Programmier-Paradigmen

Tutorium – Gruppe 4 & 8

Henning Dieterichs

MPI

1 C: Zeiger-Arithmetik, Arrays [alte Klausuraufgabe, 10 Punkte]

Welche Ausgabe erzeugt das folgende C-Programm? Begründen Sie Ihre Antwort kurz und machen Sie Ihren Lösungsweg deutlich, indem Sie unter die Programmzeilen jeweils Ihre Auswertung schreiben.

Hinweis: `printf("%i",`i`)` gibt den Zahlenwert eines Integers i aus.

```
#include <stdio.h>

int global[] = {1, 2, 3, 4, 5};

int *magic(int x[], int y) {
        printf("m");
        global[1] = *(global + y) + 3;
        return &x[y - 2];
}

int main() {
        printf("%i", *magic(&global[1], *(global + 1)));
        return 0;
}
```

Grundlegende MPI Funktionen / Konstanten

- `int **MPI_Comm_size**(MPI_Comm comm, int* size);`
- `int **MPI_Comm_rank**(MPI_Comm comm, int* my_rank);`
- `**MPI_Barrier**(MPI_COMM_WORLD);`
- `int **MPI_Send**( void* buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `int **MPI_Recv**( void* buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)`
- **MPI_COMM_WORLD**
- **MPI_ANY_SOURCE**
- **MPI_ANY_TAG**
- **MPI_INT**
- **MPI_STATUS_IGNORE**

2 MPI: Punkt-zu-Punkt-Kommunikation

Eine Firma entwickelte folgendes MPI-Programm. Es ist dafür vorgesehen, von zwei MPI-Prozessen ausgeführt zu werden. Der Quellcode ist auf unserer Webseite verfügbar[1].

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

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);

  int my_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  if (my_rank < 2) {
    int        other_rank = 1 - my_rank;
    int        tag        = 0;
    char       message[14];
    sprintf(message, "Hello, I am %d", my_rank);
    MPI_Status status;

    MPI_Send(message, strlen(message) + 1, MPI_CHAR, other_rank,
             tag, MPI_COMM_WORLD);
    MPI_Recv(message, 100, MPI_CHAR, other_rank,
             tag, MPI_COMM_WORLD, &status);
    printf("%s\n", message);
  }

  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

Wieso bleibt das Programm hängen?

Non-blocking Operations

- Non-blocking send and receive operations:

```
int MPI_Isend( void* buf, int count, MPI_Datatype type,
               int dest, int tag, MPI_Comm comm,
               MPI_Request* request)
int MPI_Irecv( void* buf, int count, MPI_Datatype type,
               int src, int tag, MPI_Comm comm,
               MPI_Request* request)
```

  - `I` stands for immediate
  - `request` is a pointer to status information about the operation
- Send and receive operations can be checked for completion

```
int MPI_Test(MPI_Request* r, int* flag, MPI_Status* s)
```

  - Non-blocking check
  - `flag` set to 1 if operation completed (0 if not yet)

```
int MPI_Wait(MPI_Request* r, MPI_Status* s)
```

  - Blocking check

3 MPI: Reduce [alte Klausuraufgabe, 14 Punkte]

1. Analysieren Sie folgenden Ausschnitt aus einem MPI-Programm unter der Annahme, dass es mit 4 Prozessen ausgeführt wird. Geben Sie in untenstehender Tabelle an, welche Werte die Puffer `sendbuffer` und `recvbuffer` nach Ausführung von `MPI_Reduce` innerhalb der jeweiligen Prozesse enthalten.

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

int sendbuffer[4];
int recvbuffer[4];

for (int i = 0; i < 4; i++) {
    sendbuffer[i] = rank + i;
}

MPI_Reduce(sendbuffer, recvbuffer, 4, MPI_INT,
           MPI_SUM, 0, MPI_COMM_WORLD);
```

2. Implementieren Sie die kollektive Operation `MPI_Reduce` für das Aufsummieren von `int`-Arrays mithilfe der folgenden MPI-Funktionen:
   `MPI_Send`, `MPI_Recv`, `MPI_Comm_size` und `MPI_Comm_rank`.

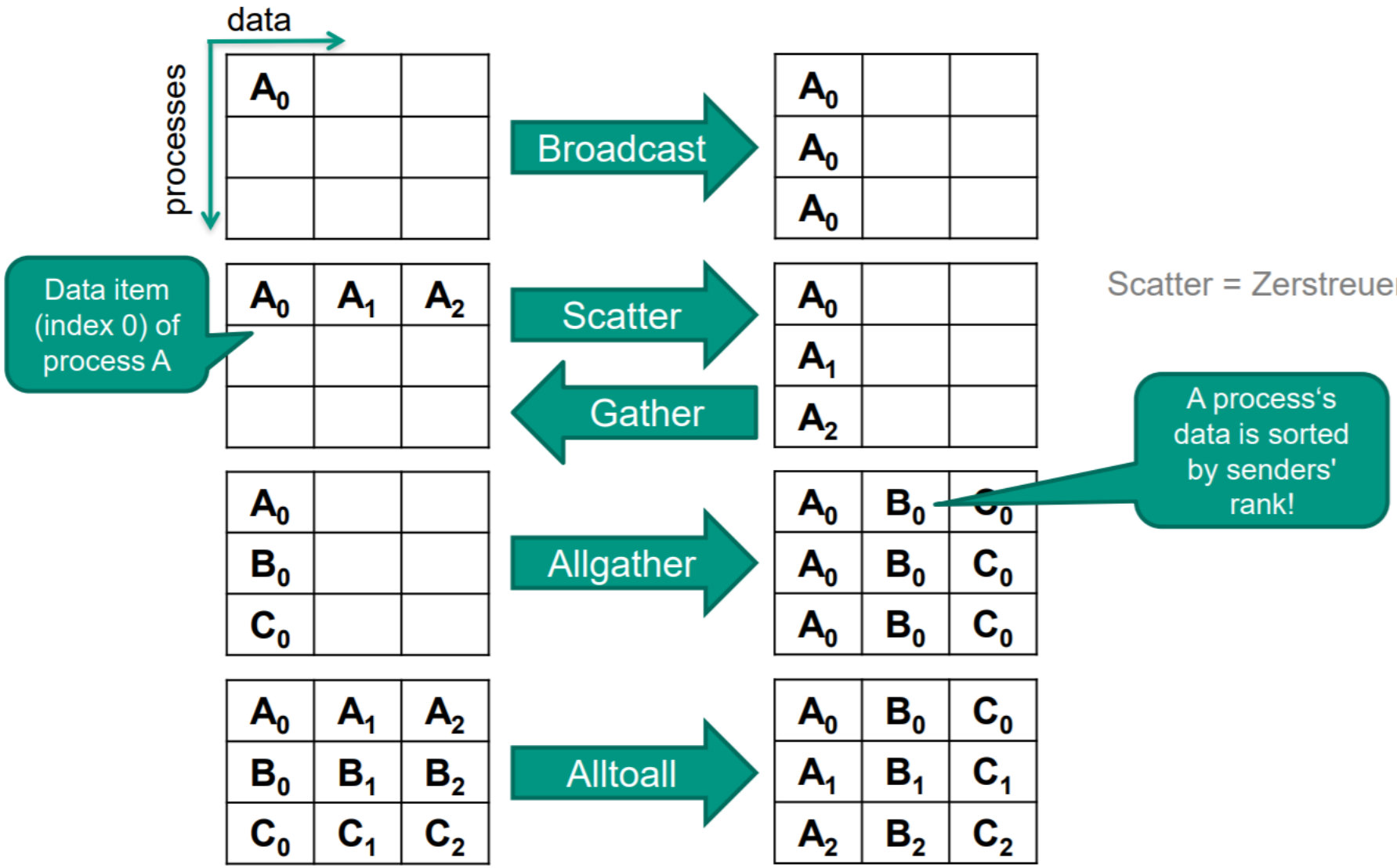
   Ergänzen Sie dazu den unten angegebenen Funktionsheader `my_int_sum_reduce` so, dass ein Aufruf der Funktion die Daten in derselben Weise verteilt, wie ein Aufruf von `MPI_Reduce` mit dem Parameterwert `MPI_SUM`.

   Hinweis: Sie dürfen davon ausgehen, dass `my_int_sum_reduce` nur mit gültigen Argumenten aufgerufen wird. Sie brauchen sich also nicht um Fehlerbehandlung aufgrund falscher Argumente zu kümmern.
   Vermeiden Sie Aufrufe von `MPI_Send`, bei denen Sender und Empfänger identisch sind, da dies zu einem Deadlock führen kann.

   Verwenden Sie für Ihre Methode die folgende Signatur:

```
void my_int_sum_reduce(int *sendbuf, int *recvbuf, int count,
                       int root, MPI_Comm comm)
```

Global Collective Operations



Weitere Operationen

- int **MPI_Bcast**(void* buffer, int count, MPI_Datatype t, int root, MPI_Comm comm)
- int **MPI_Scatter**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- int **MPI_Gather**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- int **MPI_Allgather**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
- int **MPI_Alltoall**(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
- int **MPI_Reduce**(void* sendbuf, void* recvbuf, int count, MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)
- MPI_SUM

4 MPI: Scatter & Allgather

In einem Aufruf

```
int MPI_Scatter (void *sendbuf, int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm)
```

bestimmt `sendcount`, wie viele Elemente an jeden einzelnen Prozess in `comm` gesendet werden.

Im folgenden Programmstück seien `buf_1` und `buf_2` die Anfangsadressen hinreichend großer Puffer und $k \geq 2$. Wie bei MPI-Programmen üblich, ist der Wert von `k` für alle Prozesse gleich. Die Anzahl der Prozesse, die zu `comm` gehören, ist in Variable `c` gespeichert.

```
int c;
MPI_Comm_size(comm, &c);

MPI_Scatter  (buf_1, k, MPI_INT, buf_2, k, MPI_INT, 0, comm);
MPI_Allgather(buf_2, k, MPI_INT, buf_1, k, MPI_INT,    comm);
```

5 MPI: Matrizenmultiplikation [alte Klausuraufgabe, 11.5 Punkte]

Gegeben seien zwei nxn-Matrizen (Integer-Arrays[n][n]) A und B. Die multiplikative Verknüpfung dieser zwei Matrizen sei wie folgt definiert:

$$C = A \cdot B = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix} = (c_{ij}), \tag{1}$$

wobei gilt:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{2}$$

1. Diese Matrizenmultiplikation soll mithilfe von MPI auf genau *procs* vielen Knoten des Standard-Kommunikators `MPI_COMM_WORLD` verteilt werden, so dass gilt

   $$n \ mod \ procs = 0 \tag{3}$$

   Achten Sie bei Ihrer Implementierung darauf, **wenn möglich**, nur für die Berechnung **relevante Teile** der Matrizen zu übertragen. Benutzen Sie hierfür geeignete kollektive Operationen. Geben Sie für diese die **gesamte Parameterbelegung** an. Die Einträge sollen am Ende mit einer einzigen kollektiven Operation auf **Knoten 0 in Ergebnismatrix c** zusammengeführt werden.

   Vervollständigen Sie den nachfolgenden Programmtext:

```
/* n sei die Anzahl der Zeilen und Spalten,
 * beliebig per Präprozessor vordefiniert */
void mMult(int argc, char* argv[],
    int a[n][n], int b[n][n], int c[n][n])
{
    int procs;
    int rank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // hier ergänzen

}
```