gemeinsam für die Übertragung von Daten zuständig.
- Ein Prozess führt die *send* Operation durch und der Zielprozess setzt die *receive* Operation für die zu übertragenden Daten auf.

```
! ...
if (irank == 0) call MPI_Send(..., 1, ...)
if (irank == 1) call MPI_Recv(..., 0, ...)
! ...
```

# Blocking send

```
MPI_Send(buf, icount, datatype, dest, tag, comm, ierr)
```

| | |
|---|---|
| `buf` | initial address of send buffer (choice) |
| `icount` | number of entries to send (integer) |
| `datatype` | datatype of each message entry (handle) |
| `dest` | rank of destination (integer) |
| `tag` | message tag |
| `comm` | communicator (handle) |
| `ierr` | return error code (integer) |

# Blocking receive

```
MPI_Recv(buf, icount, datatype, isource, tag, comm, istatus,i err)}
```

| | |
|---|---|
| `buf` | initial address of receive buffer (choice) |
| `icount` | number of entries to receive (integer) |
| `datatype` | datatype of each entry (handle) |
| `isource` | rank of source (integer) |
| `tag` | message tag |
| `comm` | communicator (handle) |
| `istatus` | return status array (integer) |
| `ierr` | return error code (integer) |

In allen Routinen, in denen `MPI_Recv` aufgerufen wird, muss

```
integer :: istatus(MPI_STATUS_SIZE)
```

deklariert werden. Dieses *array* enthält Informationen über den Sender.

# MPI datatypes

Für alle Fortran Datentypen gibt es entsprechende MPI Datentypen.

| | |
|---|---|
| `integer` | `MPI_INTEGER` |
| `real` | `MPI_REAL` |
| `double precision` | `MPI_DOUBLE_PRECISION` |
| `complex` | `MPI_COMPLEX` |
| `logical` | `MPI_LOGICAL` |
| `character` | `MPI_CHARACTER` |

# Beispiel 2: Punkt-zu-Punkt Kommunikation

Prozess 0 sendet das *array x* an Prozess 1:

```
! ...
integer :: icount, dest, tag, istatus(MPI_STATUS_SIZE)
integer :: comm, datatype, source
real :: x(5), y(5)
! ...
icount   = 5
datatype = MPI_REAL
tag      = 2
comm     = MPI_COMM_WORLD
x(:) = 0.0
y(:) = 0.0
if (irank == 0) then
  x(:) = [ 1.0, 2.0, 3.0, 4.0, 5.0 ]
  dest = 1
  call MPI_Send(x, icount, datatype, dest, tag, comm, ierr)
end if
if (irank == 1) then
  source = 0
  call MPI_Recv(y, icount, datatype, source, tag, comm, istatus, ierr)
end if
write(*,'(a,i4,5f6.1)') 'rank: ', irank, y(:)
! ...
```

Kopiere Beispiel 2 aus `~m214089/mpi/example2.f90`.

# Der *message envelope*

Wie wird von einem MPI Prozess bestimmt, welche Nachricht empfangen wird, wenn mehrere an ihn geschickt wurden?

Jede Nachricht transportiert einen *message envelope* mit sich, der eine Reihe von Informationen zur Verifikation enthält:

- *source*
- *destination*
- *tag*
- *communicator*

Eine *message* wird nur dann empfangen, wenn die Argumente des `MPI_Recv` zum *message envelope* passen.

# Übung 3

Verwende das Programm aus Übung 2, sende alle berechneten $y$ Werte an Prozess 0, berechne den Mittelwert und gib das Ergebnis zusammen mit dem *rank* aus.

# Lösung von Übung 3

```
! ...
integer :: tag, istatus(MPI_STATUS_SIZE)
integer :: ip
real :: x, y, buff
! ...
tag = 1
if (irank == 0) then
  do ip = 1, isize-1
    call MPI_Recv(buff, 1, MPI_REAL, ip, tag, MPI_COMM_WORLD, istatus, ierr
   )
    y = y+buff
  end do
  y = y/real(isize)
  write(*,*) 'the average value of y is ', y
else
  call MPI_Send(y , 1, MPI_REAL, 0, tag, MPI_COMM_WORLD, ierr)
end if
!...
```

Diese Lösung liegt unter ~m214089/mpi/exercise3.f90.

# Übung 3: Laufenlassen

```
mpif90 -o exercise3 exercise3.f90
mpiexec -np 3 ./exercise3
```

```
On process  0  y= 25.
On process  2  y= 625.
On process  1  y= 125.
The average value of y is  258.33333333333331
```

# *Wildcards*

Was passiert, wenn ich nun eine Nachricht unabhängig von *source* und/oder *tag* empfangen möchte?

Für diesen Fall stellt MPI sogenannte *wildcards* zur Verfügung, die im Aufruf von `MPI_Recv` verwendet werden können:

- `MPI_ANY_SOURCE` erlaubt die *source* zu ignorieren:

```
call MPI\_Recv(buf, icount, datatype, MPI_ANY_SOURCE, tag, comm,
  istatus, ierr)
```

- `MPI_ANY_TAG` erlaubt den *tag* zu ignorieren:

```
call MPI\_Recv(buf, icount, datatype, source, MPI\_ANY\_TAG, comm,
  istatus, ierr)
```

- Beide *wildcards* können auch in einem Aufruf kombiniert werden.

Wildcards sollten nur verwendet werden, wenn es absolut notwendig ist!

# *Blocking* und *Non-Blocking*

`MPI_Send` und `MPI_Recv` sind *blocking* Kommunikationsroutinen.

## Was bedeutet es, wenn man `MPI_Send` als *blocking* bezeichnet?

Nach einem Aufruf von `MPI_Send` wird dieser erst beendet, wenn die Nachrichtenübertragung so weit fortgeschritten ist, dass die zu übertragenden Daten nicht durch eine nachfolgende Anweisung überschrieben werden können.

## Was bedeutet es, wenn man `MPI_Recv` als *blocking* bezeichnet?

`MPI_Recv` wird erst beendet, wenn die erwarteten Daten vollständig empfangen wurden.

# Vermeiden hängender Prozesse

Wenn einer oder mehrere Prozesse `MPI_Recv` aufrufen, aber die *messages* nie ankommen, spricht man von *hanging*. Der Emfänger wartet beliebig lange and keine Fehlermeldung wird generiert (das gleiche passiert mit `MPI_Send`). Dieses Problem tritt immer dann auf, wenn man die genaue Reihenfolge von Senden und Empfangen nicht beachtet.

Zunächst ein funktionierendes Beispiel:

```
! ...
if (irank == 0) then
  call MPI_Send(a, 1, MPI_REAL, 1, tag, MPI_COMM_WORLD, ierr)
  call MPI_Recv(b, 1, MPI_REAL, 1, tag, MPI_COMM_WORLD, istatus, ierr)
else if (irank == 1) then
  call MPI_Recv(a, 1, MPI_REAL, 0, tag, MPI_COMM_WORLD, istatus, ierr)
  call MPI_Send(b, 1, MPI_REAL, 0, tag, MPI_COMM_WORLD, ierr)
end if
! ...
```

Kannst du erklären, warum dieser Programmablauf sicher ist?

# Beispiel für ein hängendes Programm

Nehmen wir an, dass die Reihenfolge der *send*- und *receive*- Aufrufe wie folgt verändert wurde:

```
! ...
if (irank == 0) then
  call MPI_Recv(b, 1, MPI_REAL, 1, tag, MPI_COMM_WORLD, istatus, ierr)
  call MPI_Send(a, 1, MPI_REAL, 1, tag, MPI_COMM_WORLD, ierr)
else if (irank == 1) then
  call MPI_Recv(a, 1, MPI_REAL, 0, tag, MPI_COMM_WORLD, istatus, ierr)
  call MPI_Send(b, 1, MPI_REAL, 0, tag, MPI_COMM_WORLD, ierr)
end if
! ...
```

# Beispiel für ein hängendes Programm: warum?

Das Programm aus Beispiel 3 läuft nicht, sondern hängt!

Das bedeutet, dass es niemels endet und auch keine Fehlermeldung ausgibt (mit Ausnahme eines evtl. vorhandenen Zeitlimits).

1. Sowohl Prozess 0 als auch 1 rufen die blockierende `MPI_Recv` auf und erwarten eine Nachricht vom jeweils anderen Prozess.
2. Beide Prozesse führen keine weiteren Operationen aus, solange sie nichts empfangen haben.
3. Allerdings haben beide Prozesse ihre Nachrichten an den Empfänger noch gar nicht versandt!

Folglich warten beide Prozesse unendlich lange auf ungesendete Nachrichten!

# Beispiel eines Programms, dass vielleicht hängt

Diese Reihenfolge der *send*- und receive-Aufrufe funktioniert auf manchen Computern und auf anderen nicht.

Diese Reihenfolge ist nicht empfohlen!

```
! ...
if (irank == 0) then
  call MPI_Send(a, 1,MPI_REAL, 1, tag, MPI_COMM_WORLD, ierr)
  call MPI_Recv(b, 1, MPI_REAL, 1, tag, MPI_COMM_WORLD, istatus, ierr)
else if (irank == 1) then
  call MPI_Send(b, 1, MPI_REAL, 0, tag, MPI_COMM_WORLD, ierr)
  call MPI_Recv(a, 1, MPI_REAL, 0, tag, MPI_COMM_WORLD, istatus, ierr)
end if
! ...
```

# MPI_SENDRECV

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,  &
   &          recvbuf, recvcount, recvtype, source,recvtag, &
   &          comm, istatus, ierr)
```

| | |
|---|---|
| `sendbuf` | initial address of send buffer (choice) |
| `sendcount` | # of entries to send (integer) |
| `sendtype` | type of entries in send buffer (handle) |
| `dest` | rank of destination (integer) |
| `sendtag` | send tag (integer) |
| `recvbuf` | initial address of receive buffer (choice) |
| `recvcount` | max. num. of entries to receive (integer) |
| `recvtype` | type of entries in receive buffer (handle) |
| `source` | rank of source (integer) |
| `recvtag` | receive tag (integer) |
| `istatus` | return status(integer) |
| `comm` | communicator (handle) |
| `ierr` | return error code (integer) |

# Beispiel 4: Verwendung von `MPI_Sendrecv` I

Beispiel 4 zeigt, wie man die *send/receive* Funktion von MPI verwendet. Damit kann man aufeinanderfolgende *send/receive* Aufrufe mit den selben Kommunikationspartnern ersetzen. Die kombinierte *send/receive* Funktion sorgt dafür, dass kein *dead-lock* entsteht.

```
! ...
tag1 = 1
tag2 = 2
if (irank == 0) then
  call MPI_Sendrecv(a, 1, MPI_REAL, 1, tag1, b, 1, MPI_REAL, 1, tag2,
   MPI_COMM_WORLD, istatus, ierr)
else if (irank == 1) then
  call MPI_Sendrecv(b, 1, MPI_REAL, 0, tag2, a, 1, MPI_REAL, 0, tag1,
   MPI_COMM_WORLD, istatus, ierr)
end if
! ...
```

# Beispiel 5: Verwendung von MPI_Sendrecv II

MPI_Sendrecv ist kompatible mit MPI_Send und MPI_Recv. Hier nun ein Beispiel, dass dies zeigt. Prozess 0 sendet a zu Prozess 1 und empfängt b von Prozess 2.

```
! ...
tag1 = 1
tag2 = 2
if (irank == 0) then
  call mpi_sendrecv(a, 1, mpi_real, 1, tag1, b, 1, mpi_real, 2, tag2,
   MPI_COMM_WORLD, istatus, ierr)
else if (irank == 1) then
  call mpi_recv(a, 1, mpi_real, 0, tag1, MPI_COMM_WORLD, istatus, ierr)
else if (irank == 2) then
  call mpi_send(b, 1, mpi_real, 0, tag2, MPI_COMM_WORLD, ierr)
end if
! ...
```

# *Non-blocking communications*

Die am Häufigsten verwendeten *non-blocking* Routinen sind:

```
MPI_Isend(buf, icount, datatype, dest, tag, comm, request, ierr)

MPI_Irecv(buf, icount, datatype, source, tag, comm, request,ierr)
```

| | |
|---|---|
| `buf` | initial address of send buffer (choice) |
| `count` | number of entries to send/receive (integer) |
| `datatype` | datatype of each entry (handle) |
| `dest` | rank of destination process (integer) |
| `source` | rank of source process (integer) |
| `tag` | message tag |
| `comm` | communicator (handle) |
| `request` | request handle (handle) |
| `ierror` | return error code (integer) |

# Unterschied von *blocking* und *non-blocking*

## Was ist der Unterschied zwischen `MPI_Isend` und `MPI_Send`?

`MPI_Isend` kehrt sofort in die aufrufende Routine zurück, nachdem des *send* in Auftrag gegeben wurde. Man beachte, dass es nicht sicher ist den *buffer* zu überschreiben. (`MPI_Irecv` und `MPI_Recv` unterscheiden sich auf ähnlich Weise.)

## Was ist der Vorteil der *non-blocking communication* Routinen?

Die *non-blocking communication* erlaubt es Kommunikation mit Berechnungen zu überlagern, bei denen der *send/receive buffer* nicht beteiligt ist.

## Wann können die *send/receive buffer* sicher wiederverwendet werden?

`MPI_Isend` (und auch `MPI_Irecv`) gebene einen *handle* zurück — das *request* Argument. Die Routine `MPI_Wait` kann später aufgerufen werden, um zu kontrollieren bzw. abzuwarten, ob/bis die entsprechenden *send/receive* Operationen beendet sind.

# *Non-blocking* Beispiel

1. Führe ein *non-blocking send* von Variable $a$ aus.

```
call MPI_Isend(a, ..., request1, ...)
```

2. Während die Kommunikation abläuft,berechne die Werte von $b$, $c$ und $d$ ($a$ wird nicht gebraucht):
   $b = x^2$, $c = y^3$ und $d = b + c$
3. Verhindere Berechnungen mit $a$, solange, bis die Verwendung dieser Variable sicher ist. Verwende dazu

```
call MPI_Wait(request1, istatus, ierr)
```

4. Verwende nun $a$ bei der Berechnung von $e$, modifiziere $a$, etc.
   $e = a + b$, $a = d$, ...

# *Collective* MPI Routines

Kollektive MPI Routinen ermöglichen die simultane Kommunikation aller Prozesse einer Kommunikationsgruppe.

Es gibt drei Gruppen kollektiver Kommunikation:

**Barrier synchronization**

**Global communications**

- broadcast
- gather
- scatter

**Global reduction operations**

- sum
- max
- min
- etc.

# *Broadcasting*

Die *broadcast* Routine ist die am häufigsten verwendete *collective* Routine.

- Der *root* Prozess sendet die Daten an alle anderen Prozesse im gleichen Kommunikator.
- Alle beteiligten Prozesse müssen `MPI_Bcast` mit dem gleichen Wert für den *root* Prozess aufrufen.

```
MPI_Bcast(buffer, icount, datatype, root, comm, ierr)
```

| | |
|---|---|
| `buffer` | initial address of buffer (choice) |
| `icount` | number of entries in buffer (integer) |
| `datatype` | datatype of buffer (handle) |
| `root` | rank of broadcasting process (integer) |
| `comm` | communicator (handle) |
| `ierror` | return error code (integer) |

# Übung 4: Verwende *broadcast*

Modifiziere das Programm von Übung 2 so, dass der Wert für x nur auf *rank* 0 bekannt ist. Füge den `MPI_Bcast`-Aufruf hinzu, der den Wert von x an alle beteiligten Prozessoren verteilt.

# Lösung von Übung 4

```
! ...
integer :: ip
real :: x, y
! ...
if (irank == 0) x = 5.0

call MPI_Bcast(x, 1, MPI_REAL, 0, MPI_COMM_WORLD, ierr)

do ip = 1,isize
  if (irank == ip-1) then
    y = x**(2+irank)
    write(*,*)'On process ',irank,' y = ', y
  end if
end do
! ...
```

# Und nun?

Wissenschaftliche Aufgaben:

- Erweiterung der daisyworld ...
- Eine einfache Wettervorhersage ...