Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Praktikum: Paralleles Programmieren für Geowissenschaftler

Prof. Thomas Ludwig, Hermann Lenhart, Ulrich Körner, Nathanael Hübbe

Dr. Hermann-J. Lenhart
hermann.lenhart@zmaw.de

MPI Einführung II:

- Kommunikation
- Point to Point Communication
- Nicht blockierende Kommunikation
- MPI Barrier
- Kollektive Operationen

MPI Kommunikation:

Das wichtigste Kriterium für die Entwicklung paralleler Programme besteht darin die Kommunikation effektiv zu gestalten.

Zur Information:

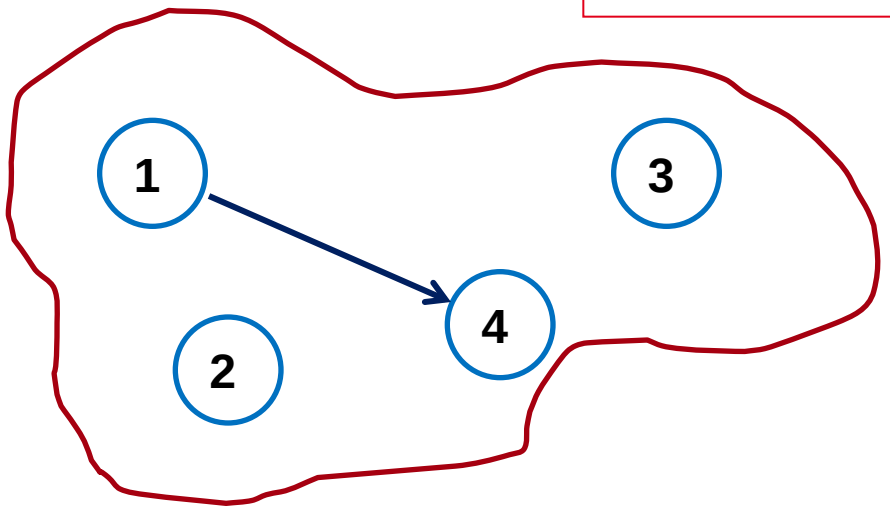
Ein moderner Parallelrechner kann
bis zu **500 Millionen floating-point Operationen pro Sekunde** berechnen,

aber nur etwa **10 Millionen Wörter pro Sekunde**
zwischen den Prozessen verschicken!

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

MPI Point to Point Communication:

Kommunikator: MPI_COMM_WORLD



Send -> Receive

MPI Point to Point Communication:

MPI verfügt über **4 verschiedene Send- Funktionen**

Die in unterschiedlichen Send Befehlen dargestellt werden:

Standard

Synchronisiert (synchronous)

Buffered

(Ready)

Alle werden mit dem gleichen Receive Befehl beendet.

„Beendigung“ heißt, der send buffer kann sicher wieder verwendet werden.

MPI Standard Send I

MPI_SEND(Message, Count, Datatype, Dest, Tag, Comm, Ierror)

z.B:

Call MPI_SEND(temp, 1, MPI_Real, dest, tag, MPI_COMM_World, Ierror)

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)
Ierror	Fehlerstatus; integer :: Ierror

MPI Standard Send II

Beim Standard send ist zu beachten:

Der Standard Send Befehl ist beendet sobald die Message verschickt wurde; unabhängig davon ob die Message schon beim Empfänger angekommen ist, sie kann immer noch für einige Zeit im Netzwerk liegen.

MPI Synchronous Send I

Wenn sicher gestellt werden soll dass die Message beim empfangenden Prozess angekommen ist, so sollte **synchronous send** verwendet werden.

Wie bei einem „Einschreiben mit Rückantwort“ erhält der Sender
eine Bestätigung das die Message angekommen ist.
Erst wenn der Sender diese Bestätigung erhalten hat
wird die Sende Operation als abgeschlossen angesehen.

Solange die Bestätigung nicht eingetroffen ist wird der Prozess auf „idle“ gestellt, d.h. das Netzwerk kann niemals mit überfrachtet werden.

MPI Synchronous Send II

MPI_**S**SEND(Message, Count, Datatype, Dest, Tag, Comm, Ierror)

z.B:

Call MPI_**S**SEND(temp, 1, MPI_Real, dest, tag, MPI_COMM_World, Ierror)

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)
Ierror	Fehlerstatus; integer :: Ierror

MPI Buffered Send I

Die Option **buffered send** garantiert den Abschluss der Sende Operation sofort, indem die gesendete Message in einen System buffer kopiert wird – für spätere Abwicklung der Zustellung.

Damit ist eine klare Annahme verbunden,

Sender und Empfänger können nicht synchron sein.

Der Programmierer muss dem Programm genug Speicher für den Buffer einräumen:

MPI_BUFFER_ATTACH (buffer, *size*,ierror) -> Array buffer mit *size* bytes

MPI_BUFFER_DETACH (buffer,*size*,ierror)

MPI Buffered Send II

MPI_**B**SEND(Message, Count, Datatype, Dest, Tag, Comm, Ierror)

z.B:

Call MPI_**B**SEND(temp, 1, MPI_Real, dest, tag, MPI_COMM_World, Ierror)

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)
Ierror	Fehlerstatus; integer :: Ierror

MPI Send/Receive Übersicht

Standard Send	MPI_SEND
Synchronous Send	MPI_SSEND
Buffered Send	MPI_BSEND
(Ready Send	MPI_RSEND)
Empfangen	MPI_RECV

MPI nicht blockierende Kommunikation

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

Hierbei ist man nicht mehr direkt vom Buffer abhängig, sondern der Programmierer entscheidet selbstständig wann die Nachricht mit dem receive Befehl platziert wird.

Die Syntax dazu bietet: MPI_ISEND und MPI_IRECV

MPI non-blocking Send

MPI_ISEND(Message, Count, Datatype, Dest, Tag, Comm, request, Ierror)

z.B:

Call MPI_ISEND(temp, 1, MPI_Real, dest, 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
dest	Angabe des Ranges des Zielprozesses; integer :: dest
tag	Nachrichtenkennung; integer :: tag
MPI_COMM_World	Kommunikator (Gruppe, Kontext)
request	Handle; integer :: req , Ierror

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 Isend Nachricht** , gibt es folgende Option:

MPI_WAIT(request,status, Ierror)

Integer :: request, status(MPI_STATUS_SIZE), ierror

CALL MPI_ISEND(buffer, count, datatype, dest, tag, comm, request, ierr)

….. Der Prozess rechnet weiter

Call WAIT (request, status, ierr)

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

MPI non-blocking Kommunikation Statusabfrage

Zur Abfrage des Status von mehreren Nachricht , gibt es **MPI_WAITALL**:

```
Integer ::  nx, s, e, comm1d, nbrtop, nbrbot, status_array(MPI_STATUS_SIZE,4), req(4)
Real: a(0:nx+1,s-1:e+1)
Call MPI_IRECV (a(1,s-1), nx, MPI_REAL,nbrbot,0,comm1d,req(1),ierr)
Call MPI_IRECV (a(1,e+1), nx, MPI_REAL,nbrtop,1,comm1d,req(2),ierr)

Call MPI_SEND(a(1,e), nx, MPI_REAL, nbrtop, 0, comm1d, req(3), ierr)
Call MPI_SEND(a(1,s), nx, MPI_REAL, nbrbot, 1, comm1d, req(4), ierr)

Call MPI_WAITALL(4,req,status_array, ierr)
```

MPI Barrier I

Der MPI_BARRIER Befehl wird zur Programmsteuerung eingesetzt.

Call MPI_BARRIER(MPI_COMM_World, Ierror)

Der MPI_Barrier Befehl erzwingt dass alle Prozesse den gleichen Punkt im Code erreicht haben bevor das Programm weiterläuft.

MPI Barrier II

Der MPI_BARRIER Befehl wird vorranging zur Zeitmessung eingesetzt, z.B.

….

```
Call MPI_BARRIER(MPI_COMM_World, ierror)
t1 = MPI_WTIME()
….
Call MPI_BARRIER(MPI_COMM_World, ierror)
total_time = MPI_WTIME() – t1
```

MPI Kollektive Operationen

MPI verfügt über umfangreiche Operationen zum kollektiven Bewegen von Daten.

Dazu gehören:

MPI_BROADCAST

MPI_REDUCE

MPI_GATHER

MPI_SCATTER

MPI Gather I

Eine weitere Möglichkeit Teilarrays der Threads (z.B. für I/O Zwecke) zusammenzuführen, bietet MPI_GATHER:

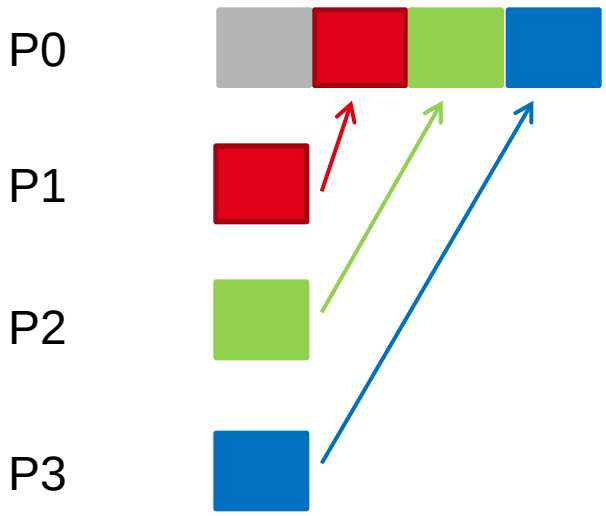
Syntax: MPI_Gather(Sendmessage, Sendcount, Sendtype,
Recvmessage, Recvcount, Recvtype,
Root, Comm, Ierror)

Call MPI_GATHER (temp, 1, MPI_Real,
temps,1, MPI_Real, 0, MPI_COMM_World, Ierror)

MPI Gather II

Call MPI_GATHER(temp, 1, MPI_Real,

tempAll,1, MPI_Real, 0, MPI_COMM_World, Ierror)



MPI Scatter I

Eine Möglichkeit z.B. eine Anfangsbelegung auf die Teilarrays der Threads ,
zu übertragen bietet MPI_SCATTER:

Syntax: MPI_Scatter(Sendmessage, Sendcount, Sendtype,
Recvmessage, Recvcount, Recvtype,
Root, Comm, Ierror)

MPI Scatter II

Die Daten werden von P0 an die anderen Threads P1 – P3 gesendet.