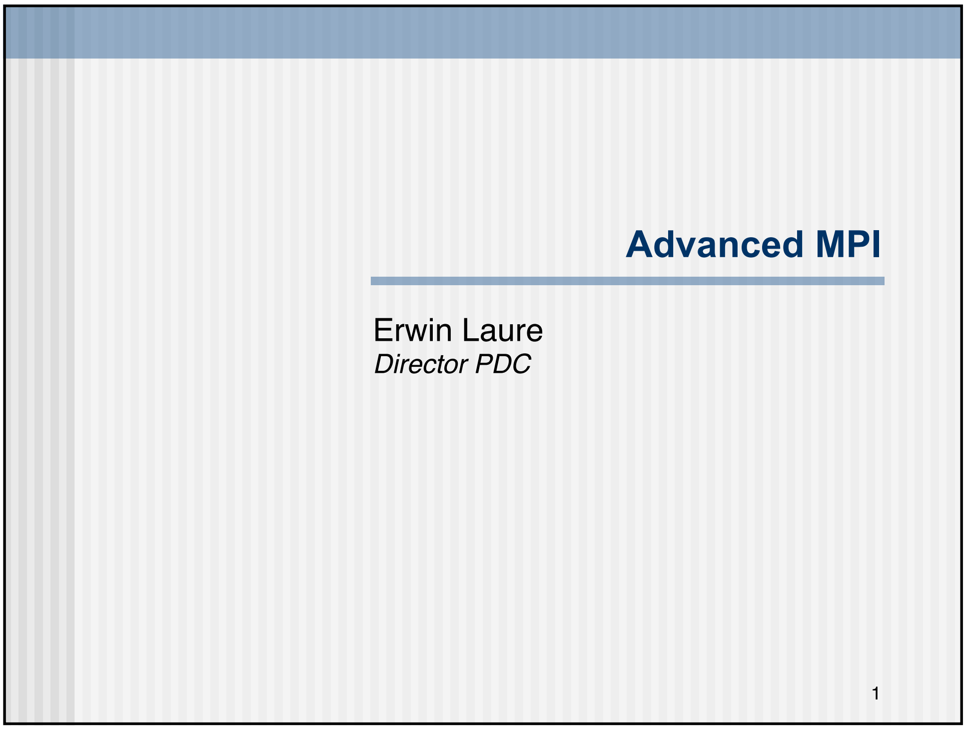
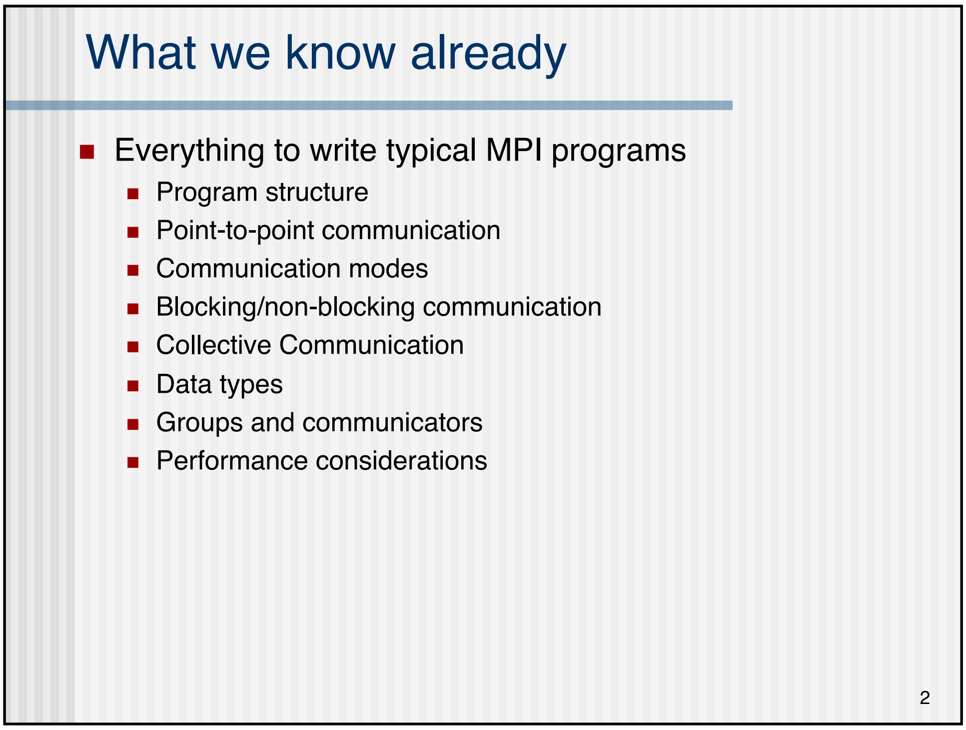


Advanced MPI

Erwin Laure
Director PDC



What we know already

- Everything to write typical MPI programs
  - Program structure
  - Point-to-point communication
  - Communication modes
  - Blocking/non-blocking communication
  - Collective Communication
  - Data types
  - Groups and communicators
  - Performance considerations

MPI provides additional, advanced features

- Virtual topologies
- MPI-IO
- One-sided communication

- Very useful in special cases – go beyond an introductory lecture

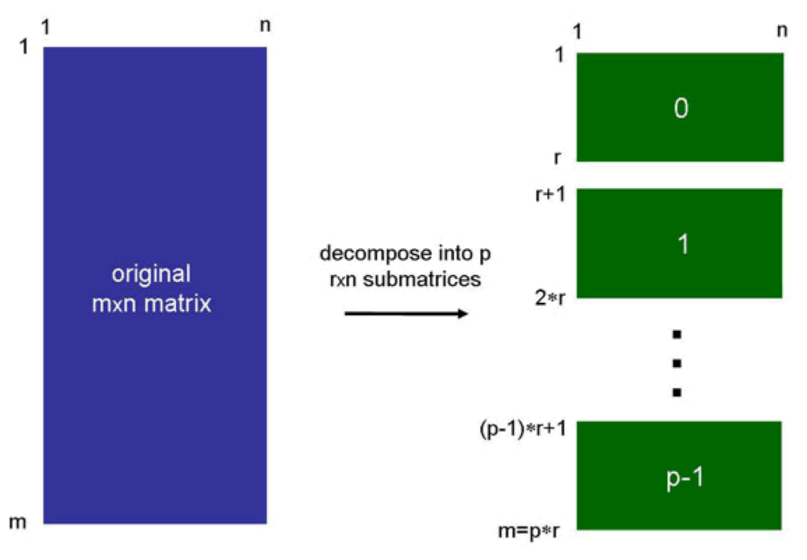
- We will touch these issues only on the surface

Virtual Topologies

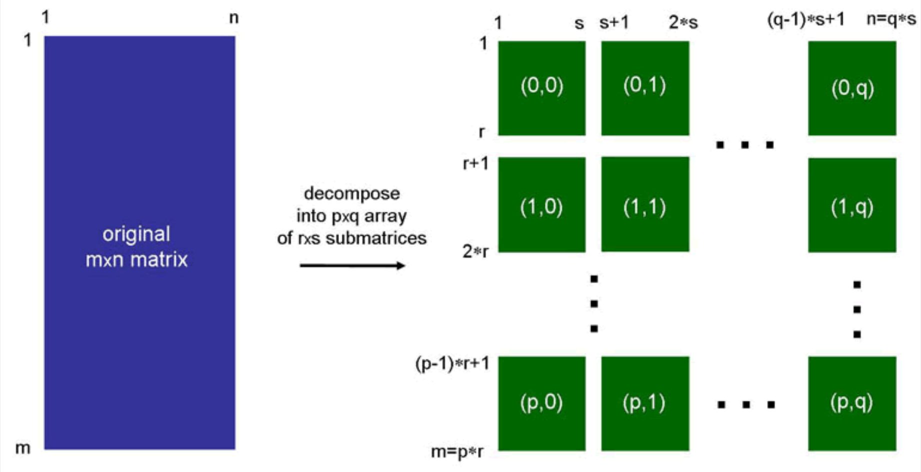
Ordering of Processes

- So far we have worked with a flat process space
  - Rank 0 … n-1
- Many application have however an inherent structure of their data
  - E.g. 2D or 3D matrices
- Likewise, the underlying network has a specific structure
  - E.g. fat tree, 3d torus, dragonfly
- Can we take advantage of this and map processes in a similar fashion?

Example – Simple (flat) topology



Example – 2D Topology



- Can still use flat process space but requires tedious and error prone mapping

MPI Virtual Topologies

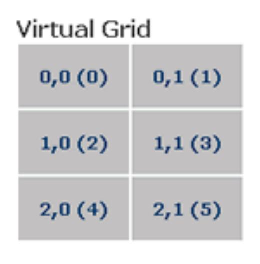
- MPI provides 2 types of virtual topologies
  - Cartesian
  - Graphs
- Cartesian topology (generalization of a grid function)
  - Each process is connected to its neighbors in a virtual grid
  - Boundaries can be cyclic (or not)
  - Processes are identified by (discrete) Cartesian coordinates
    - Eg. x,y,z
- Graph topology
  - Describe communication patterns by means of graphs
  - The most general description of communication patterns
  - Not covered here

Benefits of Virtual Topologies

- Convenient process naming
- Naming scheme to fit communication pattern
- Simplifies writing code
- Can allow MPI to optimize communications
  - Vendors can optimize mappings on their network topology
- Used in Neighborhood Collectives
  - New MPI3 feature

How do Virt. Topologies work?

- Creating a virtual topology produces a new communicator
- MPI provides mapping functions between the serial process enumeration and the virtual topology
- Mapping functions compute processor ranks based on the topology naming scheme



Virtual Grid

0,0 (0)	0,1 (1)
1,0 (2)	1,1 (3)
2,0 (4)	2,1 (5)

Main Cartesian Commands

- **`MPI_CART_CREATE`**: creates a new communicator using a Cartesian topology
- **`MPI_CART_COORDS`**: returns the corresponding Cartesian coordinates of a (linear) rank in a Cartesian communicator.
- **`MPI_CART_RANK`**: returns the corresponding process rank of the Cartesian coordinates of a Cartesian communicator.
- **`MPI_CART_SUB`**: creates new communicators for subgrids of up to (N-1) dimensions from an N-dimensional Cartesian grid.
- **`MPI_CART_SHIFT`**: finds the resulting source and destination ranks, given a shift direction and amount

MPI_CART_CREATE

```
int MPI_Cart_create(MPI_Comm old_comm, int ndims,
    int *dim_size, int *periods, int reorder,
    MPI_Comm *new_comm)
```

```
MPI_CART_CREATE(OLD_COMM, NDIMS, DIM_SIZE, PERIODS,
                REORDER, NEW_COMM, IERR)
```

`periods:` Array of size ndims specifying periodicity status of each dimension

`reorder:` whether process rank reordering by MPI is permitted

`New_comm:` Communicator handle

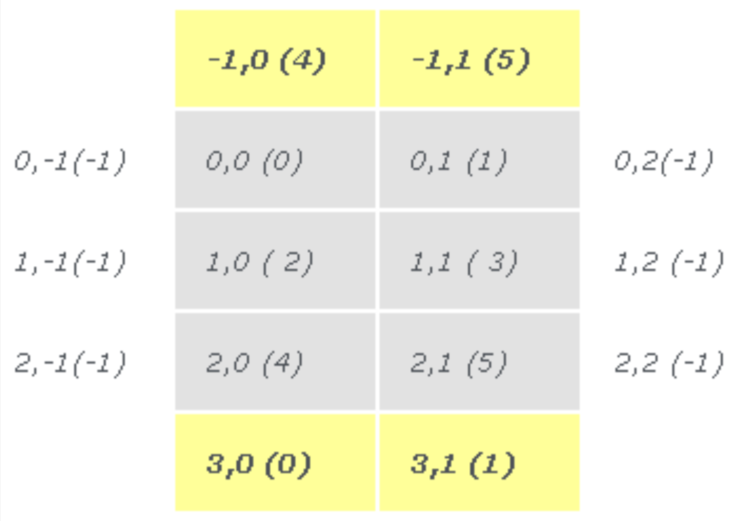
Example

```
#include "mpi.h"
MPI_Comm old_comm, new_comm;
int ndims, reorder, periods[2], dim_size[2];

old_comm = MPI_COMM_WORLD;
ndims = 2;            /*  2-D matrix/grid */
dim_size[0] = 3;      /* rows */
dim_size[1] = 2;      /* columns */
periods[0] = 1;       /* row periodic (each column forms a
                         ring) */
periods[1] = 0;       /* columns nonperiodic */
reorder = 1;          /* allows processes reordered for
                         efficiency */

MPI_Cart_create(old_comm, ndims, dim_size,
                periods, reorder, &new_comm);
```

Example Cont'd



	-1,0 (4)	-1,1 (5)	
0,-1(-1)	0,0 (0)	0,1 (1)	0,2(-1)
1,-1(-1)	1,0 (2)	1,1 (3)	1,2 (-1)
2,-1(-1)	2,0 (4)	2,1 (5)	2,2 (-1)
	3,0 (0)	3,1 (1)	

periods(0)=.true.;periods(1)=.false.

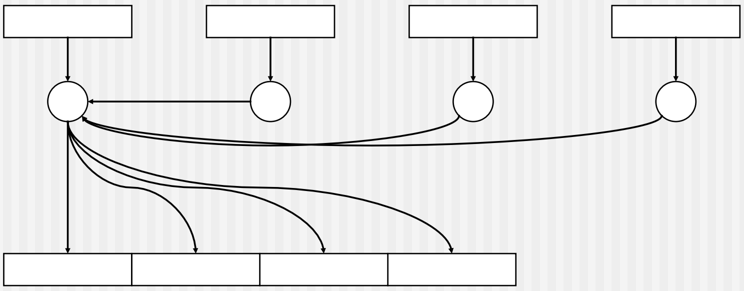
Note

- `MPI_CART_CREATE` is a collective communication function so it must be called by all processes in the group. Like other collective communication routines, `MPI_CART_CREATE` uses blocking communication. However, it is not required to be synchronized among processes in the group and hence is implementation dependent.
- If the total size of the Cartesian grid is smaller than available processes, those processes not included in the new communicator will return `MPI_COMM_NULL`.
- If the total size of the Cartesian grid is larger than available processes, the call results in error.

MPI-IO

Common Ways of Doing I/O in Parallel Programs

- Sequential I/O:
  - All processes send data to process 0, and 0 writes it to the file

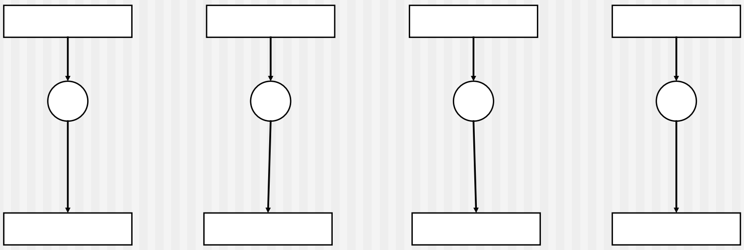


Pros and Cons of Sequential I/O

- Pros:
  - parallel machine may support I/O from only one process
    - (e.g., no common file system)
  - Some I/O libraries (e.g. HDF-4, NetCDF) not parallel
  - resulting single file is handy for **`ftp`**, **`mv`**
  - big blocks improve performance
  - short distance from original, serial code
- Cons:
  - lack of parallelism limits scalability, performance (single node bottleneck)

Another Way

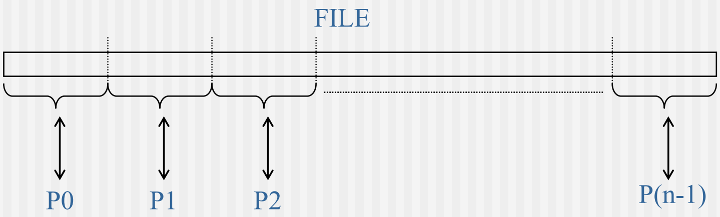
- Each process writes to a separate file



- Pros:
  - parallelism, high performance
- Cons:
  - lots of small files to manage
  - difficult to read back data from different number of processes
  - Lots of requests can make trouble to the file system

What is Parallel I/O?

- Multiple processes of a parallel program accessing data (reading or writing) from a *common* file



Why Parallel I/O?

- Non-parallel I/O is simple but
  - Poor performance (single process writes to one file) or
  - Awkward and not interoperable with other tools (each process writes a separate file)

- Parallel I/O
  - Provides high performance
  - Can provide a single file that can be used with other tools (such as visualization programs)

What is MPI-IO

- I/O interface specification for use in MPI applications
- Data model is a stream of bytes in a file
  - Same as POSIX and stdio

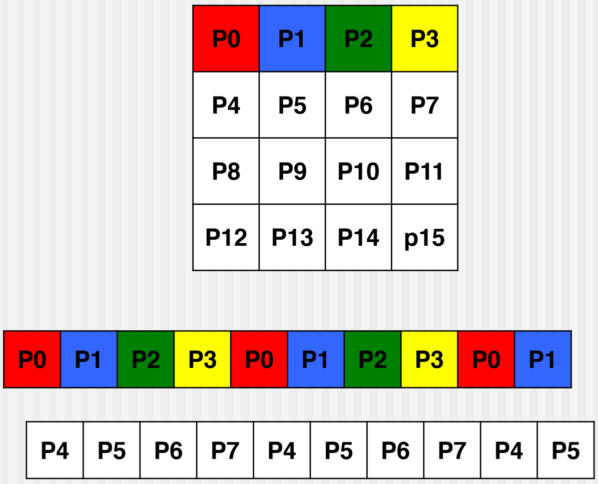
- Features
  - Noncontiguous I/O with **MPI datatypes** and file views
  - Collective I/O
  - Nonblocking I/O
  - Language bindings

MPI File Structure

- MPI defines how multiple processes access and modify data in a shared file.
- Necessary to thin this file
  - Similar to how derived d thin memory
- MPI-IO works with ple datatypes and derived datatypes
  - Derived datatypes are preferred because of performance benefits
- A view defines the current set of data, visible and accessible, from an open file.
  - Each process has its own view of the shared file that defines what data it can access.
  - A view can be changed by the user during program execution.

One big file access instead of many small ones (see e.g. http://www.mcs.anl.gov/~thakur/dtype/)

Why Derived Data Types?



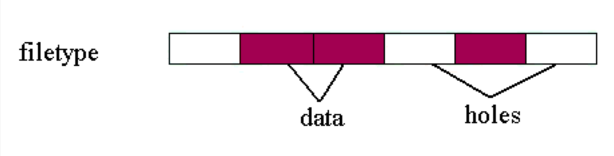
P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	p15

P0	P1	P2	P3	P0	P1	P2	P3	P0	P1

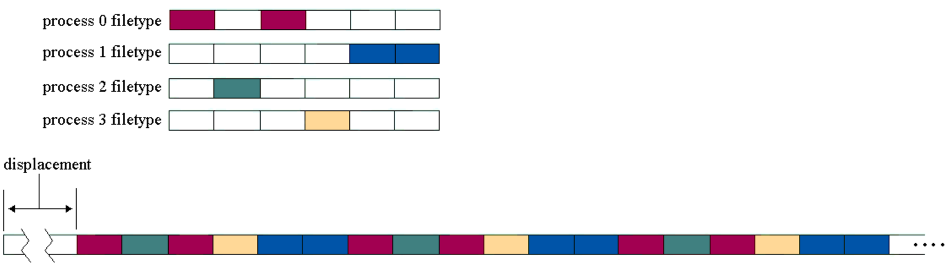
P4	P5	P6	P7	P4	P5	P6	P7	P4	P5

Essential Concepts

- Displacement
  - describes where to start in the file
- Elementary datatype (etype)
  - the type of data that is to be written or read
  - Basic or derived datatype
- Filetype
  - the pattern of how the data is partitioned in the file
  - A filetype is a defined sequence of etypes, which can have data or be considered blank



Example: File views



Simple Example

```
MPI_File fh;
MPI_Status status;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

offset

More about MPI-IO

- See for instance

[www.npaci.edu/ahm2002/ahm_ppt/**Parallel_IO_MPI_2**.ppt](www.npaci.edu/ahm2002/ahm_ppt/Parallel_IO_MPI_2.ppt)
Rajeev Thakur. Mathematics and Computer Science Division. Argonne National Laboratory

One-sided Communication

Recap: Point-to-point Communication

- Both sender and receiver must issue matching MPI calls
  - Depending on buffering semantics may require handshake
- Sometimes it is difficult to know in advance when messages have to be sent/received and what characteristics these messages have
  - Could solve such situations with extra control messages
    - Requires polling, introduces overhead, and is cumbersome
- MPI provides Remote Memory Access (RMA), or one-sided communication
  - Allows one process to specify all communication parameters for both the sender and receiver

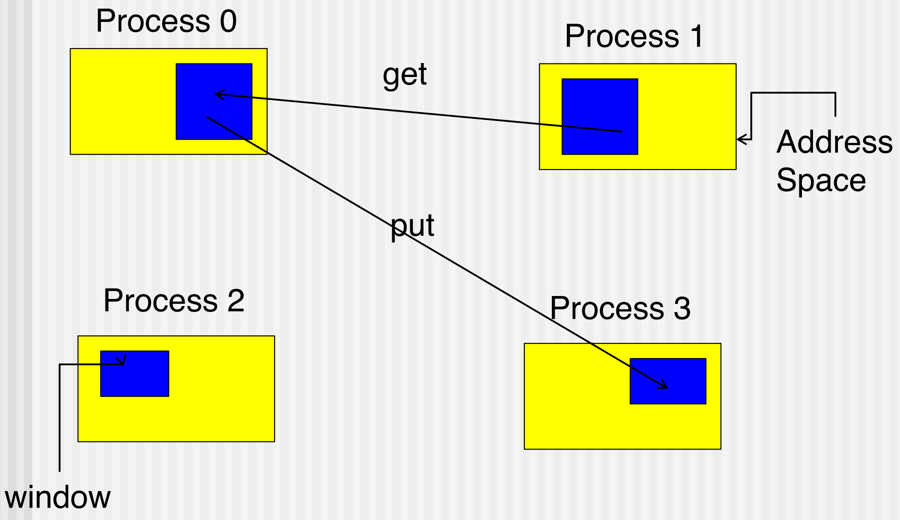
One-sided Communication

- Communication and Synchronization are separated
- Allows remote processes to
  - Write into local memory (**put**)
  - Read local memory (**get**)
- Accessible memory areas are called “windows”
- Communication can happen without synchronization
- Access to windows is synchronized

Looks a bit like shared-memory programming?

- In fact, tries to bring the advantages of shared-memory programming to MPI programs
- Effective implementation needs shared memory or hardware support for RDMA
  - Available e.g. in infiniband or Cray networks
- Need synchronization to ensure correct behavior
  - Same issues as in shared-memory programming
  - MPI provides **window objects** for synchronization
- How to implement synchronization is a great optimization field

Window Objects



Main Commands

- **`MPI_Win_create`** exposes local memory to RMA operation by other processes in a communicator
  - Collective operation
  - Creates window object
- **`MPI_Win_free`** deallocates window object
- **`MPI_Put`** moves data from local memory to remote memory
- **`MPI_Get`** retrieves data from remote memory into local memory
- **`MPI_Accumulate`** updates remote memory using local values
- Data movement operations are non-blocking
- **Subsequent synchronization on window object needed to ensure operation is complete**

Advantages of one-sided communication

- Can do multiple data transfers with a single synchronization operation
- Bypass tag matching
  - effectively precomputed as part of remote offset
- Some irregular communication patterns can be more economically expressed
- Can be significantly faster than send/receive on systems with hardware support for remote memory access, such as shared memory systems
  - **BUT**: can also be significantly slower depending on synchronization need and access patterns!

Synchronization

- `Put/Get/Accumulate` are non-blocking
  - Subsequent synchronization on window object is needed to ensure operations are complete
- MPI_Win_fence is used to synchronize access to windows
  - Should be called before and after RMA
  - Similar to a barrier in shared memory

```
Process 0               Process 1
MPI_Win_fence(win)      MPI_Win_fence(win)

MPI_Put
MPI_Put

MPI_Win_fence(win)      MPI_Win_fence(win)
```

New Modes in MPI-3

- PSCW Synchronization

- `MPI_Win_post(MPI_Group group, int assert, MPI_Win win)`
  - Start exposure
- `MPI_Win_start(MPI_Group group, int assert, MPI_Win win)`
  - Start access (may wait for post)
- `MPI_Win_complete(MPI_Win win)`
  - Finish access (origin only)
- `MPI_Win_wait(MPI_Win win)`
  - Wait for completion (at target)

- As asynchronous as possible

Other MPI-3 Features

- Lock-based synchronization
  - Locks window for access by one or all ranks

- Flush
  - Complete all outstanding operations at target and/or origin

- Request-based put and get (Rput, Rget)
  - Returns a request handle that can be tested for completion

Summary

- One-sided communication provides convenient means for irregular applications
- Communication can be more efficient with proper hardware support
- Great care needs to be put on (efficient) synchronization

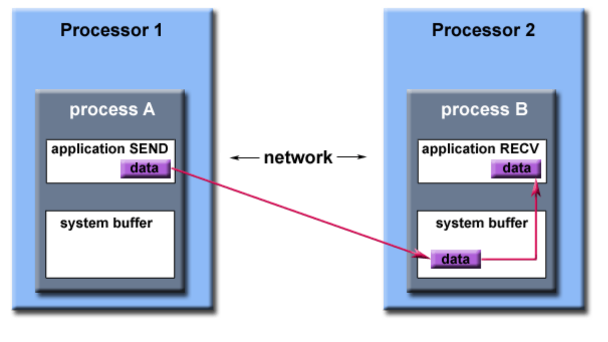
Summary

Recap: Basic MPI Concepts

- Message **buffers** described by address, data type, and count
- Processes identified by their **ranks**
- **Communicators** identifying communication contexts/groups

What is not specified

- Certain aspects are not specified in the MPI standard but left as implementation detail:
  - Process startup (how to start an MPI program)
    - All what happens before `MPI_Init` is executed
  - Richer error codes are allowed
  - Message buffering



Path of a message buffered at the receiving process

Basic Send/Receive Commands

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

```
MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, IERR)
```

Buffer
Count
Datatype
} Body

Destination
Tag
Communicator
} Envelope

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

```
MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM,
STATUS, IERR)
```

Wildcards

- Instead of specifying everything in the envelope explicitly, wildcards can be used for sender and tag:
  `MPI_ANY_SOURCE` and `MPI_ANY_TAG`
- Actual source and tag are stored in `STATUS` variable

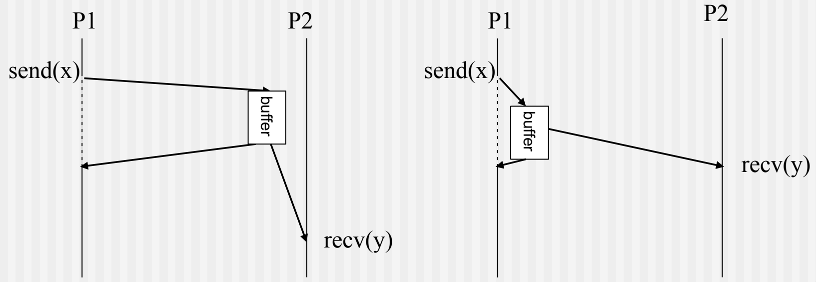
C:

```
MPI_Status status;
MPI_Recv(b, 100, MPI_DOUBLE,
         MPI_ANY_SOURCE, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status );

source = status.MPI_SOURCE;
tag = status.MPI_TAG;
```

A Word on Buffering

- MPI implementations typically use (**internal**) message buffers
  - Sending process can safely modify the sent data once it is copied into the buffer, irrespectively of status of receiving process
  - Receiving process can buffer incoming messages even if no (user space) receiving buffer is provided, yet
  - Buffers can be on both sides



Note

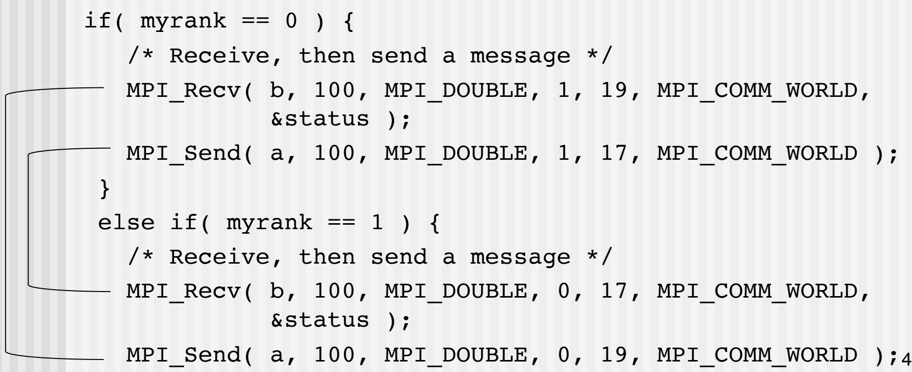
This system buffer is **DIFFERENT** to the message buffer you specify in the `MPI_Send` or `MPI_Recv` calls!

Blocking and Completion

- Both `MPI_Send` and `MPI_Recv` are blocking
  - They program only continues after they are completed
- The command is completed once it is safe to (re)use the data
  - `MPI_Recv`: data has been fully received
  - `MPI_Send`: can be completed even if no non-local action has been taking place. WHY?
  - Once data is copied into a send buffer `MPI_Send` can complete

Deadlocks

- Deadlocks are common (and hard to debug) errors in message passing programs
- A deadlock occurs when two (or more) processes wait on the progress of each other:



```
if( myrank == 0 ) {
   /* Receive, then send a message */
   MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD,
             &status );
   MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
 }
 else if( myrank == 1 ) {
   /* Receive, then send a message */
   MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
             &status );
   MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
```

Help to avoid Deadlocks Cont’d

- Careful message ordering
  - Always a good idea!

- Buffered communication
  - But comes with (quite substantial) overhead

- Non-blocking calls

Non-blocking Communication

- For all send/receive calls there is a non-blocking equivalent named `I(x)send/Irecv`

- Non-blocking calls will return immediately irrespectively of the send/receive status
  - They actually only **initiate** the action
  - Actual sending/receiving of messages will be handled internally in the MPI implementation
  - Calls return a handle that allows to check the progress of sending/receiving

- Blocking and non-blocking calls can be intermixed
  - A blocking receive can match a non-blocking send and vice-versa.

Completion of non-blocking send/receives

```
int MPI_Wait( MPI_Request *request, MPI_Status
*status );
MPI_WAIT(REQUEST, STATUS, IERR )
```

- `MPI_Wait` is blocking and will only return when the message has been sent/received
  - After `MPI_Wait` returns it is safe to access the data again

```
int MPI_Test( MPI_Request *request, int *flag,
              MPI_Status *status );
MPI_TEST(REQUEST, FLAG, STATUS, IERR)
```

- MPI_Test returns immediately
  - Status of request is returned in flag (true for done, false when still ongoing)

Collective Communication Cont'd

- Communication involving all processes in a **group** (i.e. a **communicator**)
  - MPI-3 defines "neighborhood collectives"
- All processes in a group **MUST** participate to the collective operation
- No tag mechanism, only order of program execution
  - Remember that MPI messages cannot overtake another one
- Until MPI-2 all collective routines were only blocking
  - With the standard completion semantics of blocking communication – thus no guarantee there is a full synchronization
  - MPI-3 introduced non-blocking collectives
    - Important difference to non-blocking p2p: no matching with non-blocking collectives!

List of Collective Routines

- Barrier synchronization across all processes.
- Broadcast from one process to all other processes
- Global reduction operations such as sum, min, max or user-defined reductions
- Gather data from all processes to one process
- Scatter data from one process to all processes
- All-to-all exchange of data
- Scan across all processes

Take a deeper look

- Usage of data types
  - So far we used the pre-defined data types; what if we need to deal with more complex structures?
- Usage of communicators
  - How to group processes in individual groups
- Improving Communication Performance
  - Aka how to speed up programs

Performance Considerations

- Simple and effective performance model:
  - More parameters == slower
- **contig < vector < index < struct**
- Some (most) MPIs are inconsistent
  - But this rule is portable
- Advice to users:
  - Try datatype “compression” bottom-up

Loss of performance

- Transfer time = latency + message length/bandwidth + synchronization time
- You cannot do much about bandwidth but
- Reduce latency
  - Combine many small into a single large message
  - Hide communication with computation
- Reduce message length
  - Only communicate what is absolutely needed
- Avoid synchronization

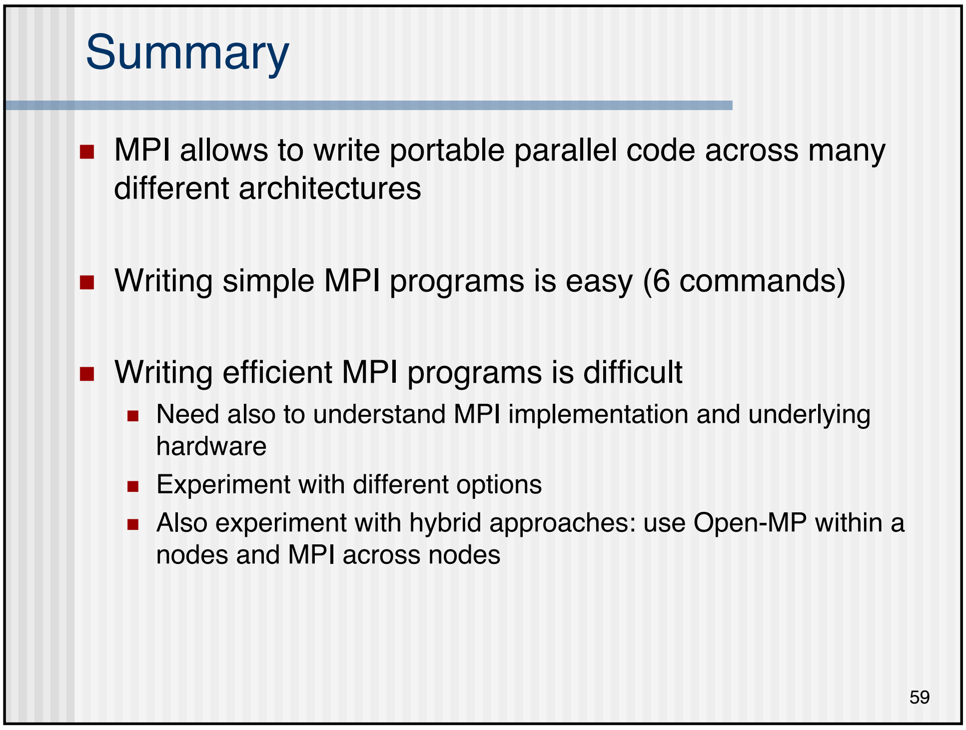
And finally …

- The top MPI Errors according to

Advanced MPI: I/O and One-Sided Communication, presented at SC2005, by William Gropp, Rusty Lusk, Rob Ross, and Rajeev Thakur
http://www.mcs.anl.gov/research/projects/mpi/tutorial/advmpi/sc2005-advmpi.pdf)

Top MPI Errors

- Fortran: missing ierr argument
- Fortran: missing MPI_STATUS_SIZE on status
- Fortran: Using integers where MPI_OFFSET_KIND or MPI_ADDRESS_KIND integers are required (particularly in I/O)
- Fortran 90: Using array sections to nonblocking routines (e.g., MPI_Isend)
- All: MPI_Bcast not called collectively (e.g., sender bcasts, receivers use MPI_Recv)
- All: Failure to wait (or test for completion) on MPI_Request
- All: Reusing buffers on nonblocking operations
- All: Using a single process for all file I/O
- All: Using MPI_Pack/Unpack instead of Datatypes
- All: Unsafe use of blocking sends/receives
- All: Using MPI_COMM_WORLD instead of comm in libraries
- All: Not understanding implementation performance settings
- All: Failing to install and use the MPI implementation according to its documentation.



Summary

- MPI allows to write portable parallel code across many different architectures
- Writing simple MPI programs is easy (6 commands)
- Writing efficient MPI programs is difficult
    - Need also to understand MPI implementation and underlying hardware
    - Experiment with different options
    - Also experiment with hybrid approaches: use Open-MP within a nodes and MPI across nodes