

Masterpraktikum - Scientific Computing, High Performance Computing

Message Passing Interface (MPI)

Thomas Auckenthaler
Wolfgang Eckhardt

Technische Universität München, Germany



Outline

- Hello World
- P2P communication
- Collective operations
- Virtual topologies and communicators

Hello World

```
#include <mpi.h>
void main(int argc, char **argv){
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello World! (rank %d of %d)", rank,
size);
    MPI_Finalize();
}
```

- compile

```
mpicc -o hello hello.c
```

- execute

```
mpirun -np number_of_processes ./hello
```

Hello World

```
#include <mpi.h>
void main(int argc, char **argv){
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello World! (rank %d of %d)", rank,
size);
    MPI_Finalize();
}
```

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
Returns the number of processes in the communicator
 - `MPI_COMM_WORLD` : Predefined standard communicator.
Includes all processes of a parallel application.
- `int MPI_Comm_rank(MPI_Comm comm, int *size)`
Returns the process number of the executing process.

Point-to-Point Communication

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

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

- Blocking operations (return when buffer can be reused)
- rank (dest/source) and tag of send- and receive-call must match
- Wildcards for receive-calls
 - MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_STATUS_IGNORE
- Messages with same destination rank do **not** overtake each other (order preservation)

MPI Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
...	
MPI_FLOAT	float
MPI_DOUBLE	double

Point-to-Point Communication

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

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

- Blocking operations (return when buffer can be reused)
- rank (dest/source) and tag of send- and receive-call must match
- Wildcards for receive-calls
 - MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_STATUS_IGNORE
- Messages with same destination rank do **not** overtake each other (order preservation)

Point-to-Point Communication

Example: ring.c

```
...
int rank, size, dest, src;
double *s_buf, *r_buf;
MPI_Status status;
...
dest = (rank + 1) % size;
src = (rank - 1 + size) % size;
MPI_Send(s_buf, 2, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
MPI_Recv(r_buf, 2, MPI_DOUBLE, src, 0, MPI_COMM_WORLD, &status);
...
```


Point-to-Point Communication

Example: ring.c

```
...
int rank, size, dest, src;
double *s_buf, *r_buf;
MPI_Status status;
...
dest = (rank + 1) % size;
src = (rank - 1 + size) % size;
MPI_Send(s_buf, 2, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
MPI_Recv(r_buf, 2, MPI_DOUBLE, src, 0, MPI_COMM_WORLD, &status);
...
```

Deadlock!

Non-blocking Communication

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

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

- Returns immediately
- Separates communication into three phases
 - (1) initiate communication
 - (2) do something else
 - (3) wait for communication to complete
- `MPI_Request`-object is used to test / wait for completion.

Non-blocking Communication

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

- Waits until pending communication is finished.

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

- Tests if pending communication is finished.

Other routines

- MPI_Waitall, MPI_Testall
- MPI_Waitany, MPI_Testany
- MPI_Waitsome, MPI_Testsome

Collective Operations

- Three types of collective operations
 - Synchronization (`MPI_Barrier`, ...)
 - Communication (`MPI_Bcast`, ...)
 - Reduction (`MPI_Allreduce`, ...)
- Must be executed by all processes of the communicator
- All collective operations are blocking operations
- MPI 3.0 will contain non-blocking collective operations

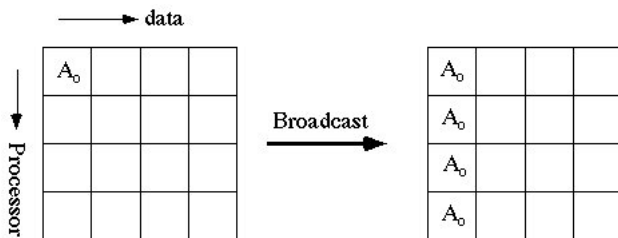
Collective Operations

```
MPI_Barrier (MPI_Comm comm) ;
```

- Blocks until **all** processes of the communicator have reached the barrier.

Collective Operations

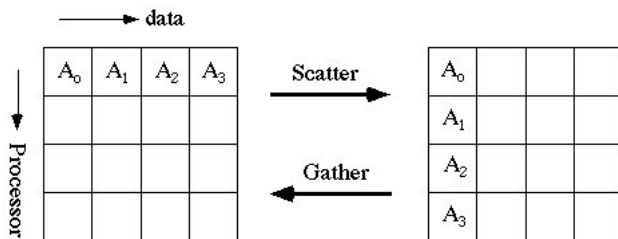
```
MPI_Bcast (void *buf, int count, MPI_Datatype dtype,  
int root, MPI_Comm comm);
```



Collective Operations

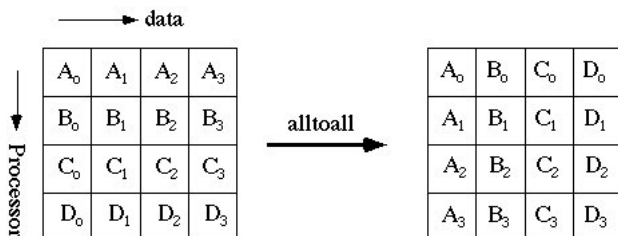
```
MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype  
sendtype, void* recvbuf, int recvcnt, MPI_Datatype  
recvtype, int root, MPI_Comm comm);
```

```
MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype  
sendtype, void *recvbuf, int recvcnt, MPI_Datatype  
recvtype, int root, MPI_Comm comm);
```



Collective Operations

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



Collective Operations

```
MPI_Reduce (void* sbuf, void* rbuf, int count,  
MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm  
comm) ;
```

- Accumulates the elements in `sbuf` and delivers the results to process `root`.
- `MPI_Op` is a Reduction Operation Handle. Possible values:
 - `MPI_MAX` (Maximum)
 - `MPI_MIN` (Minimum)
 - `MPI_SUM` (Sum)
 - `MPI_PROD` (Product)
 - `MPI_BAND` (Bitwise AND)
 - ...
- Similar routines: `MPI_Allreduce`, `MPI_Reduce_scatter`

Virtual Topologies

- Processes of a communicator (e.g. `MPI_COMM_WORLD`) can be mapped to
 - a Cartesian Topology
 - a Graph Topology
- Allow convenient process naming with **cartesian process coordinates**.
- May lead to better performance (network aware programming).

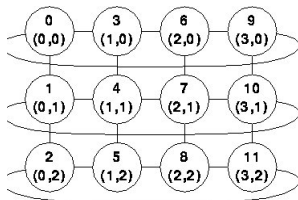
Virtual Topologies

```
MPI_Cart_create(MPI_Comm comm_old, int ndims, int
*dims, int *periods, int reorder, MPI_Comm
*comm_cart);
```

- Creates a communicator with cartesian topology

```
MPI_Cart_sub(MPI_Comm comm, int *remain_dims,
MPI_Comm *newcomm);
```

- Cuts a grid up into “slices”.



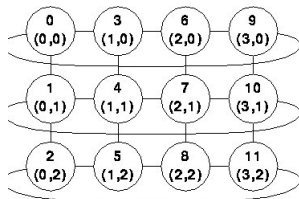
Virtual Topologies

```
MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank);
```

- Converts grid coordinates into process rank.

```
MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords);
```

- Returns the grid coordinates of process rank.



Other useful routines

```
double MPI_Wtime();
```

- Returns the elapsed time on the calling processor

Resources

- MPI 2.2 Standard
<http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf>
- List of MPI routines
http://mpi.deino.net/mpi_functions/