

Introduction to the Message Passing Interface (MPI)

F. Schornbaum

University Erlangen-Nuremberg – System Simulation

December 2014

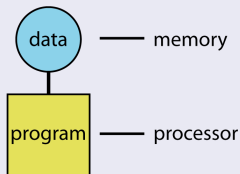


- MPI Overview
- Process model and language bindings
- Messages and point-to-point communication
- Non-blocking communication
- Derived data types
- Virtual topologies
- Collective communication

- MPI Overview
 - one program on several processors
 - work and data distribution
 - the communication network
- Process model and language bindings
- Messages and point-to-point communication
- Non-blocking communication
- Derived data types
- Virtual topologies
- Collective communication

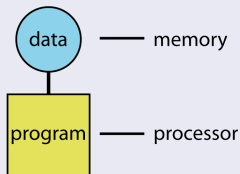
The Message-Passing Programming Paradigm

Sequential Programming Paradigm

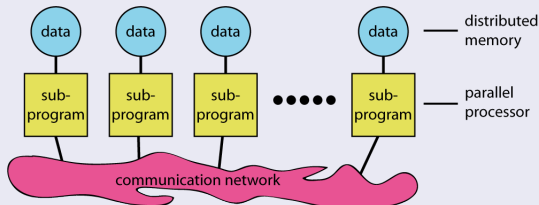


The Message-Passing Programming Paradigm

Sequential Programming Paradigm



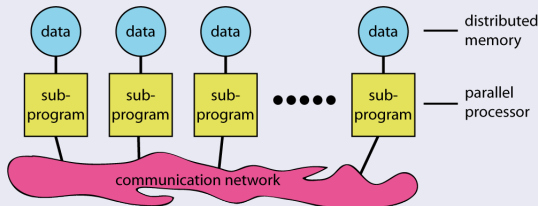
Message-Passing Programming Paradigm



The Message-Passing Programming Paradigm

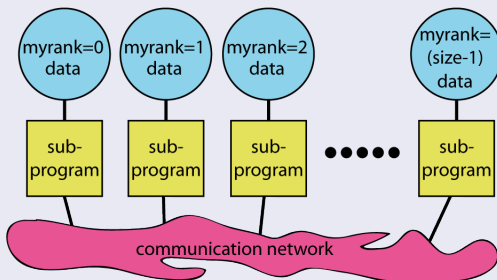
Each processor in a message passing program runs a **sub-program**

- written in a conventional sequential language, e.g., C(++) or Fortran
- typically the same on each processor (SPMD)
- the variables of each sub-program have
 - the same name
 - but different locations (distributed memory) and different data!
 - i.e., all variables are private
- communicate via special send & receive routines (**message passing**)

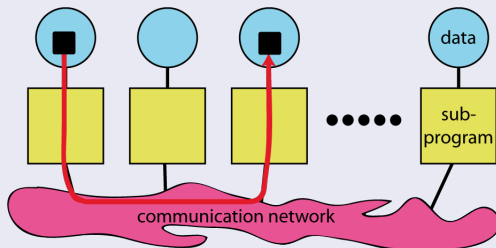


Data and Work Distribution

- the value of **myrank** is returned by special library function
- the system of **size** processes is started by special MPI initialization program (mpirun or mpiexec)
- all distribution decisions are based on **myrank**
- i.e., which process works on which data



Messages



- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:
 - sending process
 - source location
 - source data type
 - source data size
 - receiving process
 - destination location
 - destination data type
 - destination buffer size

- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
 - mail box
 - phone line
 - fax machine
 - etc.
- MPI:
 - sub-program must be linked with an MPI library
 - the total program (i.e., all sub-programs of the program) must be started with the MPI startup tool

- Messages need to have addresses to be sent to
- Addresses are similar to:
 - mail addresses
 - phone number
 - fax number
 - etc.
- MPI: addresses are ranks of the MPI processes (sub-programs)

Point-to-Point communication

- Simplest form of message passing
- One process send a message to another
- Different types of point-to-point communication:
 - non-buffered = synchronous send
 - buffered = asynchronous send

Synchronous Sends

- The sender gets an information that the message is received
- Analog to the beep or okay-sheet of a fax

Synchronous Sends

- The sender gets an information that the message is received
- Analog to the beep or okay-sheet of a fax

Buffered = Asynchronous Sends

- We only know when the message has left

Synchronous/Asynchronous Sends – Examples

Process 0

```
send    ( &a, 1, 1 );  
receive( &b, 1, 1 );
```

Process 1

```
send    ( &a, 1, 0 );  
receive( &b, 1, 0 );
```

Synchronous/Asynchronous Sends – Examples

Process 0

```
send    ( &a, 1, 1 );  
receive( &b, 1, 1 );
```

Process 1

```
send    ( &a, 1, 0 );  
receive( &b, 1, 0 );
```

Depending on the implementation of send and receive this might lead to a **deadlock!**

Synchronous/Asynchronous Sends – Examples

Process 0

```
send    ( &a, 1, 1 );  
receive( &b, 1, 1 );
```

Process 1

```
send    ( &a, 1, 0 );  
receive( &b, 1, 0 );
```

Depending on the implementation of send and receive this might lead to a **deadlock!**

Process 0

```
receive( &b, 1, 1 );  
send    ( &a, 1, 1 );
```

Process 1

```
receive( &b, 1, 0 );  
send    ( &a, 1, 0 );
```


Synchronous/Asynchronous Sends – Examples

Process 0

```
send    ( &a, 1, 1 );  
receive( &b, 1, 1 );
```

Process 1

```
send    ( &a, 1, 0 );  
receive( &b, 1, 0 );
```

Depending on the implementation of send and receive this might lead to a **deadlock!**

Process 0

```
receive( &b, 1, 1 );  
send    ( &a, 1, 1 );
```

Process 1

```
receive( &b, 1, 0 );  
send    ( &a, 1, 0 );
```

Even with buffered send this code will lead to a **deadlock!**

Blocking Operations

- Operations are local activities, e.g.,
 - sending (a message)
 - receiving (a message)
- Some operations may **block** until another process acts:
 - synchronous send operation **blocks until** receive is posted
 - receive operation **blocks until** message is sent
- Relates to the completion of an operation
- Blocking subroutine returns only when the operation has completed

Non-Blocking Operations

- Non-blocking operation: returns immediately and allow the sub-program to perform other work.
- At some later time the sub-program must **test** or **wait** for the completion of the non-blocking operation.
- All non-blocking operations must have matching wait (or test) operations. (Some system or application resources can be freed only when the non-blocking operation is completed.)
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation
- Non-blocking operations are not the same as sequential subroutine calls: the operation may continue while the application executes the next statements!

- Collective communication routines are higher level routines.
- Several processes are involved at a time
- May allow optimized internal implementations, e.g., tree based algorithms
- Can be built out of point-to-point communications

Broadcast

- A one-to-many communication

Broadcast

- A one-to-many communication

Reduction Operations

- Combine data from several processes to produce a single result

Collective Communications (cont'd)

Broadcast

- A one-to-many communication

Reduction Operations

- Combine data from several processes to produce a single result

Barriers

- Synchronize processes

Goals and Scope of MPI

- MPI's prime goals
 - to provide a message-passing interface
 - to provide source-code portability
 - to allow efficient implementations
- it also offers:
 - a great deal of functionality
 - support for heterogeneous parallel architectures
- With MPI-2:
 - important additional functionality
 - no changes to MPI-1.
- With MPI-2.1, 2.2, 3.0:
 - important additional functionality to fit on new hardware principles
 - deprecated MPI routines moved to chapter "Deprecated Functions"

- MPI Overview
- Process model and language bindings
 - starting several MPI processes
- Messages and point-to-point communication
- Non-blocking communication
- Derived data types
- Virtual topologies
- Collective communication

Compilation

```
g++ ... -L<mpi_library_path> -l<mpi_library_name>  
      -I<mpi_include_path> ...  
mpiCC ...
```

Compilation

```
g++ ... -L<mpi_library_path> -l<mpi_library_name>  
      -I<mpi_include_path> ...  
mpiCC ...
```

Header files

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

Compilation/Header Files/Function Format

Compilation

```
g++ ... -L<mpi_library_path> -l<mpi_library_name>  
      -I<mpi_include_path> ...  
mpiCC ...
```

Header files

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

MPI Function Format

```
error = MPI_Xxxxx( parameter, ... );  
MPI_Xxxxxx( parameter, ... );
```

MPI Function Format Details

- MPI_... namespace is reserved for MPI constants and routines, i.e., application routines and variable names must not begin with MPI_.
- Output arguments in C(++) are handled via passing-by-pointer:

Function definitions

```
MPI_Comm_rank( ..., int *rank );  
MPI_Recv( ..., MPI_Status *status );
```

Usage in your code

```
main( ... ) {  
    int myrank;  
    MPI_Status recv_status;  
    MPI_Comm_rank( ..., &myrank );  
    MPI_Recv( ..., &recv_status );  
}
```

Initializing MPI

- Should be the very first function call inside the `main` function
- Must be the very first MPI function call

Function definition

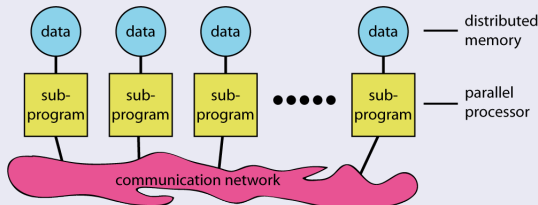
```
MPI_Init( int *argc, char ***argv );
```

Usage in your code

```
#include <mpi.h>
int main( int argc, char** argv )
{
    MPI_Init( &argc, &argv );
    ...
}
```

Starting the MPI Program

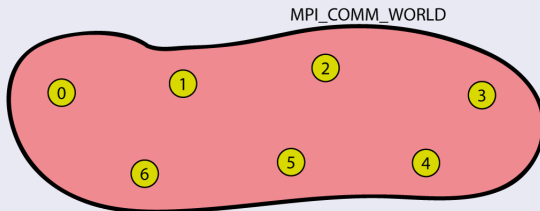
- Start mechanism is implementation dependent
- `mpirun -np number_of_processes ./executable` (most implementations)
- `mpiexec -n number_of_processes ./executable` (with MPI-2 standard)



- The parallel MPI processes exist at least after `MPI_Init` was called.

Communicator MPI_COMM_WORLD

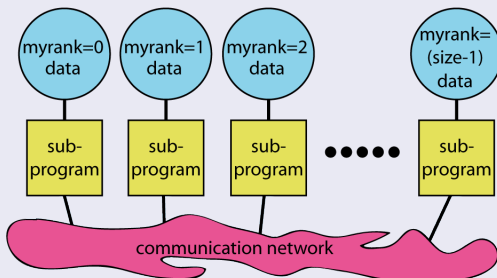
- All processes (= sub-programs) of one MPI program are combined in the communicator MPI_COMM_WORLD
- MPI_COMM_WORLD is a predefined **handle** in `mpi.h`
- Each process has its own **rank** in a communicator:
 - starting with 0
 - ending with size-1



- Handles identify MPI objects
- For the programmer, handles are
 - predefined constants in `mpi.h`
 - example: `MPI_COMM_WORLD`
 - predefined values exist only after `MPI_Init` was called
 - values returned by some MPI routines, to be stored in variables that are defined as special MPI typedefs
- Handles refer to internal MPI data structures

- The rank identifies different processes
- The rank is the basis for any work and data distribution

```
int MPI_Comm_rank( MPI_Comm comm, int *rank );
```



- How many processes are contained within a communicator?

```
int MPI_Comm_size( MPI_Comm comm, int *size );
```

```
int MPI_Finalize();
```

- **MUST** be called last by all processes
- After `MPI_Finalize`:
 - Further MPI-calls are forbidden
 - Especially re-initialization with `MPI_Init` is forbidden

Example: MPI Hello World (I)

```
#include <mpi.h>

int main( int argc, char **argv )
{
    // Definition of the variables
    int size; //The total number of processes
    int rank; //The rank/number of this process

    // MPI initialization
    MPI_Init( &argc, &argv );

    // Determining the number of CPUs and the rank for each CPU
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    ...
}
```



Example: MPI Hello World (II)

```
...

// 'Hello World' output for CPU 0
if( rank == 0 )
    cout << " Hello World" << endl;

// Output of the own rank and size of each CPU
cout << " I am CPU " << rank << " of " << size << " CPUs" << endl;

// MPI finalizations
MPI_Finalize();

return 0;
}
```

Example: MPI Hello World (III)

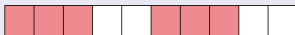
Possible output of the program:

```
I am CPU 2 of 4 CPUs  
Hello World  
I am CPU 0 of 4 CPUs  
I am CPU 3 of 4 CPUs  
I am CPU 1 of 4 CPUs
```

Note: The output of the program is non-deterministic!

- MPI Overview
- Process model and language bindings
- Messages and point-to-point communication
 - the MPI processes can communicate
- Non-blocking communication
- Derived data types
- Virtual topologies
- Collective communication

- A message contains a number of elements of some particular datatype
- MPI datatypes:
 - Basic datatype
 - Derived datatypes



- Derived datatypes can be built up from basic or derived data types
- Data type handles are used to describe the type of the data in the memory.
- Example: message with 5 integers

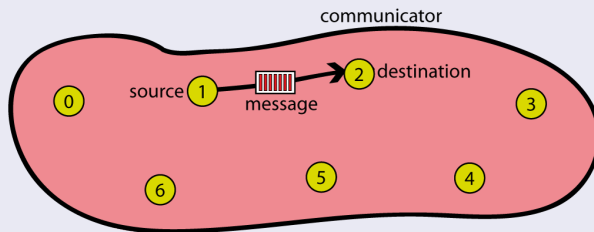
2345	654	96574	-12	7676
------	-----	-------	-----	------

MPI Basic Datatypes

MPI_CHAR	char	Treated as printable character
MPI_SHORT	signed short int	
MPI_INT	signed int	
MPI_LONG	signed long int	
MPI_LONG_LONG	signed long long	
MPI_SIGNED_CHAR	signed char	Treated as integral value
MPI_UNSIGNED_CHAR	unsigned char	Treated as integral value
MPI_UNSIGNED_SHORT	unsigned short int	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_UNSIGNED_LONG_LONG	unsigned long long	
MPI_FLOAT	float	
MPI_DOUBLE	double	
MPI_LONG_DOUBLE	long double	
MPI_BYTE		
MPI_PACKED		

Point-to-Point Communication

- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator, e.g., `MPI_COMM_WORLD`
- Processes are identified by their ranks in the communicator



Sending a Message

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

- `buf` is the starting point of the message with `count` elements
- `dest` is the rank of the destination process within the communicator `comm`
- `tag` is an additional nonnegative integer piggyback information, additionally transferred with the message
- The tag can be used by the program to distinguish different types of messages

Receiving a Message

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

- buf, count, datatype describe the receive buffer
- Receiving the message sent by process with rank source in comm
- Envelope information is returned in status
- Only messages with matching tag are received

Requirements for Point-to-Point Communications

For a communication to succeed:

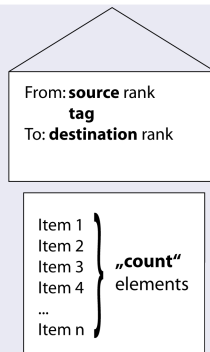
- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Message datatypes must match
- Receiver's buffer must be large enough

- Receiver can wildcard
- To receive from any source \Rightarrow `source = MPI_ANY_SOURCE`
- To receive from any tag \Rightarrow `tag = MPI_ANY_TAG`
- Actual source and tag are returned in the receiver's status parameter

Communication Envelope

Envelope information is returned from
MPI_Recv in status

```
status.MPI_SOURCE  
status.MPI_TAG  
status.MPI_ERROR  
count via MPI_Get_count()
```



```
int MPI_Get_count( MPI_Status *status,  
                  MPI_Datatype datatype, int *count );
```


Communication Modes

Send communication modes	⇒	
- synchronous send	⇒	MPI_Ssend
- buffered [asynchronous] send	⇒	MPI_Bsend
- standard send	⇒	MPI_Send
- ready send	⇒	MPI_Rsend

Receiving all modes	⇒	MPI_Recv
---------------------	---	----------

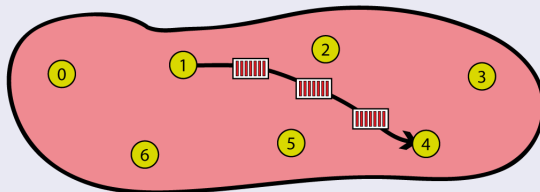
Communication Modes – Definitions

Sender mode	Definition	Notes
Synchronous send MPI_Ssend	Only completes when the receive has started	
Buffered send MPI_Bsend	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_Buffer_attach
Standard send MPI_Send	Either synchronous or buffered	Uses an internal buffer
Ready send MPI_Rsend	May be started only if the matching receive is already posted!	Highly dangerous!
Receive MPI_Recv	Completes when a message has arrived	same routine for all communication modes

- Standard send (**MPI_Send**)
 - minimal transfer time
 - may block due to synchronous mode
 - → risks with synchronous send
- Synchronous send (**MPI_Ssend**)
 - risk of deadlock
 - risk of serialization
 - risk of waiting → idle time
- Ready send (**MPI_Rsend**)
 - use **never**, except you have a 200% guarantee that Recv is already called in the current version and all future versions of your code

Message Order Preservation

- Rule for messages on the same connection, i.e., same communicator, source, and destination rank:
- **Messages do not overtake each other**
- This is true even for non-synchronous sends.



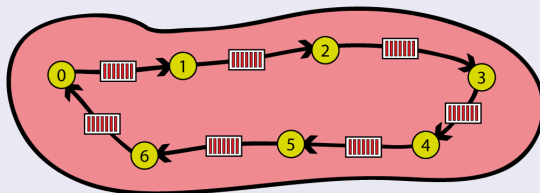
- If both receives match both messages, the order is preserved.

- MPI Overview
- Process model and language bindings
- Messages and point-to-point communication
- **Non-blocking communication**
 - - to avoid idle time and deadlocks
- Derived data types
- Virtual topologies
- Collective communication

Deadlock

- Code in each MPI process:

```
// This will block and never return, because MPI_Recv  
// cannot be called in the right-hand MPI process  
MPI_Ssend( ..., right_rank, ... );  
MPI_Recv( ..., left_rank, ... );
```



- Same problem with standard send mode (MPI_Send) if MPI implementation chooses synchronous protocol

Separate communication into three phases:

- Initiate non-blocking communication
 - returns immediately
 - routine name starting with `MPI_I...`
- Do some work (perhaps involving other communications?)
- Wait for non-blocking communication to complete

Non-Blocking Examples

Non-blocking send

MPI_Isend(...)
doing some other work
MPI_Wait(...)



Non-Blocking Examples

Non-blocking send

MPI_Isend(...)
doing some other work
MPI_Wait(...)



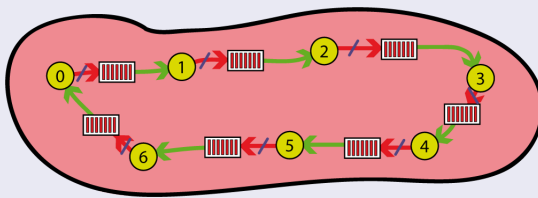
Non-blocking receive

MPI_Irecv(...)
doing some other work
MPI_Wait(...)



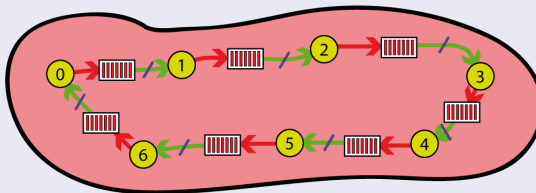
Non-Blocking Send

- Initiate non-blocking send
→ In the ring example: initiate non-blocking send to the right neighbor
- Do some work:
→ In the ring example: Receive the message from left neighbor
- Now, the message transfer can be completed
- Wait for non-blocking send to complete



Non-Blocking Receive

- Initiate non-blocking receive
In the ring example: Initiate non-blocking receive from left neighbor
- Do some work:
in the ring example: Sending the message to the right neighbor
- Now, the message transfer can be completed
- Wait for non-blocking receive to complete



Request handles

- are used for non-blocking communication
- must be stored in local variables (MPI_Request)
- the value
 - **is generated** by a non-blocking communication routine
 - **is used** (and freed) in the MPI_Wait routine

Non-Blocking Synchronous Send

```
int MPI_Issend( void* buf, int count,  
               MPI_Datatype datatype, int dest,  
               int tag, MPI_Comm comm,  
               MPI_Request *request );  
  
int MPI_Wait( MPI_Request *request, MPI_Status *status );
```

- buf must not be accessed between Issend and Wait
(In MPI-2.2, this restriction is relaxed to “must not be modified”)
- “Issend + Wait directly after Issend” is equivalent to blocking call (Ssend)
- status is not used in Issend, but in Wait (with send: nothing returned)

```
int MPI_Irecv( void* buf, int count,  
              MPI_Datatype datatype, int source,  
              int tag, MPI_Comm comm,  
              MPI_Request *request);  
  
int MPI_Wait( MPI_Request *request, MPI_Status *status );
```

- buf must not be used between Irecv and Wait

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking
- A blocking send can be used with a non-blocking receive, and vice-versa.
- Non-blocking sends can use any mode
 - standard – `MPI_Isend`
 - synchronous – `MPI_Issend`
 - buffered – `MPI_Ibsend`
 - ready – `MPI_Irsend`
- Synchronous mode affects completion, i.e., `MPI_Wait`/`MPI_Test`, not initiation, i.e., `MPI_I...`
- The non-blocking operation immediately followed by a matching wait is equivalent to the blocking operation

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

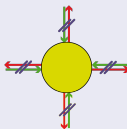
One must

- Wait or
- loop with Test until request is completed, i.e., `flag == 1`

Multiple Non-Blocking Communications

You have several request handles:

- Wait or test for completion of **one** message
`MPI_Waitany` / `MPI_Testany`
- Wait or test for completion of **all** messages
`MPI_Waitall` / `MPI_Testall` *)
- Wait or test for completion of **as many** messages as possible
`MPI_Waitsome` / `MPI_Testsome` *)



*) Each status contains an additional error field. This field is only used if `MPI_ERR_IN_STATUS` is returned (also valid for send operations)

- MPI Overview
- Process model and language bindings
- Messages and point-to-point communication
- Non-blocking communication
- **Derived data types**
 - - transfer of any combination of typed data
- Virtual topologies
- Collective communication

- Description of the memory layout of the buffer
 - for sending
 - for receiving
- Basic types
- Derived types
 - vectors
 - structs
 - others

Data Layout and the Describing Datatype Handle

```
const int count      = 2  
const int blocklength = 2  
const int stride     = 4  
MPI_Datatype columntype;
```

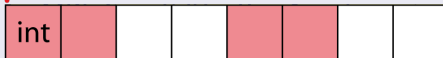
```
MPI_Type_vector( count, blocklength, stride, MPI_INT,  
                &columntype );
```

```
MPI_Type_commit( &columntype );
```

`MPI_Send(&buffer, 1, columntype, ...);`

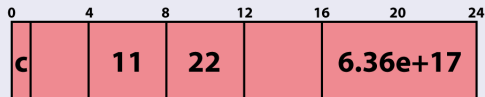
&buffer = the start address
of the data

the datatype handle
describes the data layout



Derived Data Types – Type Maps

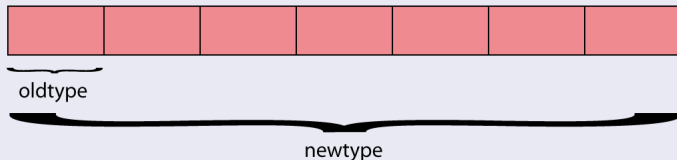
- A derived datatype is logically a pointer to a list of entries
- A derived datatype describes the memory layout of, e.g., structures, common blocks, subarrays, some variables in the memory
- Example:



basic datatype	displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

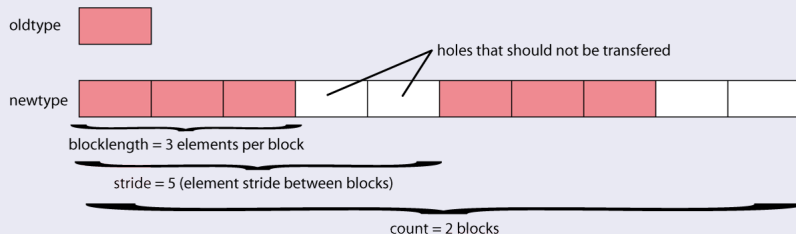
Contiguous Data

- The simplest derived datatype
- Consists of a number of contiguous items of the same datatype



```
int MPI_Type_contiguous( int count, MPI_Datatype oldtype,  
                        MPI_Datatype *newtype );
```

Vector Datatype



```
int MPI_Type_vector( int count, int blocklength,  
                    int stride, MPI_Datatype oldtype,  
                    MPI_Datatype *newtype );
```

Committing a Datatype

- Before a datatype handle is used in message passing communication, **it needs to be committed with MPI_Type_commit**
- This must be done only once (by each MPI process)

```
int MPI_Type_commit( MPI_Datatype *datatype );
```


Size and Extent of a Datatype

- Size := number of bytes that have to be transferred
- Extend := spans from first to last byte
- Basic datatypes: Size = Extend = number of bytes used by the compiler
- Derived datatypes:

```
int MPI_Type_size      (MPI_Datatype datatype, MPI_int *size);  
int MPI_Type_get_extent(MPI_Datatype datatype, MPI_int *extent); [MPI-2]  
  
int MPI_Type_extent(MPI_Datatype datatype, MPI_int *extent); [deprecated]
```

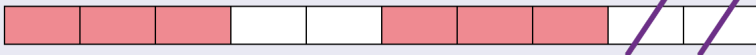
Size and Extent of a Datatype (cont'd)

Example 1

oldtype



newtype



$\text{size} := 6 * \text{sizeof}(\text{oldtype})$

$\text{extend} := 8 * \text{sizeof}(\text{oldtype})$

better visualization of newtype:



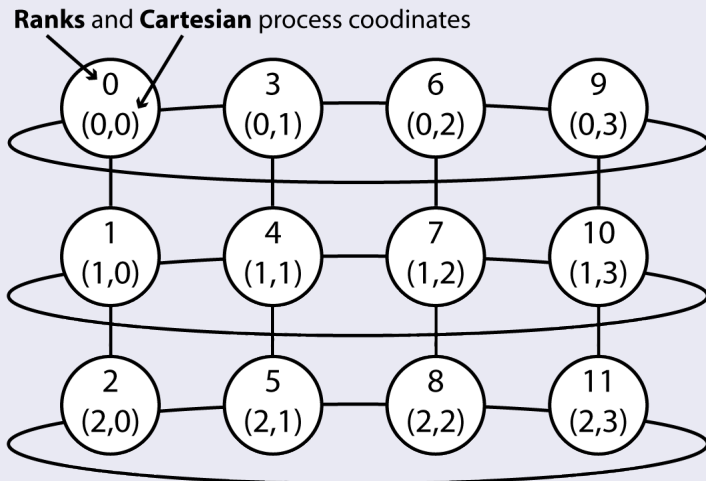
- MPI Overview
- Process model and language bindings
- Messages and point-to-point communication
- Non-blocking communication
- Derived data types
- Virtual topologies
 - - a multi-dimensional process naming scheme
- Collective communication

- Convenient process naming
- Naming scheme to fit the communication pattern
- Simplifies writing of code
- Can allow MPI to optimize communications

How to use a Virtual Topology

- Creating a topology produces a new communicator
- MPI provides mapping functions
 - to compute process ranks, based on the topology naming scheme
 - and vice versa.

Example – A 2-dimensional Cylinder



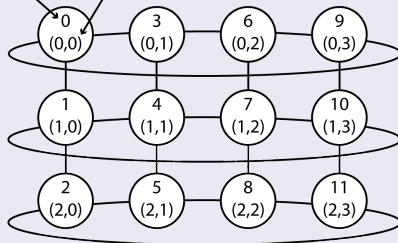
- Cartesian Topologies
 - each process is connected to its neighbor in a virtual grid
 - boundaries can be cyclic, or not
 - processes are identified by Cartesian coordinates
 - of course, communication between any two processes is still allowed
- Graph Topologies
 - general graphs
 - not covered here

Creating a Cartesian Virtual Topology

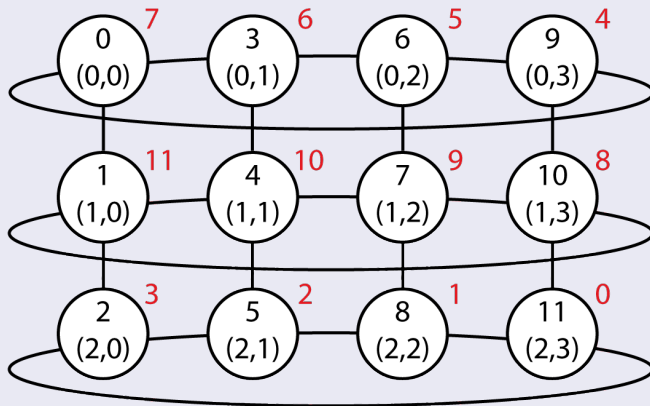
```
int MPI_Cart_create( MPI_Comm comm_old, int ndims,  
                    int *dims, int *periods, int reorder,  
                    MPI_Comm *comm_cart );
```

```
comm_old = MPI_COMM_WORLD  
ndims = 2  
dim = { 3, 4 }  
periods = { 0, 1 }  
reorder = see next slide
```

Ranks and Cartesian process coordinates



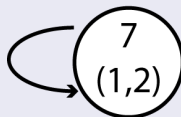
Example – A 2-dimensional Cylinder



- Ranks in `comm` and `comm_cart` may differ, if `reorder == 1`
- This reordering can allow MPI to optimize communications

Cartesian Mapping Functions

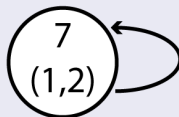
Mapping ranks to process grid coordinates



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

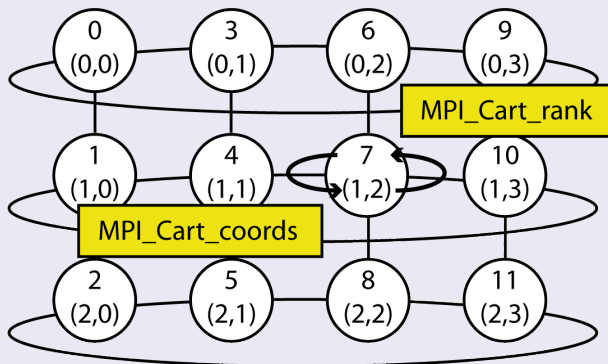
Cartesian Mapping Functions

Mapping process grid coordinates to ranks



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

Own Coordinates



Each process gets its own coordinates with

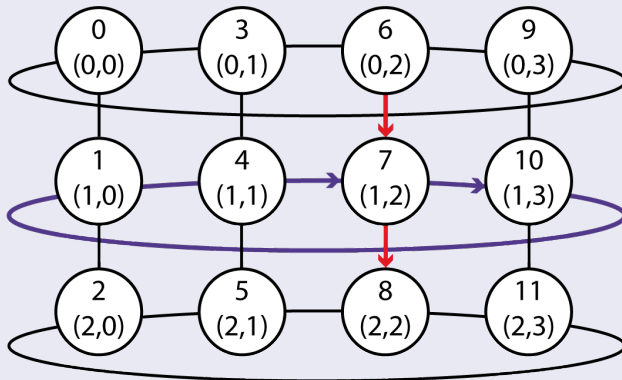
```
MPI_Comm_rank ( comm_cart, &my_rank );  
MPI_Cart_coords( comm_cart, my_rank, maxdims, my_coords );
```

- Computing ranks of neighboring processes

```
MPI_Cart_shift( MPI_Comm comm, int direction, int disp,  
                int *rank_source, int *rank_dest );
```

- Returns MPI_PROC_NULL if there is no neighbor
- MPI_PROC_NULL can be used as source or destination rank in each communication → Then, this communication will be a noop!

MPI_Cart_shift – Example



`MPI_Cart_shift(cart, direction, displace, rank_source, rank_dest);`
example of process rank=7

0	+1	6	8
1	+1	4	10

- MPI Overview
- Process model and language bindings
- Messages and point-to-point communication
- Non-blocking communication
- Derived data types
- Virtual topologies
- **Collective communication**
 - - e.g., broadcast

- Communications involving a group of processes
- Called by all processes in a communicator
- Examples:
 - Barrier synchronization
 - Broadcast, scatter, gather
 - Global sum, global maximum, etc.

Characteristics of Collective communication

- Collective action over a communicator
- All processes of the communicator must communicate, i.e., must call the collective routine
- Synchronization may or may not occur, therefore all processes must be able to start the collective routine
- All collective operations are blocking
- No tags
- Receive buffers must have exactly the same size as send buffers

Barrier Synchronization

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

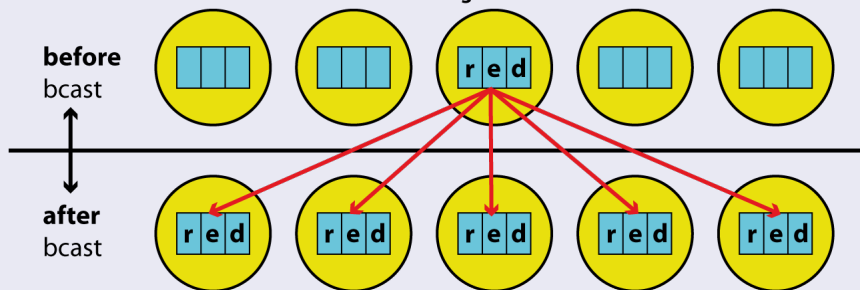
MPI_Barrier is normally never needed:

- all synchronization is done automatically by the data communication
 - a process cannot continue before it has the data that it needs
- if used for debugging
 - please guarantee, that it is removed in production
- for profiling: to separate time measurement of
 - Load imbalance of computation [MPI_Wtime(); MPI_Barrier(); MPI_Wtime();]
 - communication epochs [MPI_Wtime(); MPI_Allreduce(); ...; MPI_Wtime();]
- if used for synchronizing external communication (e.g., I/O)
 - exchanging tokens may be more efficient and scalable than a barrier on MPI_COMM_WORLD

Broadcast

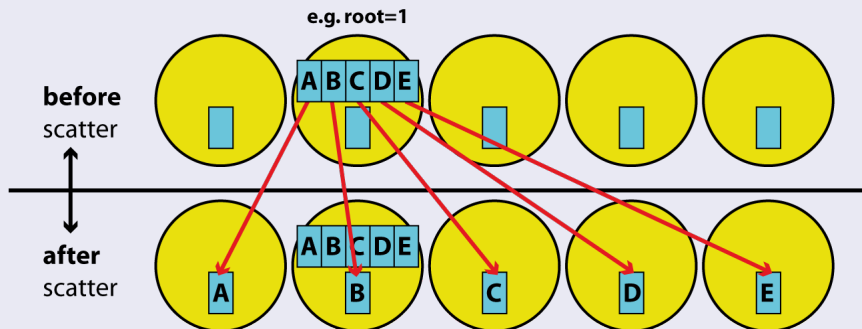
```
int MPI_Bcast( void* buffer, int count,  
               MPI_Datatype datatype,  
               int root, MPI_Comm comm );
```

e.g. root=2



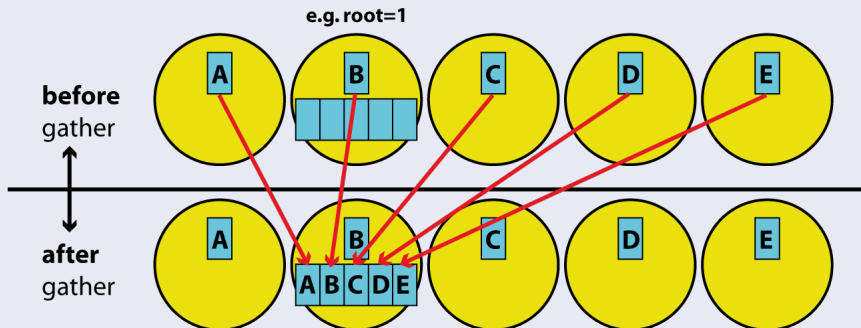
Scatter

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



Gather

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



Global Reduction Operations

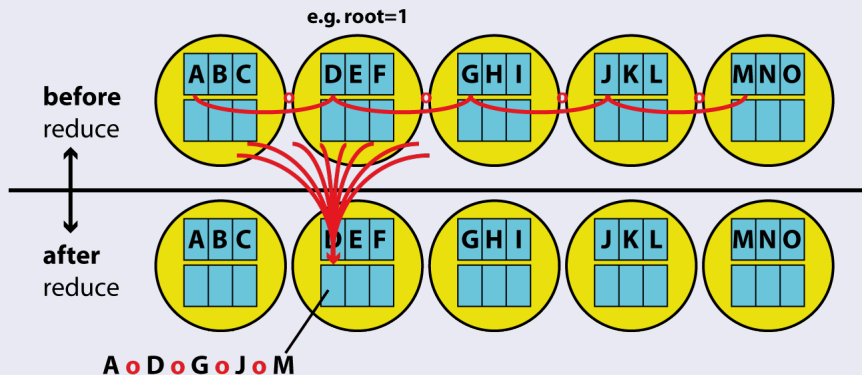
- To perform a global reduce operation across all members of a group
- $d_0 \circ d_1 \circ d_2 \circ \dots \circ d_{s-2} \circ d_{s-1}$
 - d_i = data in process rank i
 - single variable, or
 - vector
 - \circ = associative operation
 - Example:
 - global sum or product
 - global maximum or minimum
 - global user-defined operation
 - floating point rounding may depend on usage of associative law
 - $[(d_0 \circ d_1) \circ (d_2 \circ d_3)] \circ [\dots \circ (d_{s-2} \circ d_{s-1})]$
 - $(((((d_0 \circ d_1) \circ d_2) \circ d_3) \circ \dots) \circ d_{s-2}) \circ d_{s-1})$

Predefined Reduction Operation Handles

Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

MPI_Reduce

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



Example of Global Reduction

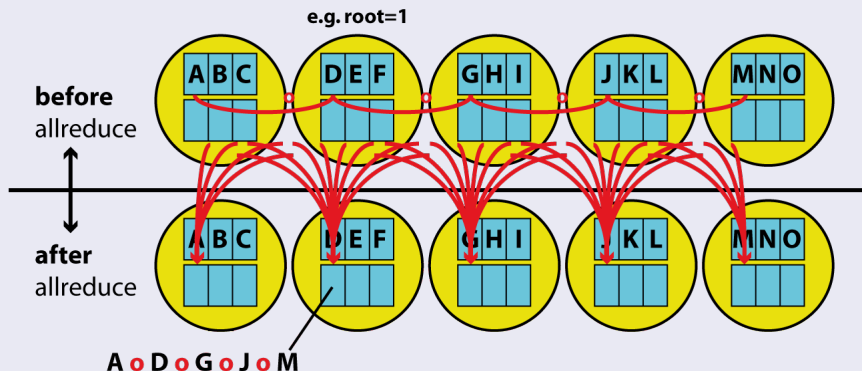
- Global integer sum
- Sum of all `inbuf` values should be returned in `resultbuf`
- `root = 0;`
`MPI_reduce(&inbuf, &resultbuf, 1, MPI_INT, MPI_SUM,`
`root, MPI_COMM_WORLD);`
- The result is only placed in `resultbuf` at the root process

Variants of Reduction Operations

- `MPI_Allreduce`
 - no root
 - returns the result in all processes
- `MPI_Reduce_scatter`
 - result vector of the reduction operation is scattered to the processes into the real result buffers
- `MPI_Scan`
 - prefix reduction
 - result at process with rank i := reduction of inbuf-values from rank 0 to rank i

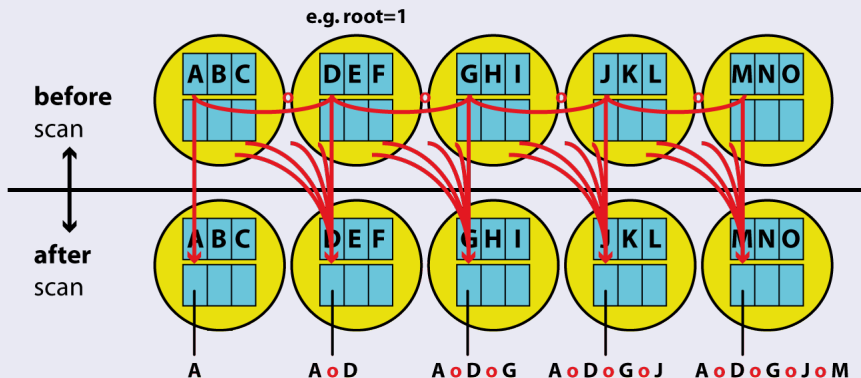
MPI_Allreduce

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



MPI_Scan

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



- MPI Overview
- Process model and language bindings
- Messages and point-to-point communication
- Non-blocking communication
- Derived data types
- Virtual topologies
- Collective communication
- **Serialization**
 - - how to order the execution of MPI processes

Question

How can MPI processes be serialized?

Solution 1

```
if( rank == 0 ) /* ... */  
MPI_Barrier( MPI_COMM_WORLD );  
if( rank == 1 ) /* ... */  
MPI_Barrier( MPI_COMM_WORLD );  
// ...
```



Solution 1

```
if( rank == 0 ) /* ... */  
MPI_Barrier( MPI_COMM_WORLD );  
if( rank == 1 ) /* ... */  
MPI_Barrier( MPI_COMM_WORLD );  
// ...
```

```
MPI_Comm_size( MPI_COMM_WORLD, &size );  
MPI_Comm_rank( MPI_COMM_WORLD, &rank );  
  
for( int i=0; i<size; ++i ) {  
    if( i == rank ) /* ... */  
        MPI_Barrier( MPI_COMM_WORLD );  
}
```


Solution 2

- Each process waits for a message from the processes with rank-1 (blocking receive)
- Do some work
- Send a message to the process with rank rank+1