



Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

Praktikum: Paralleles Programmieren für Geowissenschaftler

Prof. Thomas Ludwig, Hermann Lenhart, Tim Jammer



Dr. Hermann-J. Lenhart

hermann.lenhart@informatik.uni-hamburg.de



MPI Kommunikation:

Ein wesentliches Kriterium für die Entwicklung paralleler Programme besteht darin die Kommunikation effektiv zu gestalten.

Zur Information:

Ein moderner Parallelrechner kann

bis zu **3 Mrd floating-point Operationen pro Sekunde** berechnen,

aber nur etwa **10 Millionen Wörter pro Sekunde**

zwischen den Prozessen verschicken!

(Using MPI; Gropp,Lusk,Skjellm, 1999)



MPI Standard Send / Receive = Blocking Communication

MPI_SEND Programm läuft erst weiter nachdem
Send-Buffer zur Wiederverwendung gelehrt wurde

MPI_RECV Programm läuft erst weiter nachdem
Receive-Buffer gefüllt ist, d.h. die Daten stehen zur Verfügung

Die Ausführung der Kommunikation ist abhängig von der Größe der Nachricht
und der Größe des Systembuffers .

Blockierende Kommunikaton ist einfach einzusetzen, aber anfällig für Deadlocks!

„It is incorrect for an MPI program to require buffering in MPI send“ (Groop,Lusk & Skjellum)

Programmbeispiel für Deadlock I

program main

```
use mod_communication
use mpi
implicit none
```

```
integer :: ierr, rank, numProcesses,, root=0, array_size, i, max_size=20000
real, dimension(:), allocatable :: array
```

```
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numProcesses, ierr)
```

```
do i=1,max_size
    array_size = i
```

```
    call initialize_array(array_size, array) => allocate(array(array_size))
```

```
    call deadlock(numProcesses, rank, array)
```

```
    if (rank == root) print*, 'Program finished successfully: ', i, ' array entries', i*4, ' bytes'
```

```
    deallocate(array)
```

```
end do
```

```
call MPI_FINALIZE(ierr)
```

end program main

Programmbeispiel für Deadlock II

```
subroutine deadlock(numProcesses, rank, array)
```

```
integer, intent(in) :: numProcesses, rank
```

```
real, dimension(:), intent(in) :: array
```

```
integer :: partner1, partner2, array_size, ierr, status(MPI_STATUS_SIZE)
```

```
array_size = size(array)
```

```
call calc_partner(rank, numProcesses, partner1, partner2)
```

```
call MPI_SEND(array, array_size, MPI_REAL, partner1, rank, MPI_COMM_WORLD, ierr)
```

```
call MPI_RECV(array, array_size, MPI_REAL, partner2, partner2, MPI_COMM_WORLD, status, ierr)
```

```
end subroutine deadlock
```

```
subroutine calc_partner(rank, numProcesses, partner1, partner2)
```

```
integer, intent(in) :: rank, numProcesses
```

```
integer, intent(out) :: partner1, partner2
```

```
partner1 = rank+1
```

```
partner2 = rank-1
```

```
if (rank == numProcesses-1) then
```

```
    partner1 = 0
```

```
else if (rank == 0) then
```

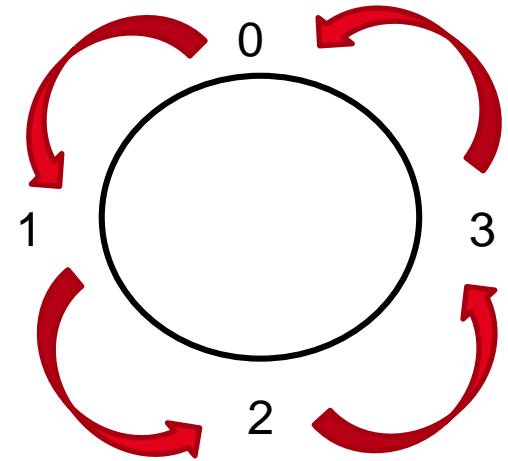
```
    partner2 = numProcesses-1
```

```
end if
```

```
end subroutine calc_partner
```

Programmbeispiel für Deadlock IIIA

```
subroutine calc_partner(rank, numProcesses, partner1, partner2)
  integer, intent(in) :: rank, numProcesses
  integer, intent(out) :: partner1, partner2
  partner1 = rank+1
  partner2 = rank-1
  if (rank == numProcesses-1) then
    partner1 = 0
  else if (rank == 0) then
    partner2 = numProcesses-1
  end if
end subroutine calc_partner
```

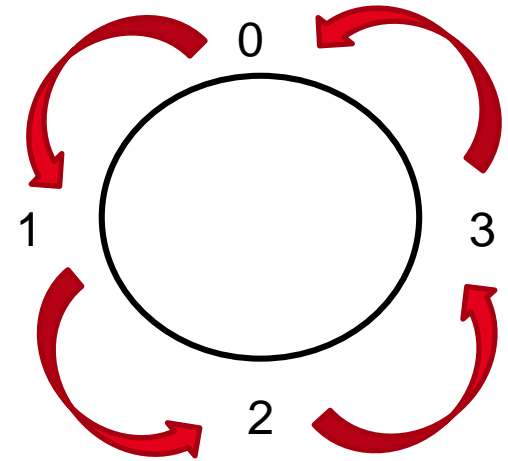


Beispiel: numProcesses = 4

	SEND	Dest	Tag
rank = 0		1	0
rank = 1		2	1
rank = 2		3	2
rank = 3		0	3

Programmbeispiel für Deadlock IIIB

```
subroutine calc_partner(rank, numProcesses, partner1, partner2)
  integer, intent(in) :: rank, numProcesses
  integer, intent(out) :: partner1, partner2
  partner1 = rank+1
  partner2 = rank-1
  if (rank == numProcesses-1) then
    partner1 = 0
  else if (rank == 0) then
    partner2 = numProcesses-1
  end if
end subroutine calc_partner
```



Beispiel: numProcesses = 4

	SEND	Dest	Tag	RECV	Source	Tag
rank = 0		1	0		3	3
rank = 1		2	1		0	0
rank = 2		3	2		1	1
rank = 3		0	3		2	2

Programmbeispiel für Deadlock VI

```
subroutine deadlock(numProcesses, rank, array)
```

```
.....
```

```
array_size = size(array)
```





```
call calc_partner(rank, numProcesses, partner1, partner2)
```

```
call MPI_SEND(array, array_size, MPI_REAL, partner1, rank, MPI_COMM_WORLD, ierr)
```

```
call MPI_RECV(array, array_size, MPI_REAL, partner2, partner2, MPI_COMM_WORLD, status, ierr)
```

```
end subroutine deadlock
```

Beispiel: numProcesses = 4

	SEND	Dest	Tag		RECV	Source	Tag
rank = 0		1	0			3	3
rank = 1		2	1			0	0
rank = 2		3	2			1	1
rank = 3		0	3			2	2

Kein Deadlock durch Anordnung der send-receive Befehle

```
if (mod(rank,2) == 0) then
```

```
    call MPI_SEND(array, array_size, MPI_REAL, partner1, rank, MPI_COMM_WORLD, ierr)
```

```
    call MPI_RECV(array, array_size, MPI_REAL, partner2, partner2, MPI_COMM_WORLD, status, ierr)
```

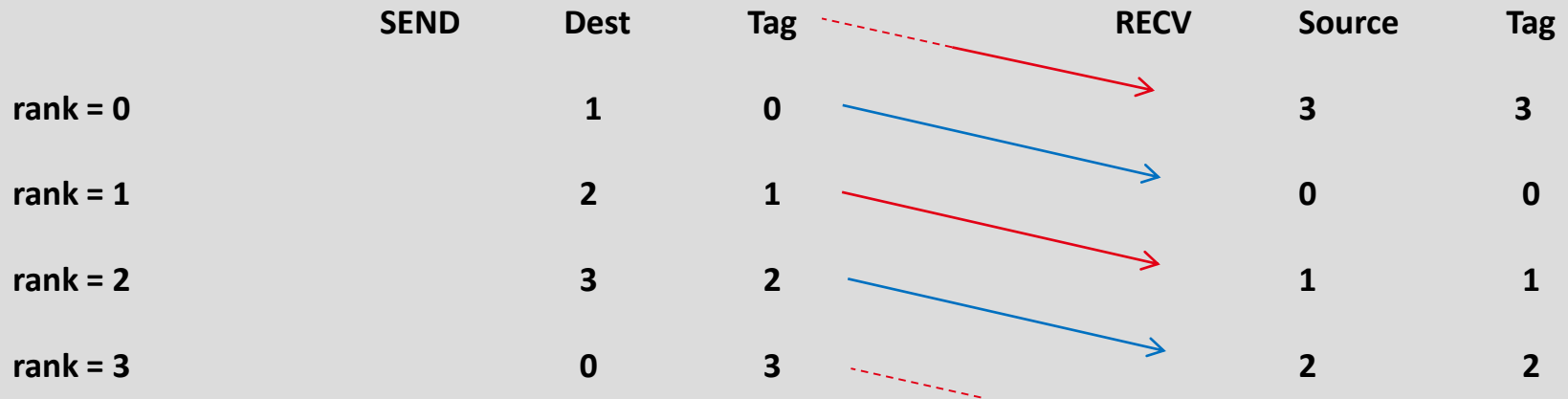
```
else
```

```
    call MPI_RECV(array, array_size, MPI_REAL, partner2, partner2, MPI_COMM_WORLD, status, ierr)
```

```
    call MPI_SEND(array, array_size, MPI_REAL, partner1, rank, MPI_COMM_WORLD, ierr)
```

```
end if
```

Beispiel: numProcesses = 4





MPI sendrecv

Eine weitere Möglichkeit Deadlocks zu vermeiden ist die Anwendung des kombinierten Send-Receive Befehls

Syntax:

```
MPI_SENDRECV(Send-Message, Send-Count, Send-Datatype, Dest, Send-Tag,  
              Recv-Message, Recv-Count, Recv-Datatype, Source, Recv-Tag,  
              Comm, status, lerror)
```

Kombinierter Send-Receive Befehl gegen Deadlocks

```
subroutine sendrecv(numProcesses, rank, array)
```

```
integer, intent(in) :: numProcesses, rank
```

```
real, dimension(:), intent(in) :: array
```

```
integer :: partner1, partner2, array_size, ierr, status(MPI_STATUS_SIZE)
```

```
array_size = size(array)
```

```
call calc_partner(rank, numProcesses, partner1, partner2)
```

```
call MPI_SENDRECV(array, array_size, MPI_REAL, partner1, rank, &  
& array, array_size, MPI_REAL, partner2, partner2, &  
& MPI_COMM_WORLD, status, ierr)
```

```
end subroutine sendrecv
```

Erlaubt das Versenden und Empfangen von Daten
ohne Sorge vor Deadlock
aus Mangel an Bufferkapazität!



MPI nicht blockierende Kommunikation

Eine weitere Option bietet die nicht-blockierende Kommunikation (non-blocking communication)

Zwischen dem Senden (nach Füllen des Buffers) kann dann weiter gerechnet werden
=> Umgehen der Latenz in der Kommunikation

Unabhängig vom Zeitpunkt des Füllens des Buffers entscheidet der Programmierer wann die Nachricht mit dem receive Befehl empfangen wird.

Nachteil: Programmierer muss Abfragen einfügen um die Abwicklung zu checken.



MPI non-blocking Send

MPI_IREND(Message, Count, Datatype, Dest, Tag, Comm, **request**, lerror)

z.B:

Call MPI_IREND(temp, 1, MPI_Real, dest, tag, MPI_COMM_World, req, lerror)

temp	Adresse des Sendepuffers; Real :: temp
1	Count – Anzahl der Elemente im Puffer
MPI_Real	Datentyp des gesendeten Elementes
dest	Angabe des Ranges des Zielprozesses; integer :: dest
tag	Nachrichtenkennung; integer :: tag
MPI_COMM_World	Kommunikator (Gruppe, Kontext)
request	Handle; integer :: req , lerror



MPI non-blocking Receive

MPI_IRECV(Message, Count, Datatype, Source, Tag, Comm, **request**, ierror)

z.B:

Call MPI_IRECV(temp, 1, MPI_Real, source, tag, MPI_COMM_World, **req**, ierror)

temp	Adresse des Sendepuffers; Real :: temp
1	Count – Anzahl der Elemente im Puffer
MPI_Real	Datentyp des gesendeten Elementes
source	Angabe des Ranges des Sourceprozesses; integer :: source
tag	Nachrichtenkennung; integer :: tag
MPI_COMM_World	Kommunikator (Gruppe, Kontext)
request	Handle; integer :: req !!! Kein Status , ierror



MPI non-blocking Kommunikation Statusabfrage

Zur Abfrage des **Status der Nachricht** , gibt es das Wait Kommando:

MPI_WAIT(request, **status**, ierror)

Integer :: request, **status(MPI_STATUS_SIZE)**, ierror

D.h. die Statusabfrage wird quasi aus dem Send und Receive Befehl „ausgelagert“!



MPI non-blocking Kommunikation - Buffering

Ein Send ist abgeschlossen wenn der dem User zur Verfügung gestellte Buffer wieder neu verwendet werden kann.

Ein abgeschlossener Send-Befehl bedeutet nicht, dass auch der Receive Befehl abgeschlossen wurde. Es können folgende Situationen auftreten:

- > Die Nachricht kann vom System gebuffert werden
- > Die Nachricht kann sich noch im Transfer befinden



MPI non-blocking Kommunikation Statusabfrage

Bei der Verwendung von ISEND und IRECV ergibt sich folgende Abfolge der Befehle :

Call MPI_ISEND(Message, Count, Datatype, Dest, Tag, Comm, request, lerror)

..... Der Prozess rechnet weiter

Call WAIT (request, status, ierr)

Call MPI_IRECV(Message, Count, Datatype, Source, Tag, Comm, request, lerror)

Call WAIT (request, status, ierr)



Geht weiter wenn die ISEND-Nachricht angekommen ist und der Buffer wieder frei ist und neu belegt werden kann.

Programmbeispiel für non-blocking Kommunikation

```
subroutine non_blocking_communication(numProcesses, rank, array)
```

```
integer, intent(in) :: numProcesses, rank
```

```
real, dimension(:), intent(in) :: array
```

```
integer :: partner1, partner2, array_size, ierr, request, status(MPI_STATUS_SIZE)
```

```
array_size = size(array)
```

```
call calc_partner(rank, numProcesses, partner1, partner2)
```

```
call MPI_ISEND(array, array_size, MPI_REAL, partner1, rank, MPI_COMM_WORLD, request, ierr)
```

```
call MPI_WAIT(request, status, ierr)
```

```
call MPI_IRECV(array, array_size, MPI_REAL, partner2, partner2, MPI_COMM_WORLD, &  
& request, ierr)
```

```
call MPI_WAIT(request, status, ierr)
```

```
end subroutine non_blocking_communication
```



MPI non-blocking Kommunikation Statusabfrage II

Die WAIT Routine blockiert den Programmablauf
bis die Kommunikation beendet ist.

d.h. man kann zwischen iSend und iRecv Berechnungen einschieben
bis WAIT erreicht wird.

Hinweis: Anywhere you use MPI_Send or MPI_Recv,
you can use the pair of MPI_Send/MPI-Wait or MPI_iRecv/MPI-Wait

Quelle: MPI Tutorial, P. Bangalore & A. Skjellum



MPI non-blocking Kommunikation Statusabfrage III

Zur Abfrage des aktuellen **Status der Nachricht zur Programmsteuerung**

gibt es das TEST Kommando:

```
MPI_TEST(request, flag, status, lerror)
```

```
Integer :: request, status(MPI_STATUS_SIZE), ierror
```

```
Logical :: flag
```

Gibt einen TRUE Wert zurück sobald die Kommunikation abgeschlossen wurde.

```
Do while ( .not. flag)
```

```
.....
```

```
call MPI_TEST(request, flag, status, lerror)
```

```
enddo
```



Zusammenfassung

- Achte auf effektive Kommunikation - 3 Mrd. floating point op./s vs. 10 Mio .Wörter /s
- Standard (blocking) send und recv-Befehle sind anfällig für Deadlocks
- Kombiniertes Sendrecv-Befehl umgeht dieses Problem, aber immer noch Latenz
- Non-Blocking Kommunikation mittels lsend und lrecv-Befehlen erlaubt eine bessere Verzahnung von Berechnung und Kommunikation
- Allerdings muss der Nachrichtenaustausch genau bekannt sein!



Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG



Danke das wars!