Grundlagen zu MPI

Martin Lanser
Mathematisches Institut
Universität zu Köln

October 24, 2017

Überblick

- Einige Grundlagen
- Punkt-zu-Punkt Kommunikation
- Globale Kommunikation
- Kommunikation plus Berechnung

Basics

- **MPI** steht für **M**essage **P**assing **I**nterface.
- **MPI** ist ein festgelegter Standard, der die Kommunikation bzw. den Nachrichtenaustausch zwischen den verschiedenen Prozessen bei parallelen Berechnungen beschreibt.
- **MPI** ist eine Programmierschnittstelle und legt die möglichen Operationen und deren Semantik fest.
- **MPI** ist vor allem für den Einsatz auf Systemen mit verteiltem Speicher **(distributed memory)** gedacht, funktioniert aber auch auf Systemen mit gemeinsam genutztem Speicher **(shared memory)**.
- **MPI** kann auch über Rechnergrenzen hinweg eingesetzt werden.

- Ein mit MPI parallelisiertes Programm besteht aus mehreren miteinander kommunizierenden aber unabhängigen **MPI-Prozessen**.

- Alle Prozesse werden beim Programmstart parallel initialisiert und führen meist im SPMD Sinne das gleiche Programm aus, aber auf unterschiedlichen Daten!

- Jeder MPI-Prozess hat seinen eigenen privaten Speicher und **keinen direkten Zugriff auf den Speicher der anderen Prozesse!**

- Alle Prozesse arbeiten gemeinsam an einem Problem und tauschen mithilfe der MPI-Schnittstelle explizit Nachrichten bzw. Daten aus.

- Es ist sowohl eine Punkt-zu-Punkt Kommunikation (zwischen zwei Prozessen) als auch eine globale Kommunikation (alle Prozesse in einer Gruppe kommunizieren) möglich.

- Zwei Grundbegriffe in der Kommunikationsstruktur von MPI:

- Ein **Kommunikator** ist eine Gruppe von **MPI-Prozessen**.

- Der vordefinierte Kommunikator **MPI_COMM_WORLD** enthält alle gestarteten MPI-Prozesse.

- Jeder MPI-Prozess hat eine eindeutige Kennung, den sogenannten **Rang (rank)**.

Kompilieren eines MPI Programms

- Zwei frei verfügbare Implementierungen von MPI sind **MPICH und OpenMPI**.
- Nach Installation einer der beiden, steht ein MPI Compiler zur Verfügung.
- Das Kompilieren eines mit MPI parallelisierten C-Programms funktioniert im Wesentlichen wie in C. Einziger Unterschied: Anstelle des Wrappers **gcc**, der zu einem gewählten C-Compiler verlinkt, nutzt man **mpicc**.
- Der Vollständigkeit halber einige MPI-Compiler für andere Sprachen: **mpicxx** (C++) ; **mpif77** (Fortran 77); **mpif90** (Fortran 90).

Aufruf eines MPI Programms

- Der Programmaufruf startet mit einem **mpirun** oder **mpiexec**.
- Nun weiß der Rechner, dass er das Programm parallel und mithilfe von MPI ausführen soll.
- Mit **-np=# Prozesse** legt man die Anzahl der MPI-Prozesse fest. (**np** = **n**umber of **p**rocesses)
- **Beispiel:** Ausführung des Programmes **test** mit 4 Prozessen

mpirun -np=4 ./test

Grundlegende Befehle

- Die MPI-Bibliothek muss mit **#include "mpi.h"** eingebunden werden.
- Ein MPI Programm beginnt mit dem Befehl **MPI_Init(int *argc, char ***argv)**.
- **MPI_Init** startet die MPI Umgebung und fasst alle Prozesse in dem Kommunikator **MPI_COMM_WORLD** zusammen.
- Jeder Prozess bekommt eine eindeutige "Nummer" zugewiesen, den sogenannten **Rang (rank)**.
- **MPI_Init** wird nur vom Hauptprozess (meistens der Prozess mit dem Rang 0) aufgerufen. Das gilt ebenfalls für:
- **MPI_Finalize()** beendet die MPI Umgebung und sollte am Ende jedes Programmes stehen.

- **MPI_Comm_rank(MPI_Comm comm, int *rank)** liest den Rang des Prozesses aus, von dem der Befehl ausgeführt wird, und speichert ihn in der Variablen **rank**.

- **MPI_Comm_size(MPI_Comm comm, int *size)** bestimmt die Anzahl der MPI Prozesse im Kommunikator **comm** und speichert sie in der Variablen **size**.

- **MPI_Barrier(MPI_Comm comm)** lässt alle Prozesse im Kommunikator **comm** solange warten, bis alle Prozesse die Barriere erreicht haben.

- Zeigen und erläutern mit test.c.

Listing 1: Basics MPI

```
int main(int argc, char *argv[])
{
    int size,rank;

    //initialisiert die parallelen Prozesse und gibt jedem einen Rang (rank).
    MPI_Init(&argc, &argv);

//——Alle folgenden Befehle werden parallel von allen Prozessen bearbeitet!!

    //"Wenn ich ein Prozess aus MPI_COMM_WORLD bin, speichere ich meinen Rang!!"
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    //Gesamtzahl der Prozesse in MPI_COMM_WORLD
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    //Gibt auf jedem Prozess rank und size aus
    printf("ich bin Prozess %i von %i\n",rank,size);

    //Hier soll ein Prozess was anderes ausgeben als die anderen.
    if (rank==3)
        printf("ich bin Prozess %i von %i\n",rank,size);
    else
        printf("ich bin nicht Prozess 3 von %i\n",size);
```

```
    //Beendet MPI.
    MPI_Finalize();
    return 0;
}
```

Punkt-zu-Punkt Kommunikation

- Punkt-zu-Punkt Kommunikation bedeutet die Kommunikation zweier Prozesse untereinander. Präziser: Ein Prozess **versendet** Daten, ein anderer **empfängt** diese Daten.

- Hierzu benötigen wir einen Sendebefehl, **MPI_Send**, und den entsprechenden Empfangsbefehl, **MPI_Recv**.

- Hierbei handelt es sich um **blockierende Kommunikation**, da sowohl **MPI_Send** als auch **MPI_Recv** den Prozess solange blockieren oder anhalten, bis die Datenübertragung vollständig ausgeführt wurde.

- Zu jedem **MPI_Send** muss immer ein passendes **MPI_Recv** existieren, sonst blockiert das Programm **(Gefahr eines Deadlocks)**.

- Die zu verschickenden Daten werden zu sogenannten **Nachrichten** gepackt, die durch drei Parameter beschrieben werden: **buffer, count, datatype**.

- Der **buffer** ist eine ausreichende Menge an Speicher, in dem die zu verschickenden oder zu empfangenden Daten abgelegt werden können.

- Zu versenden sind **count** viele Daten vom Typ **datatype**.

MPI_Send

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

- **buf** : Zeiger auf den Sendepuffer
- **count** : Anzahl der zu verschickenden Daten // Anzahl der Elemente im Sendepuffer
- **datatype** : Datentyp der Elemente im Sendepuffer
- **dest** : Rang des Zielprozesses
- **tag** : Markierung der Nachricht
- **comm** : Kommunikator

MPI_Recv

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

- **buf** : Zeiger auf den Empfangspuffer // z.B. ein Array, in dem die empfangenen Daten abgelegt werden sollen.
- **count** : Anzahl der zu empfangenden Daten // Anzahl der Elemente im Empfangspuffer
- **datatype** : Datentyp der Elemente im Empfangspuffer
- **source** : Rang des Quellprozesses // Rang des sendenden Prozesses
- Mit source=**MPI_ANY_SOURCE** werden Nachrichten von jedem Prozess empfangen.

- **tag** : Markierung der Nachricht
- Mit tag=**MPI_ANY_TAG** werden Nachrichten mit beliebiger Markierung empfangen.
- **comm** : Kommunikator
- **status** : Zeiger auf eine Statusstruktur / Speicher für Informationen über die empfangene Nachricht.
- (Zeige ptop.c // Zeige Blockieren)

Listing 2: Beispiel MPI_Send und MPI_Recv

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int size,rank,tag=42;
    MPI_Init(&argc, &argv);
    MPI_Status status;

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

    //Erstelle einen 4er Array, der auf jedem Prozess
    //zwei unterschiedliche Eintraege hat.
    int array[4]={0,0,0,0};

    array[0]=rank*size;
    array[1]=rank*size+1;

    //Wie sehen die Arrays aus??
    printf("Thread %i haelt array [%i,%i,%i,%i]\n",rank,array[0],
    array[1],array[2],array[3]);
```

```
    //proc 0 soll zwei Eintraege von proc 1 empfangen
    // und an die Stellen 3 und 4 im Array schreiben
    if (rank==0)
        MPI_Recv(array+2,2,MPI_INT,1,MPI_ANY_TAG,MPI_COMM_WORLD,&status);

    //proc 1 soll die ersten beiden Eintraege des Arrays senden.
    else if (rank==1)
        MPI_Send(array,2,MPI_INT,0,tag,MPI_COMM_WORLD);

    //Ergebnis anschauen:
    printf("Thread %i haelt nach Senden array [%i,%i,%i%i]\n",
    rank,array[0],array[1],array[2],array[3]);

    //Beendet MPI.
    MPI_Finalize();
    return 0;
}
```

Frage: Wie könnte man eine Blockade erzeugen?

Nichtblockierende Kommunikation

- In manchen Situation ist es für die Effizienz der parallelen Software besser, bei der Kommunikation nicht zu blockieren.
- Zum Beispiel kann die Kommunikation oft parallel zu anderen Berechnungen (Additionen, Multiplikationen, Allokationen) bearbeitet werden.
- Der MPI Standard hält hierfür die sogenannte **nichtblockierende Kommunikation** bereit.
- Hier wird die Kommunikation nur angestoßen und läuft danach parallel zu den folgenden Aufgaben weiter, ohne auf die Beendigung des Sende-/Empfangsvorgangs zu warten.
- Die benötigten Befehle unterscheiden sich nur geringfügig von den blockierenden Varianten: **MPI_Isend** und **MPI_Irecv**.

- Um sicher zu stellen, dass zu einem bestimmten Zeitpunkt / an einer bestimmten Codestelle die Kommunikation abgeschlossen ist, nutzt man die spezifische Barriere **MPI_Wait**.

- Das ist z.B. notwendig, falls man an einer bestimmten Stelle die vollständig übertragenen Daten benötigt.

- Ist die Übertragung bereits geschehen, bewirkt **MPI_Wait** nichts, ansonsten blockieren die beteiligten Prozesse und der Sende-/Empfangsvorgang wird beendet.

MPI_Isend und MPI_Irecv

int **MPI_Isend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

int **MPI_Irecv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

- Alle Parameter bis auf **request** sind analog zu **MPI_Send** bzw. **MPI_Recv**.
- Mithilfe von **request** kann abgefragt werden, ob die Sende-/Empfangsoperation beendet ist und ggf. mithilfe von **MPI_Wait** auf ihre Beendigung gewartet werden.

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

- Gleiches Beispiel wie zuvor, nur mit nichtblockierender Kommunikation.

Listing 3: Beispiel MPI_Isend und MPI_Irecv

```
...
...
    if (rank==0)
        MPI_Irecv(array+2,2,MPI_INT,1,MPI_ANY_TAG,MPI_COMM_WORLD,&request);

    else if (rank==1)
        MPI_Isend(array,2,MPI_INT,0,tag,MPI_COMM_WORLD,&request);

        ////////////////////////////////
        //Do Work here!!!///////
        //Parallel to communication!!

        MPI_Wait(&request,&status);
...
...
```

Weitere Punkt-zu-Punkt Kommunikationen

- Mit **MPI_Sendrecv** kann man Senden und Empfangen in einem Befehl zusammenfassen.
- **MPI_Sendrecv_replace** fasst ebenfalls Senden und Empfangen in einem Befehl zusammen und nutzt nur einen **Puffer als Sende- und Empfangspuffer**.
- **MPI_Request_free** löscht ein MPI_Request Element aus dem Speicher.
- Mit **MPI_PROC_NULL** steht ein Dummy-Prozess zur Verfügung.
- Kommunikation mit einem Dummy-Prozess bewirkt nichts, benötigt kaum Zeit und blockiert nie.

- Solche Kommunikation kann oft nützlich sein um den Code zu vereinfachen, z.B. an Rändern eines Problems.

Globale Kommunikation

- Im MPI-Standard stehen einige Kommunikationsbefehle zur Verfügung, die alle Prozesse in einer Kommunikatorgruppe beteiligen.
- Dies kann vor allem als Vereinfachung gesehen werden, da hier im Prinzip viele Punkt-zu-Punkt Kommunikationen zu einem Muster gesammelt werden.
- Es gibt es im Wesentlichen drei verschiedene Typen:
- **Synchronisation** wie z.B. MPI_Barrier oder MPI_Wait
- **Kommunikation** wie z.B. MPI_Bcast, MPI_Gather oder MPI_Scatter
- **Kommunikation plus Berechnung** wie z.B. MPI_Reduce

Kommunikation

- Grundsätzlich gibt es vier wichtige Kommunikationsmuster:
- **MPI_Bcast**: Ein $root$-Prozess schickt das gleiche Datenpaket an alle anderen Prozesse in der Kommunikatorgruppe.
- Bcast gehört zur Klasse der **One-to-All** Kommunikationen.

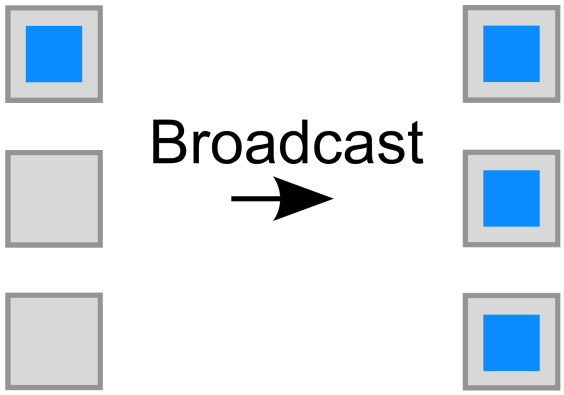


Figure 1: Quelle: http://de.wikipedia.org/wiki/Message_Passing_Interface

- **MPI_Gather**: Ein $root$-Prozess sammelt Daten von allen anderen Prozessen in der Kommunikatorgruppe ein und setzt sie hintereinander in den Empfangspuffer.

- Gather gehört zur Klasse der **All-to-One** Kommunikationen.

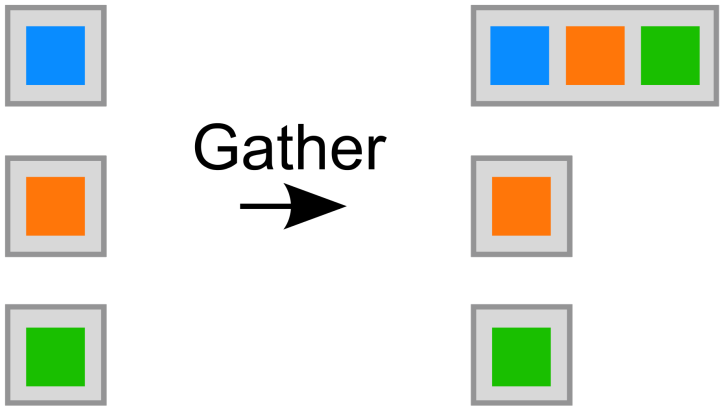


Figure 2: Quelle: http://de.wikipedia.org/wiki/Message_Passing_Interface

- **MPI_Scatter**: Ein $root$-Prozess verteilt ein Datenpaket in gleich großen Stücken auf alle anderen Prozesse in der Kommunikatorgruppe.

- Scatter gehört zur Klasse der **One-to-All** Kommunikationen.

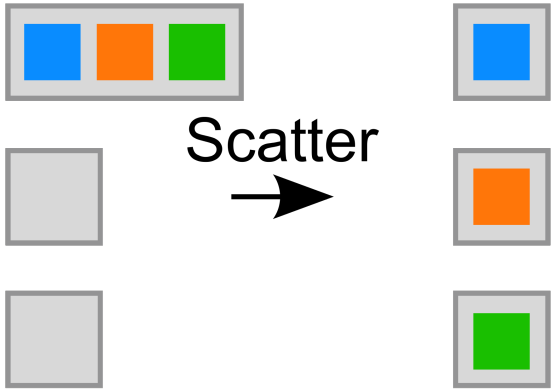


Figure 3: Quelle: http://de.wikipedia.org/wiki/Message_Passing_Interface

- **MPI_Alltoall**: Jeder Prozess in der Kommunikatorgruppe Schickt den i-ten Block eines Datenpaketes an den i-ten Prozess. Jeder Prozess sammelt von jedem Prozess ein Datenpaket und setzt Sie hintereinander.

- Alltoall gehört zur Klasse der **All-to-All** Kommunikationen.

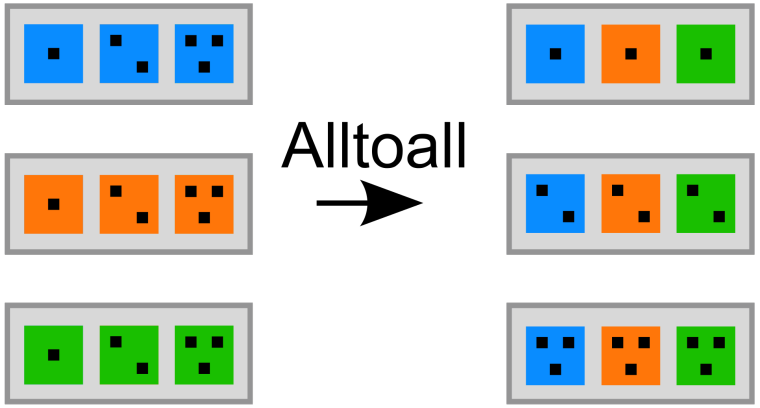
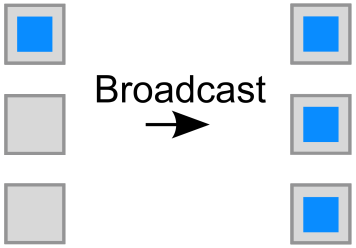


Figure 4: Quelle: http://de.wikipedia.org/wiki/Message_Passing_Interface

MPI_Bcast (Daten verteilen)



- int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- **buffer**: Startadresse des Sende- und Empfangspuffers.
- **count**: Anzahl Elemente im Puffer.
- **datatype**: zu versendender Datentyp.
- **root**: Rang des verteilenden root-Prozesses.
- **comm**: Kommunikatorgruppe.

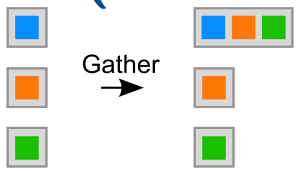
Listing 4: Beispiel MPI_Bcast

```
....
....
   //Puffer auf allen Prozessen allokieren
   int array[4]={0,0,0,0};
   //root (in diesem Falle rank=0) bekommt ein paar Werte.
    if (rank==0)
    {
        array[0]=rank*size;
        array[1]=rank*size+1;
        array[2]=rank*size+2;
        array[3]=rank*size+3;
    }

    //Verteile an alle anderen Prozesse
    MPI_Bcast(array,4, MPI_INT, 0, MPI_COMM_WORLD);


    //Ergebnis anschauen:
    printf("Thread %i haelt nach Senden array [%i,%i,%i, %i]\n"
    ,rank,array[0],array[1],array[2],array[3]);
....
....
```

- Frage: Kann man den **MPI_Bcast** auch nur mit **MPI_Send** und **MPI_Recv** ausdrücken?

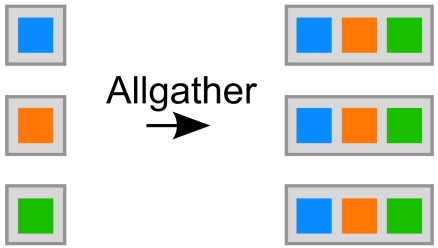
```
int array[4]={0,0,0,0};

if (rank==0)
{
    array[0]=rank*size;
    array[1]=rank*size+1;
    array[2]=rank*size+2;
    array[3]=rank*size+3;
}
//Wenn rank==0, dann senden, sonst empfangen
if (rank>0)
{
    MPI_Recv(array,4,MPI_INT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
}
else if (rank==0)
{
    //An jeden anderen Prozess senden!!
    for (i=1;i<size;i++)
    {
        MPI_Send(array,4,MPI_INT,i,tag,MPI_COMM_WORLD);
    }
}
```

MPI_Gather (Daten sammeln)



- int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
- **sendbuf / recvbuf**: Startadresse für Sendepuffer / Empfangspuffer.
- **sendcnt / recvcnt**: Anzahl der zu sendenden / empfangenden Elemente.
- **sendtype / recvtype**: Datentyp der zu sendenden / empfangenden Daten.
- **root**: Rang des $root$-Prozesses, der die Daten sammeln soll.

MPI_Allgather (Daten sammeln)



- int MPI_Allgather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)
- Input im Wesentlichen analog zum Input von MPI_Gather, bis auf:
- Kein $root$: Daten werden auf allen Prozessen gesammelt.
- Allgather gehört zur Klasse der **All-to-All** Kommunikationen.

Listing 5: Beispiel MPI_Gather und MPI_Allgather

```
int array[3]={0,0,0};
array[0]=rank*size;
array[1]=rank*size+1;
array[2]=rank*size+2;

//Empfangspuffer ausreichender Groesse
int *array_recv;
array_recv=(int *)malloc(size*3*sizeof(int));

if (modus==0)
    //Sammeln aller array in array_recv auf rank==0
    MPI_Gather(array,3, MPI_INT,array_recv,3, MPI_INT,0,MPI_COMM_WORLD);

//Sammeln auf allen Prozessen mit Gather
else if (modus==1)
{
    for (i=0;i<size;i++)
    {
        MPI_Gather(array,3, MPI_INT,array_recv,3, MPI_INT,i,MPI_COMM_WORLD);
    }
}
//Sammeln auf allen Prozessen mit Allgather
else
    MPI_Allgather(array,3, MPI_INT,array_recv,3, MPI_INT,MPI_COMM_WORLD);
```

- Die Reihenfolge in der Gather oder Allgather die Blöcke im Empfangspuffer ablegt entspricht der Nummerierung der Prozesse: Block j kommt vom Prozess mit dem Rang j.

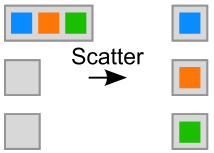
- Alle Prozesse müssen Blöcke der gleichen Größe versenden.

- Für das versenden verschiedener Blockgrößen gibt es sogenannte vektorielle Varianten: **MPI_Gatherv und MPI_Allgatherv**.

- Hier benötigt man zusätzlich ein Array der Blockgrößen und ein Array, dass die Verschiebungen der Blöcke in Relation zur Startadresse beschreibt.

- Details überlasse ich den Hörern und diversen Suchmaschinen ;-)

MPI_Scatter (Daten streuen)



- int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
- **sendbuf / recvbuf**: Startadresse für Sendepuffer / Empfangspuffer.
- **sendcnt / recvcnt**: Anzahl der zu sendenden / empfangenden Elemente.
- **sendtype / recvtype**: Datentyp der zu sendenden / empfangenden Daten.
- **root**: Rang des $root$-Prozesses, der die Daten verteilen soll.

Listing 6: Beispiel MPI_Scatter

```
....
....
    //Grosse Datenstruktur nur auf einem Prozess vorhanden
    int *array_send;
    if (rank==0)
        array_send=(int *)malloc(size_global*sizeof(int));

    if (rank==0)
    {
        for (i=0;i<size_global;i++)
        {
            array_send[i]=i;
        }
    }

    //Bestimme lokale Groessen
    int size_local;
    div_t part=div(size_global,size);

    if (part.rem!=0)
        size_local=part.quot+1;
    else
        size_local=part.quot;
```

```
    //Verteilte Arrays auf allen Prozessen vorhanden
    int *array;
    array=(int *)malloc(size_local*sizeof(int));

    //ein root-Prozess verteilt Daten aus array_send (nur auf rank==0)
    // an array (auf allen Prozessen)
    MPI_Scatter(array_send,size_local, MPI_INT,array,size_local,
                MPI_INT,0,MPI_COMM_WORLD);

    //Nach Verteilung Original nicht mehr gebraucht??
    if (rank==0)
        free(array_send);

    //Ausgabe
    printf("Thread %i haelt nach Scatter array [",rank);
    for (i=0;i<size_local-1;i++)
        printf("%i,",array[i]);
    printf("%i]\n",array[size_local-1]);
....
....
```

Bemerkungen zu Scatter

- Auch für **MPI_Scatter** gibt es die vektorielle Version **MPI_Scatterv**, die variable Blockgrößen erlaubt.

- Einige Bemerkungen zu den Blockgrößen:

- Scatter teilt die zu verteilende Struktur in Blöcke der Größe $sendcnt$ auf.

- Angenommen wir nutzen n Prozesse.

- Wenn das zu verteilende Array kleiner ist als n*$sendcnt$ gehen die "überflüssigen" Prozesse leer aus und der letzte der Daten bekommt, bekommt ggf. weniger als die Vorherigen.

- Wenn das zu verteilende Array größer ist als n*$sendcnt$ wird nicht der komplette Array verteilt, sondern nur die ersten n*$sendcnt$ Einträge.

MPI_Scatter: Darstellung verschiedener Blockgrößen

MPI_Scatter mit geeigneten Blockgrößen

MPI_Scatter mit zu großen Blockgrößen

MPI_Scatter mit zu kleinen Blockgrößen

Wann brauche ich was?

- **MPI_Scatter**: Ich habe auf einem Prozess / Core eine große Datenstruktur (z.B. durch das serielle Einlesen aus einer Datei, als Output eines seriellen Programms) und will die Daten für eine parallele Weiterverarbeitung verteilen.

- **MPI_Gather / MPI_Allgather**: Ich habe eine parallele Berechnung beendet und will nun alle Ergebnisse gesammelt und sortiert ausgeben (in Datei oder Standard-Output). Ich habe eine parallele Berechnung beendet und es ist zur Weiterverarbeitung notwendig alle Daten auf einem oder allen Prozessen zusammenzufügen.

- **MPI_Bcast**: Ich habe Daten auf einem MPI-Prozess, die ich auf allen benötige. Gerne auch genutzt um vorab Größen von Blöcken vor Gather(v) oder Scatter(v) "rum zu schicken".

Globale Kommunikation und gekoppelte Rechnung

- Will man nicht nur die Daten von verschiedenen Prozessen auf einem $root$-Prozess sammeln, sondern sie auch noch zu einer reduzierten Variable sammeln (z.B. einer Summe, einem Maximum usw.) so kann man beides kombinieren und muss weniger Speicher zur Verfügung stellen.

- Der dazu benötigte Befehl ist **MPI_Reduce**.

- Auch hier gibt es das Pendant **MPI_Allreduce**, welches das Ergebnis an alle Prozesse weiter gibt.

- Im Prinzip ist **MPI_Reduce** also eine Variante von **MPI_Gather** mit integrierter Weiterverarbeitung der Daten.

MPI_Reduce

- int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
- **sendbuf / recvbuf**: Startadresse von Sende- / Empfangspuffer.
- **count**: Anzahl zu sendenden Elemente.
- **MPI_Op**: Operation, welche die Verarbeitung der Daten bestimmt (siehe Liste auf nähster Folie).
- **root** erhält das Ergebnis.

MPI_Op

- Alle Operationen agieren **elementweise** auf dem Sendepuffer (Array).
- **MPI_SUM**: Bildet die Summe aller beteiligten Elemente.
- **MPI_PROD**: Bildet das Produkt aller beteiligten Elemente.
- **MPI_MAX**: Bestimmt das Maximum aller beteiligten Elemente.
- **MPI_MIN**: Bestimmt das Minimum aller beteiligten Elemente.
- Es gibt auch logische Verknüpfungen wie **MPI_LAND** (logisches und) oder **MPI_LOR** (logisches oder).
- Mit **MPI_Op_create** und **MPI_User_function** kann der Programmierer, falls nötig, auch selbstdefinierte Operationen erstellen.

Listing 7: Beispiel MPI_Reduce // Gauß Summe

```
int main(int argc, char *argv[])
{
    int i,size,rank;
    MPI_Init(&argc, &argv);

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

    int gauss[1]={rank+1};
    int gauss_sum[1]={0};

    //Reduce mit Summierungs-Operator
    //Ergebnis zu rank=0
    MPI_Reduce(gauss,gauss_sum,1, MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

    if (rank==0)
    {
        //Ausgabe und Vergleich
        printf("\ngauss_sum=%i\n\n",gauss_sum[0]);
        printf("Check: (size+1)*size/2=%i\n\n",size*(size+1)/2);
    }

    MPI_Finalize();
    return 0;
}
```

MPI_Allgatherv

- Gleiche Funktion wie MPI_Allgather, aber es sind unterschiedliche Blocklängen erlaubt.
- Frei wählbar ist auch der Ort, wo im Receive-Array welcher Block abgelegt werden soll.
- Alle sammelnden Prozesse müssen die Größen aller ankommenden Pakete kennen.

- int MPI_Allgatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)

- **int sendcount** ist die Paketgröße, die von dem jeweiligen Prozess gesendet wird.

- **int *recvcounts** ist hier ein Array, welches hintereinander alle Paketgrößen enthält. Also alle beteiligten **sendcount** hintereinander in einem Array.

- **int *displs** ist ein Array, das bestimmt, wo das jeweilige Paket abgelegt werden soll. Hier werden die relativen Verschiebungen zum **recvbuf** angegeben. Das von **Prozess i** gesendete Paket der Größe **recvcounts[i]** wird zum Beispiel in **recvbuf[displs[i]]** bis **recvbuf[displs[i] + recvcounts[i]-1]** abgelegt.

- Beispiel: 2 MPI-Prozesse; rank 0 schickt 2 Integer; rank 1 schickt 5 integer; beide sollen im recvbuf direkt hintereinander abgelegt werden.

- **recvcounts** sollte das Array [2, 5] sein.

- **displs** sollte das Array [0, 2] sein.

- **Tipp:** den Array **recvcounts** muss man oft vorab mit **MPI_Allgather** erstellen, falls man die zu versendenden Paketgrößen nur auf den einzelnen Prozessen kennt!

Was könnte man noch machen?

- MPI_Datentypen: MPI structure oder contiguous?
- Nutzerdefinierte Reduce-Operationen?
- Nutzerdefinierte Kommunikatoren // arbeiten mit unterschiedlichen Kommunikatoren?
- Nichtblockierende, globale Kommunikation?
- Alltoall?

Einige (Online-) Nachschlagewerke

- http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (Dokumentation)
- http://www.mcs.anl.gov/research/projects/mpi/ (Hier findet man alles inkl. einiger Links zum Thema)
- http://www.mpich.org (Homepage von mpich)
- https://computing.llnl.gov/tutorials/mpi/ (Detailliertes Tutorial / Autor: Blaise Barney, Lawrence Livermore National Laboratory)
- http://de.wikipedia.org/wiki/Message_Passing_Interface (Kurz aber in Ordnung)