MPI

Mit dem **MPI_Init** Befehl wird MPI initalisiert. Am Ende werden alle Ressourcen mit **MPI_Finalize** freigegeben. Vor **MPI_Init** und nach **MPI_Finalize** können keine MPI calls durchgeführt werden. Außer **MPI_Initialized** and **MPI_Finalized**, welche dem Benutzer sagen, ob MPI initialisiert/fertig ist.

Mit **MPI_Abort** kann das Programm beendet werden.

```
int MPI_Init(int *argc, char ***argv); // Initate a computation
int MPI_Finalize(void); // Shut down a computation
int MPI_Finalized(int *flag); //
int MPI_Initialized(int *flag);

int MPI_Abort(MPI_Comm comm, int errorcode);
```

Zeit abfragen.

```
double MPI_Wtime(void);
double MPI_Wtick(void);
```

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

- *buf*: send buffer, welcher die Datenelement beinhaltet, die gesendet werden.
- *count*: Anzahl der Elemente, welche gesendet werden müssen.
- *datatype*: Datentyp der Einträge aus dem Send Buffer.
- *dest*: Rank des Zielprozesses
- *tag*: Message Tag zur Unterscheidung der Nachrichten
- *comm*: Communicator für die Kommunikation

```
int MPI_Recv(
    void *buf,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm comm,
```

```
    MPI_Status *status
)
```

- *buf*: Receive Buffer
- *count*: Maximale Anzahl der Elemente, die empfangen werden sollen.
- *datatype*: Datentyp der Elemente
- *source*: Rank des sendenden Prozesses.
- *tag*: Message Tag, die die Nachricht hat
- *comm*: Communicator für die Kommunikation
- *status*: Spezifiziert die Datenstruktur, welche Informationen über die Nachricht erhält.

```
int MPI_Get_count(
    MPI_Status *status,
    MPI_Datatype datatype,
    int *count_ptr
)
```

MPI_Send(...) und *MPI_Recv(...)* sind blockierende, asynchrone Operationen.

```
/*
Retourniert den rank des aufrufenden
Prozeses in die Variable rank
*/
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

```
/*
Anzahl der Prozesse
im Communicator in der Variable size
*/
int MPI_Comm_size(MPI_Comm comm, int *size);
```

Beispiel

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

int main(int argc, char *argv[]) {
    int my_rank, p, tag = 0;
    char msg[20];
    MPI_Status status;

    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank == 0) {
        strcpy(msg, "Hello ");
        MPI_Send(
            msg,
            strlen(msg)+1,
            MPI_CHAR,
            1,
            tag,
            MPI_COMM_WORLD
        );
    }

    if (my_rank == 1) {
        MPI_Recv(
            msg,
            20,
            MPI_CHAR,
            0,
            tag,
            MPI_COMM_WORLD,
            &status
        );
    }

    MPI_Finalize();
}
```

MPI garantiert, dass die Nachrichten in der Reihenfolge empfangen werden, in der sie abgesendet werden.

Ein MPI Programm wird *secure* genannt, wenn die Korrektheit des Programms nicht auf Annahmen über das System basiert. Beispiel für ein *unsecure* MPI-Programm: Sich beim Senden einer Nachricht mit MPI_Send darauf zu verlassen, dass die Nachricht aufgrund der Implementierung gepuffert wird. Dies kann zu einem Deadlock führen falls dem nicht so ist.

MPI bietet eine Kombination aus Senden und Empfangen.

```
int MPI_Sendrecv(
    void *sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    int dest,
    int sendtag,
    void *recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    int source,
    int recvtag,
    MPI_Comm comm,
```

```
    MPI_Status *status
)
```

Diese Operation ist blockierend und kombiniert senden und empfangen. Das Laufzeitsystem garantiert in diesem Fall, dass keine Deadlocks auftreten.

Non Blocking Operations

Nicht blockierende Version von MPI_Send(...).

```
int MPI_Isend(
    void *buffer,
    int count,
    MPI_Datatype type,
    int dest,
    int tag,
    MPI_Comm comm,
    MPI_Request *request
);
```

Nicht blockierende Version von MPI_Recv(...). Das retorunieren des Befehls bedeutet nicht, dass der Buffer bereits die Daten beinhaltet.

```
int MPI_Irecv(
    void *buffer,
    int count,
    MPI_Datatype type,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Request *request
);
```

Mit folgendem Befehl kann man testen, ob die nicht blockierenden Operationen erfolgreich beendet wurden. Wenn **1** retouniert wird, dann wurde die Operation erfolgreich abgeschlossen.

```
int MPI_Test(
    MPI_Request *request,
    int *flag,
    MPI_Status *status
)
```

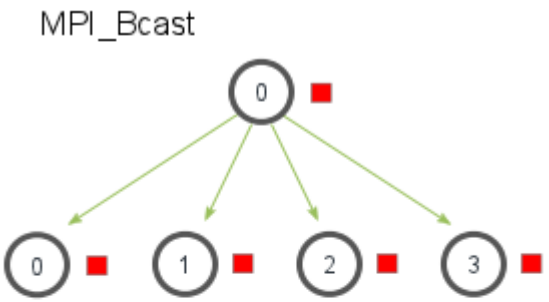
Mit folgenden Befehl kann man warten, bis eine nicht blockierende Operation fertig ist. Nach MPI_Wait(...) kann der Buffer einer Send Operation wiederverwendet werden. Und nach MPI_Wait(...) enthält der Receive Buffer die Nachricht.

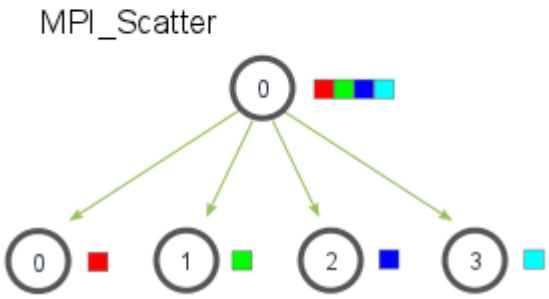
```
int MPI_Wait(
    MPI_Request *request,
    MPI_Status *status
)
```

Blockierende und nicht blockierende Operationen können beliebig kombiniert werden.

Collective Communication Operations

```
MPI_Bcast() // Broadcast operation
MPI_Reduce() // Accumulation operation
MPI_Gather() // Gather operation
MPI_Scatter() // Scatter operation
MPI_Allgather() // Multi-broadcast operation
MPI_Allreduce() // Multi-accumulation operation
MPI_Alltoall() //Total exchange
```





Broadcast Operation

Ein Prozess einer Gruppe sendet eine Nachricht an alle anderen Prozesse einer Gruppe.

MPI_Bcast(..) ist eine kollektive Operation. Jeder Prozess des Kommunikators muss die Operation *MPI_Bcast(..)* aufrufen. *MPI_Recv(...)* kann keine Nachrichten von *MPI_Bcast(...)* empfangen. Die Nachrichten werden so empfangen, wie sie gesendet wurden.

- *root*: Gibt den Prozess an der die Nachricht sendet.

```
int MPI_Bcast(
    void *message,
```

```
    int count,
    MPI_Datatype type,
    int root,
    MPI_Comm comm
)
```

Jeder Prozess stellt einen Teil der Daten zur Verfügung. Das Ergebnis beim Root Prozess ist p mal so groß, wie die gesendeten Daten.

- *sendbuf* ist der Buffer, der von jedem Prozess gesendet wird.
- *receivebuf* wird nur root-Prozess bereitgestellt.
- *count*: count * sizeof(datatype)

```
int MPI_Reduce(
    const void *sendbuf,
    void *recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm comm
)
```

Jeder Prozess stellt einen Teil der Daten zur Verfügung. Das Ergebnis beim Root Prozess ist p mal so groß, wie die gesendeten Daten.

- *sendbuf* ist der Buffer, der von jedem Prozess gesendet wird.
- *receivebuf* wird nur root-Prozess bereitgestellt.
- *count*: count der Elemente pro Prozess

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

Bei der Scatter Operation gibt der Root Prozess den anderen Prozessen Daten.

- *sendbuf* ist der Buffer von root-prozess
- *recvbuf* Dort werden die Daten reingespielt

```

```

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

Jeder Prozess stellt einen Block von Daten zur Verfügung. Es gibt keinen Root Prozess, da jeder Prozess jeden Datenblock bekommt.

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

```
int MPI_Allreduce(
    const void *sendbuf,
    void *recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm comm
)
```

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

```
int MPI_Barrier(MPI_Comm Communicator)
```

- *color*: Gibt an zu welchem neuen Kommunikator der Prozess gehört
- *key*: Gibt die Sortierung der Ranks im neuen Kommunikator an

```
int MPI_Comm_Split(
    MPI_Comm comm,
    int color,
    int key,
    MPI_Comm* newcomm
)
```

Äquivalenzen:

- MPI_Allgather(...) -> MPI_Gather(...) + MPI_Bcast(...)
- MPI_Allreduce(...) -> MPI_Reduce(...) + MPI_Bcast(...)