MPI - The Message Passing Interface

The Message Passing Interface (MPI) was first standardized in 1994. De facto standard for distributed memory machines. All Top500 machines (`http://www.top500.org`) are distributed memory machines.

The Message Passing Paradigm:

- Processors can access local memory only
- but can communicate using messages.
- In the messages data from the local memory of one processor
- is transfered to the local memory of another processor.
- Passing messages is magnitudes slower than access to local memory.

Examples (2011):

- Latency Infiniband $< 2\ \mu$s, max 15m cable
- Bandwidth Infiniband > 20 Gbit/s
- Latency Myrinet $2.6\ \mu$s,
- Bandwidth Myrinet > 10 Gbit/s,
- (Latency GigE standard commodity network $> 100\ \mu$s)
- (Latency GigE standard ommodity network < 1 Gbit/s)

The bandwidth is not much higher than in commodity networks such as GigE. But Latency is!

Advantages of MPI

Message Passing is designed for **distributed memory** machines. But it can also improve speed of shared memory programs because the programmer has explicit control over the data locality. Message Passing is then not implemented using network devices but simply as write to the shared memory $\Rightarrow$ You don't loose anything apart from the function call.

One of the most obscure errors in shared memory parallel machines is illegal write to the shared memory. In Message Passing such writes are always explicit $\Rightarrow$ May be **simpler to debug**.

MPI is implemented as a **library**, i.e. a set of **functions to call**, not as a programming language. Binding to many languages exist, notably C, C++, Python (!) and Fortran. There are open MPI implementations (mpich, openmpi, lam) but for high performance machines it is often vendor provided and vendor tuned.

MPI is **portable**, it is a standard, not an implementation;

The MPI paradigm is **a set of processes which communicate using messages**. They may run on a single processor but that is very inefficient!

Communicator

A group of processes communicating using MPI is identified using a **communicator**. The communicator containing all processes is `MPI_COMM_WORLD` (a macro defined in `mpi.h`).

Every processor within a communicator has a `rank` from 0 to `size-1`.

On the cluster you first load the module `mpich/ge/gcc/64/1.2.7`

```
[rheinba@login01 ~]$ module load mpich/ge/gcc/64/1.2.7
```

or shorter

```
[rheinba@login01 ~]$ module load mpich/ge/gcc/
```

You can get help on mpicc using

```
[rheinba@login01 ~]$ mpicc -help
```

`module avail` will tell you all available modules.

Compiling

`mpicc`

or for C++

`mpiCC`

or for Fortran

`mpif77`

MPI Example Program (C)

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

int main(int argc, char **argv)
{
  int rank, size, err;
  MPI_Init(&argc, &argv);

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

  printf("Hello world!\n");
  printf("rank=%d size=%d\n",rank,size);

  MPI_Finalize();
  return 0;
}
```

MPI Example Program (C) – Compile and Run

```
~/mpi>mpicc mpi_hello.c -ompi_hello
~/mpi>./mpi_hello

~/mpi> mpirun -np 2 ./a.out
Hello world!
rank=0 size=2
Hello world!
rank=1 size=2
```

`err` is either `MPI_SUCCESS` or not equal to `MPI_SUCCESS`.

Example Program (Fortran)

Fortran 77:

```
        program main
        include "mpif.h"

        integer rank, size, err;
        call MPI_Init(err);

        call MPI_Comm_size(MPI_COMM_WORLD,size,err)
        call MPI_Comm_rank(MPI_COMM_WORLD,rank,err)

        print *,'Hello world!'
        print *,'rank=',rank,' size=',size

        call MPI_Finalize(err);

        end
```

Example Program (C++)

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

int main(int argc, char **argv)
{
  int rank, size;
  MPI::Init(argc, argv);

  size=MPI::COMM_WORLD.Get_size();
  rank=MPI::COMM_WORLD.Get_rank();

  printf("Hello C++-world!\n");
  printf("rank=%d size=%d\n",rank,size);

  MPI::Finalize();
  return 0;
}
```

MPI Example Program (C++) – Compile and Run

```
~/mpi> mpiCC mpi_hello.cpp
~/mpi> mpirun -np 3 ./a.out

Hello C++-world!
rank=0 size=3
Hello C++-world!
rank=2 size=3
Hello C++-world!
rank=1 size=3
```

MPICH

MPICH is one of the most common implementations of MPI.

MPICH supports several different communications ”devices”, such as myrinet, infiniband and TCP. The “ch_p4” device is the TCP device. It is the usual device, unless you have special, high-speed interconnects between your machines.

MPI Broadcast

Declaration:

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_comm comm);
```

Description: Broadcasts the message in buffer of process with rank `root` to all processes ("root" of the send tree). Note that ALL processes call `MPI_Bcast()`! Processes (except root) will block until they have received the message.

Example C:

```
err=MPI_Bcast(&n, 1, MPI_INT, root, MPI_COMM_WORLD);

int buffer[4]={0,1,2,3};
int count=4;
root=0;
err=MPI_Bcast(buffer, count, MPI_INT, root, MPI_COMM_WORLD);
```

Fortran

```
call MPI_BCast(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,err)
```

Broadcast Example Program

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int rank, size, err, n;
  MPI_Init(&argc, &argv);

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

  printf("Hello world!\n");
  printf("rank=%d size=%d\n",rank,size);

  err=MPI_Barrier(MPI_COMM_WORLD);

  n=(rank+1)*4711;
  err=MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
  printf("   Received =%d\n",n);
  return 0;
}
```

Broadcast Example Output

```
~/mpi> mpicc mpi_bcast.c
~/mpi> mpirun -np 3 ./a.out

Hello world!
rank=0 size=3
   Received =4711
Hello world!
rank=2 size=3
   Received =4711
Hello world!
rank=1 size=3
   Received =4711
```

The most important `MPI_Datatype` are in C:

`MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_CHAR, MPI_LONG_DOUBLE, MPI_UNSIGNED, MPI_UNSINGED_LONG`

and in FORTRAN:

`MPI_INTEGER, MPI_REAl, MPI_DOUBLE_PRECISION, MPI_CHARACTER, MPI_COMPLEX, MPI_LOGICAL`

MPI Reduction

Declaration:

```
int MPI_Reduce(void *sendbuf, void *receivebuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_COMM_WORLD);
```

Description:
Reduces the buffers using the operation to the buffer belonging to `root`. The `receivebuf` on `root` is overwritten with the sum! Unlike Allreduce, only the `root` blocks.

Example call:

```
MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

MPI Reduction Example Program

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int rank, size, err, n, sum;
  MPI_Init(&argc, &argv);

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

  printf("Hello world!\n");
  printf("rank=%d size=%d\n",rank,size);

  err=MPI_Barrier(MPI_COMM_WORLD);
  n=rank+1;
  err=MPI_Reduce(&n,&sum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  if(rank==0)
    printf("   Received =%d\n",sum);
  return 0;
}
```

MPI Reduction Output

```
~/mpi> mpicc mpi_reduction.c -ompi_reduction
~/mpi> mpirun -np 3 mpi_reduction

Hello world!
rank=0 size=3
Hello world!
rank=2 size=3
   Received =6
Hello world!
rank=1 size=3
```

Important operations are `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_MAXLOC (also gives rank of min)`, `MPI_MINLOC`, `MPI_LAND (logical and)`, `MPI_LOR`, `MPI_LXOR`, `MPI_BAND (bitwise and)`, `MPI_BOR`, `MPI_BXOR`.

Data types for MPI Reductions

```
/* Collective operations in mpi.h */
typedef int MPI_Op;

#define MPI_MAX    (MPI_Op)(100)
#define MPI_MIN    (MPI_Op)(101)
#define MPI_SUM    (MPI_Op)(102)
#define MPI_PROD   (MPI_Op)(103)
#define MPI_LAND   (MPI_Op)(104)
#define MPI_BAND   (MPI_Op)(105)
#define MPI_LOR    (MPI_Op)(106)
#define MPI_BOR    (MPI_Op)(107)
#define MPI_LXOR   (MPI_Op)(108)
#define MPI_BXOR   (MPI_Op)(109)
#define MPI_MINLOC (MPI_Op)(110)
#define MPI_MAXLOC (MPI_Op)(111)
```

MPI Allreduce

Declaration:

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

Description:
Reduces the buffers using the operation to all ranks.

Example call:

```
MPI_Allreduce(&n,&sum,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
```

Allreduce Example Program

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int rank, size, err, n, sum;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  printf("Hello world!\n");
  printf("rank=%d size=%d\n",rank,size);
  n=rank+1;
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Reduce(&n,&sum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  printf("   Received reduce=%d\n",sum);
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Allreduce(&n,&sum,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
  printf("   Received Allreduce =%d\n",sum);
  return 0;
}
```

Allreduce Example Output

```
~/mpi> mpicc mpi_allreduction.c -ompi_allreduction.c
~/mpi> mpirun -np 3 mpi_allreduction

Hello world!
rank=0 size=3
   Received reduce=6
   Received Allreduce =6
Hello world!
rank=2 size=3
   Received reduce=1075047376
   Received Allreduce =6
Hello world!
rank=1 size=3
   Received reduce=1075047376
   Received Allreduce =6
```

Remark: Note that after the reduction only rank 0 has obtained the sum!

Note: partial sums can be computed using `MPI_Scan` with `MPI_SUM`.

Measuring Walltime

MPI has builtin functions to measure walltime (i.e. the clock on the wall as opposed to user time etc.).

Usage:

```
double starttime,endtime;

MPI_Barrier(MPI_COMM_WORLD);
starttime=MPI_Wtime();
...
MPI_Barrier(MPI_COMM_WORLD);
endtime=MPI_Wtime();
```

MPI Send

Declaration:

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

Description:

Sends `buf` from calling process to `dest`. The integer value `tag` is an ID from `0...32767` (minimum guaranteed by the standard).

Example call:

```
MPI_Send(&n,1,MPI_INT,1,112,MPI_COMM_WORLD);
```

MPI Receive

Declaration:

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

Description:
Will receive a message from `source` (or `source=MPI_ANY_SOURCE`). Will only accept messages with `tag` (or `tag=MPI_ANY_TAG`).

Example call:

```
MPI_Recv(&n,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
```

MPI Receive Example Program

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  printf("Hello world!\n");
  printf("rank=%d size=%d\n",rank,size);
  if(rank==0)
    {
    int n=4711;
    MPI_Send(&n,1,MPI_INT,1,112,MPI_COMM_WORLD);
    }

...
```

MPI Receive Example Program (continued)

```
...

  if(rank==1)
    {
    int n=0;
    MPI_Status status;
    MPI_Recv(&n,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
    printf("n=%d\n",n);
    }
  MPI_Finalize();
  return 0;
}
```

MPI Receive Example Output

```
~/mpi> mpicc mpi_sendreceive.c
~/mpi> mpirun -np 2 ./a.out

Hello world!
rank=0 size=2
Hello world!
rank=1 size=2
n=4711
```

Get MPI Version

Declaration:

```
int MPI_Get_version(int *version, int *subversion);
int MPI_Get_processor_name(char *name, int *resultlen);
```

Description:
Get Information about the MPI Version and the architecture.

Example call:

```
MPI_Get_version(&version, &subversion);
MPI_Get_processor_name(name, &len);
```

MPI AlltoAll

Declaration:

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *receivebuf, int receivecount, MPI_Datatype recvtype,
                 MPI_Comm comm);
```

Description: MPI_ALLTOALL sends a distinct message from each task to every task. The jth block of data sent from task i is received by task j and placed in the ith block of the buffer recvbuf.

Example call:

```
err=MPI_Alltoall(sendbuf, sendcount, MPI_INT,
                 receivebuf, receivecount, MPI_INT,
                 MPI_COMM_WORLD);
```

MPI Barrier

```
err=MPI_Barrier(MPI_COMM_WORLD);
```

In C:

```
mpicc mm.c -omm.c
mpirun -np 4 ./mm
```

In Fortran:

```
mpif77 mm.f -omm
mpirun -np 4 ./mm
```

In Fortran:

```
mpif90 mm.f -omm
mpirun -np 4 ./mm
```

Debugging MPI: Some MPE extensions

MPE is a library provided with MPICH.

You can use logging with MPE:
Initalize, then give numbers and names to the events. Then log using the numbers, an optional number to output and an optional string to output `MPE_Log_event(5,0,nothing");`

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

// ~/mpi> mpicc -lmpe mpi_bcast_mpe_logging.c
// ~/mpi> export MPE_LOG_FORMAT=ALOG  for processing with upshot

int main(int argc, char **argv)
{
  int rank, size, err, n;
  MPI_Init(&argc, &argv);

  MPE_Init_log();
```

```
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
n=(rank+1)*4711;

MPE_Describe_state(1,2,"Bcast", "red:vlines3");
MPE_Describe_state(3,4,"Barrier", "yellow:gray");
MPE_Describe_state(5,6,"IO", "blue:gray3");

MPE_Log_event(5,0,"nothing");
printf("Hello world!\n");
printf("rank=%d size=%d\n",rank,size);
MPE_Log_event(6,0,"nothing");

MPE_Log_event(3,1,"barrierstart");
err=MPI_Barrier(MPI_COMM_WORLD);
MPE_Log_event(4,1,"barrierend");

MPE_Log_event(1,0,"Bcast");
err=MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
MPE_Log_event(2,0,"Bcast");

printf("   Received =%d\n",n);
```

```
  MPE_Finish_log("mpelog.txt");

  MPI_Finalize();
  return 0;
}
```

The output can by analized using `upshot` or `logviewer`, a script provided with mpe, or `jumpshot` a java program. See also the man-pages in the `mpich/man` directory. Read them by `man -l MPI_Issend.3`.

Output of upshot

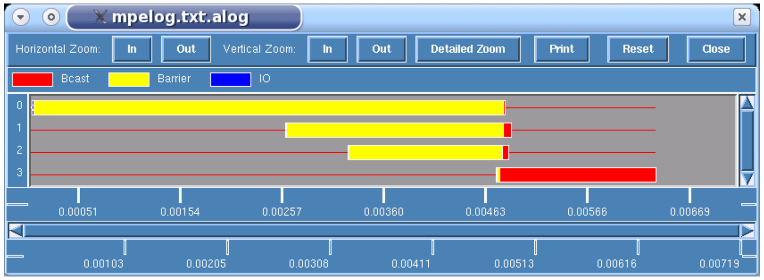

Abbildung 1: Output of upshot. Set MPE_LOG_FORMAT to "ALOG".

Note that there are also some basic graphics capabilities in MPE:

```
MPE_XGraph graph;
MPE_Open_graphics(&graph, MPI_COMM_WORLD, 0, -1, -1,
                  WINDOW_SIZE, WINDOW_SIZE,
                  MPE_GRAPH_INDEPENDENT);
MPI_Draw_point(graph,x,y,MPE_BLACK);
```

Communicators

Sometimes it is convenient to use collective operations like summing or synchronization on subgroups. For this you can create new communicators additional to `MPI_COMM_WORLD`. Here it is done by excluding the process `0` from the communicator. You extract the group from the communicator then exclude the process then create a communicator from the group. Using the communicators is possible after freeing the groups.

```
MPI_Comm world, subworld;
MPI_Group world_group, subworld_group;

int ranks[1];
ranks[0]=0;

world=MPI_COMM_WORLD;
MPI_Comm_group(world,&world_group);

MPI_Group_excl(world_group,1,ranks,&subworld_group);
MPI_Comm_create(world,subworld_group,&subworld);

MPI_Group_free(&subworld_group);
MPI_Group_free(&world_group);
```

Splitting Communicators

You can also split communicators using

```
int MPI_Comm_split(MPI_Comm oldcomm, int color, int key, MPI_Comm *newcomm);
```

The new communicator includes all the processes giving the same `color` as argument. Using `key` the sorting in the new communicator can be influenced.

We can now create the subworld from above using

```
MPI_Comm_split(world,rank==0,0,&subworld);
```

Optimization: Overlapping Communication and Computation

Declaration:

```
       int MPI_Isend( void *buf, int count, MPI_Datatype datatype,
                      int dest, int tag,
                      MPI_Comm comm, MPI_Request *request )
INPUT PARAMETERS
       buf    - initial address of send buffer (choice)
       count  - number of elements in send buffer (integer)
       datatype
              - datatype of each send buffer element (handle)
       dest   - rank of destination (integer)
       tag    - message tag (integer)
       comm   - communicator (handle)
OUTPUT PARAMETER
       request - communication request (handle)
```

Description:
Overlaps computation and communication: Use calls to `MPI_Isent` and `MPI_Irecv` and then do computations. The usage is exactly as `MPI_Send()` - only make sure not to change the `buf` before everything was sent. This function does not have a status variable.

MPI Wait

To wait for a send by `MPI_Isent` to finish use

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

```
INPUT PARAMETER
       request - request (handle)
OUTPUT PARAMETER
       status - status object (Status) .
                May be MPI_STATUS_IGNORE .
```

Status is only good for checking for errors.

MPI Waitall

To wait for all open requests use

```
int MPI_Waitall(
        int count,
        MPI_Request array_of_requests[],
        MPI_Status array_of_statuses[] )

INPUT PARAMETERS
       count  - lists length (integer)
       array_of_requests
              - array of requests (array of handles)
OUTPUT PARAMETER
       array_of_statuses
              - array of status objects (array of Status).
                May be MPI_STATUSES_IGNORE
```

MPI Test

To test if the request was send use

```
      int MPI_Test (
               MPI_Request  *request,
               int          *flag,
               MPI_Status   *status)
INPUT PARAMETER
       request
              - communication request (handle)
OUTPUT PARAMETER
       flag   - true if operation completed (logical)
       status - status object (Status).  May be MPI_STATUS_IGNORE .
```

and test for `flag`.

MPI Irecv

To receive use

```
       int MPI_Irecv( void *buf, int count, MPI_Datatype datatype,
                      int source,
                      int tag, MPI_Comm comm, MPI_Request *request )
INPUT PARAMETERS
       buf    - initial address of receive buffer (choice)
       count  - number of elements in receive buffer (integer)
       datatype
              - datatype of each receive buffer element (handle)
       source - rank of source (integer)
       tag    - message tag (integer)
       comm   - communicator (handle)
OUTPUT PARAMETER
       request
              - communication request (handle)
```

Manual pages

Use

```
rheinbach@hilbert:~/sync/mpich-1.2.5.2/man/man3> man -l MPI_Isend.3
```

to learn more. The receiver can also use `MPI_Wait()` and `MPI_Test()` to wait/test for the receive to finish.

There is also a save sent that blocks until the received has started to receive it is called `MPI_Ssend`.

Sending and receiving at the same time

Sometimes it is more economical to send and receive at the same time.

```
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 )
INPUT PARAMETERS
       sendbuf   - initial address of send buffer (choice)
       sendcount - number of elements in send buffer (integer)
       sendtype  - type of elements in send buffer (handle)
       dest      - rank of destination (integer)
       sendtag   - send tag (integer)
       recvcount - number of elements in receive buffer (integer)
       recvtype  - type of elements in receive buffer (handle)
       source    - rank of source (integer)
       recvtag   - receive tag (integer)
       comm      - communicator (handle)
OUTPUT PARAMETERS
       recvbuf   - initial address of receive buffer (choice)
       status    - status object (Status). This refers to the receive operation.
```

Gathers and Scatters: MPI Gather

Declaration:

```
int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm )
INPUT PARAMETERS
       sendbuf   - starting address of send buffer (choice)
       sendcount - number of elements in send buffer (integer)
       sendtype  - data type of send buffer elements (handle)
       recvcount - number of elements for any single receive  (inte-
                   ger, significant only at root)
       recvtype  - data  type  of recv buffer elements (significant
                   only at root) (handle)
       root      - rank of receiving process (integer)
       comm      - communicator (handle)
OUTPUT PARAMETER
       recvbuf   - address of receive  buffer  (choice,  significant
                   only at root )
```

Description: Gathers buffers from all processes in a large array of the root process.

Gathers and Scatters: MPI Gather (2)

Example call:

```
MPI_Gather (sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcount, MPI_INT,
            0, MPI_COMM_WORLD )
```

Remark: You may wonder why you have to give `recvcount` since you could always define it as the same as `sendcount`. But since `recvtype` is provided (and possibly `recvtype!=sendtype`) it is clear that you also have to provide `recvcount`. In MPI the receiver always has to know beforehand how much data it will receive.

Gather and Scatter: MPI Allgather

Gather plus a broadcast can more efficiently be handled by `MPI_Allgather`: The block of data sent from the jth process is received by every process and placed in the jth block of the buffer recvbuf.

```
NAME
       MPI_Allgather -  Gathers data from all tasks and distribute it to all
SYNOPSIS
       #include "mpi.h"
       int MPI_Allgather ( void *sendbuf, int sendcount, MPI_Datatype sendtype,
                           void *recvbuf, int recvcount, MPI_Datatype recvtype,
                          MPI_Comm comm )
INPUT PARAMETERS
       sendbuf   - starting address of send buffer (choice)
       sendcount - number of elements in send buffer (integer)
       sendtype  - data type of send buffer elements (handle)
       recvcount - number of elements received from any process (integer)
       recvtype  - data type of receive buffer elements (handle)
       comm      - communicator (handle)
OUTPUT PARAMETER
       recvbuf   - address of receive buffer (choice)
```

MPI Gatherv and MPI Allgatherv

There is also `MPI_Gatherv` and `MPI_Allgatherv`. Here you can specify the location where the an additional array `int *displs` is given with the location where to write the received data.

```
int MPI_Gatherv ( void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                  void *recvbuf, int *recvcnts, int *displs,
                  MPI_Datatype recvtype,
                  int root, MPI_Comm comm )

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

MPI Scatter (1)

The inverse operation is a scatter

```
NAME
       MPI_Scatter -  Sends data from one task to all other tasks in a group

SYNOPSIS
       #include "mpi.h"
       int MPI_Scatter (
               void *sendbuf,
               int sendcnt,
               MPI_Datatype sendtype,
               void *recvbuf,
               int recvcnt,
               MPI_Datatype recvtype,
               int root,
               MPI_Comm comm )
```

MPI Scatter (2)

```
INPUT PARAMETERS
       sendbuf    - address of send buffer (choice, significant only at root )
       sendcount  - number of elements sent to each process (integer, significant onl
       sendtype   - data type of send buffer elements (significant only at root ) (ha
       recvcount  - number of elements in receive buffer (integer)
       recvtype   - data type of receive buffer elements (handle)
       root       - rank of sending process (integer)
       comm       - communicator (handle)


OUTPUT PARAMETER
       recvbuf    - address of receive buffer (choice)
```