Parallel programming

MPI

FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

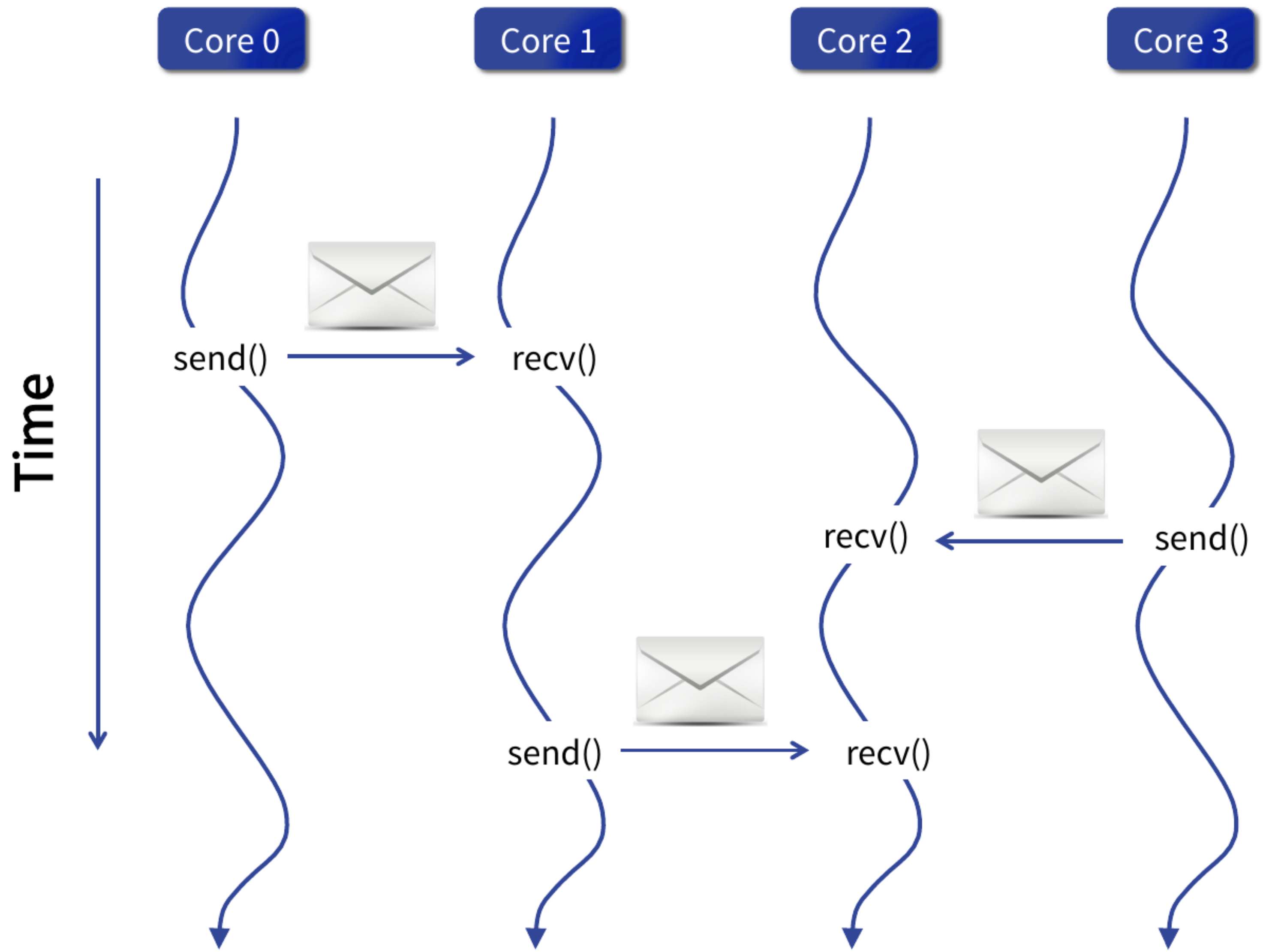
Distributed memory

- Each unit has its **own memory space**
- If a unit needs data in some other memory space, **explicit communication** (often through network) is required
  - Point-to-point and collective communication model
- Cluster computing

MPI

- MPI: **Message passing interface**
- All processes run the **same program**.
- Processes have assigned a **rank** (i.e., identification of the process).
- Based on the rank, processes can differ in an execution.
- Processes communicate by **sending and receiving** messages through **communicator**.
- Message passing:
  - Data transfer requires cooperative operations to be performed by each process.
  - For example, a send operation must have a matching receive operation.

Communication example



MPI implementations

- OpenMPI
  - Open source
  - Project founded in 2003 after intense discussion between multiple open source MPI implementations.
  - Developed by a consortium of research, academic, and industry partners
- MPICH
  - Open source
  - Reference implementation of the latest MPI standard
- Intel MPI
  - Proprietary
- MS MPI, MVAPICH ...

MPI installation

- MPI compilers not part of GCC, needs to be installed and loaded separately
- Linux
  - Fedora
    ```
    dnf install openmpi
    module load mpi/openmpi-x86_64
    ```
  - Ubuntu
    ```
    apt install libopenmpi-dev
    ```
- MacOS
  ```
  brew install openmpi
  ```
- Windows
  - MinGW: see https://www.math.ucla.edu/~wotaoyin/windows_coding.html (the link for MS mpi sdk does not work, use https://www.microsoft.com/en-us/download/details.aspx?id=52981)
  - Visual Studio + Intel compiler, see https://software.intel.com/en-us/mpi-developer-guide-windows-configuring-a-visual-studio-project

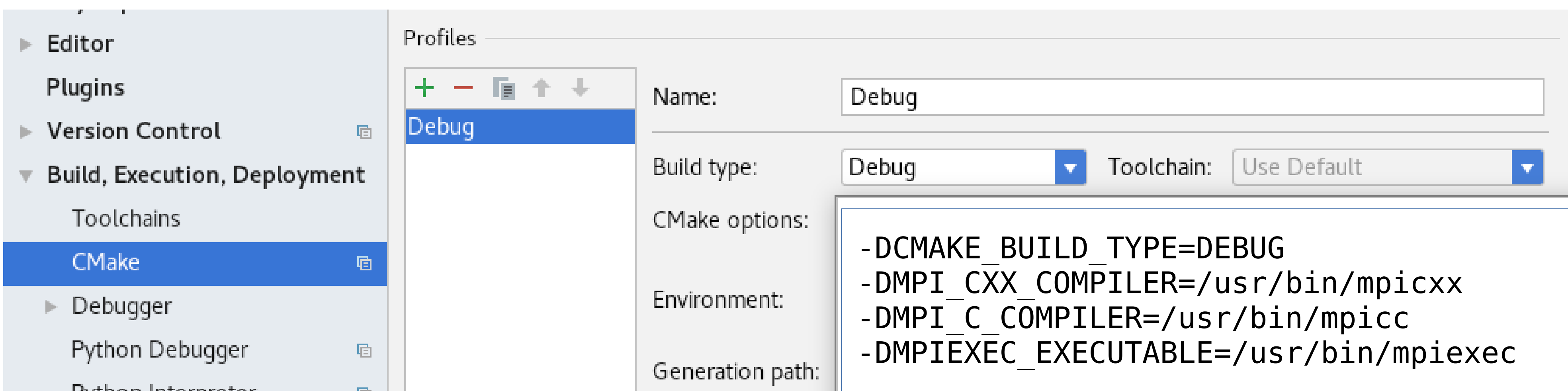
Compilation - CMake

```
cmake_minimum_required(VERSION 3.5)
project(MyProject)

find_package(MPI)
include_directories(${MPI_INCLUDE_PATH})

add_executable(Program Program.cpp)
target_compile_options(Program PRIVATE ${MPI_CXX_COMPILE_FLAGS})
target_link_libraries(Program ${MPI_CXX_LIBRARIES} ${MPI_CXX_LINK_FLAGS})
```

- CLion setup (use **whereis** command to locate paths in your operating system)



Basic MPI operations

- `#include <mpi.h>`
  - Include header file with MPI functions.
- Almost all MPI functions return an integer representing the error code (see the documentation of each function for the error codes)
- **int MPI_Init(int ****argc***, **char ******argv***)**
  - Initializes MPI runtime environment and process the arguments (trim the MPI arguments/options from argument list)
- **int MPI_Finalize()**
  - Terminates MPI execution environment.
- **int MPI_Comm_size(MPI_Comm** ***comm***, **int ****size***)**
  - Queries the ***size*** of the group associated with communicator ***comm***
  - **MPI_COMM_WORLD**: default communicator grouping all the processes
- **int MPI_Comm_rank(MPI_Comm** ***comm***, **int ****rank***)**
  - Queries the ***rank*** (identifier) of the process in communicator ***comm***. Rank is a value from 0 to ***size***.

Hello world

HelloWorld.cpp

Running MPI programs

- **`mpiexec -np 4 -f hostfile PROGRAM ARGS`**
  - **`np`** – number of used processes
  - **`hostfile`** – file with a list of hosts on which to launch MPI processes (for cluster computing)
  - **`PROGRAM`** – program to run
  - **`ARGS`** – arguments for program
- This will run **`PROGRAM`** using 4 processes of the cluster.
- All nodes run the same program.
- The processes may be running on different cores of the same node
- Visual Studio: to change the arguments passed to **`mpiexec`**, change Project Properties → Debugging → Command arguments
  - First start of an MPI program will ask you for your username+passwords.

Send a message

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

- ***buf*** - buffer which contains the data elements to be sent
- ***count*** - number of elements to be sent
- ***datatype*** - data type of elements
- ***dest*** - rank of the target process
- ***tag*** - message tag which can be used by the receiver to distinguish between different messages from the same sender
- ***comm*** - communicator used for the communication

Datatypes in MPI

MPI data type	C data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	special data type for packing
MPI_BYTE	single byte value

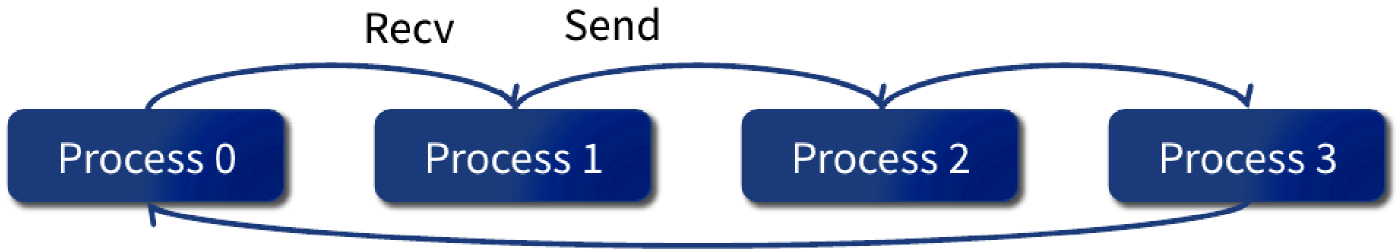
Receive a message

- ```
  int MPI_Recv(void *buf,
               int count,
               MPI_Datatype datatype,
               int source,
               int tag,
               MPI_Comm comm,
               MPI_Status *status)
  ```
- Same as before. New arguments:
  - ***`count`*** – maximal number of elements to be received
  - ***`source`*** – rank of the source process
  - ***`status`***
    - data structure that contains information (rank of the sender, tag of the message, actual number of received elements) about the message that was received
    - can be used by functions as **`MPI_Get_count`** (returns number of elements in msg.)
    - If not needed, **`MPI_STATUS_IGNORE`** can be used instead
- Each **Send** must be matched with a corresponding **Recv**.
- Messages are delivered in the order in which they have been sent.


  ```

Simultaneous Send and receive

-
```
int MPI_Sendrecv(const 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)
```
- Parameters: Combination of parameters for **Send** and **Receive**
- Performs send and receive at the same time.
- Useful for data exchange and ring communication:



Example 1 – Send me a secret code

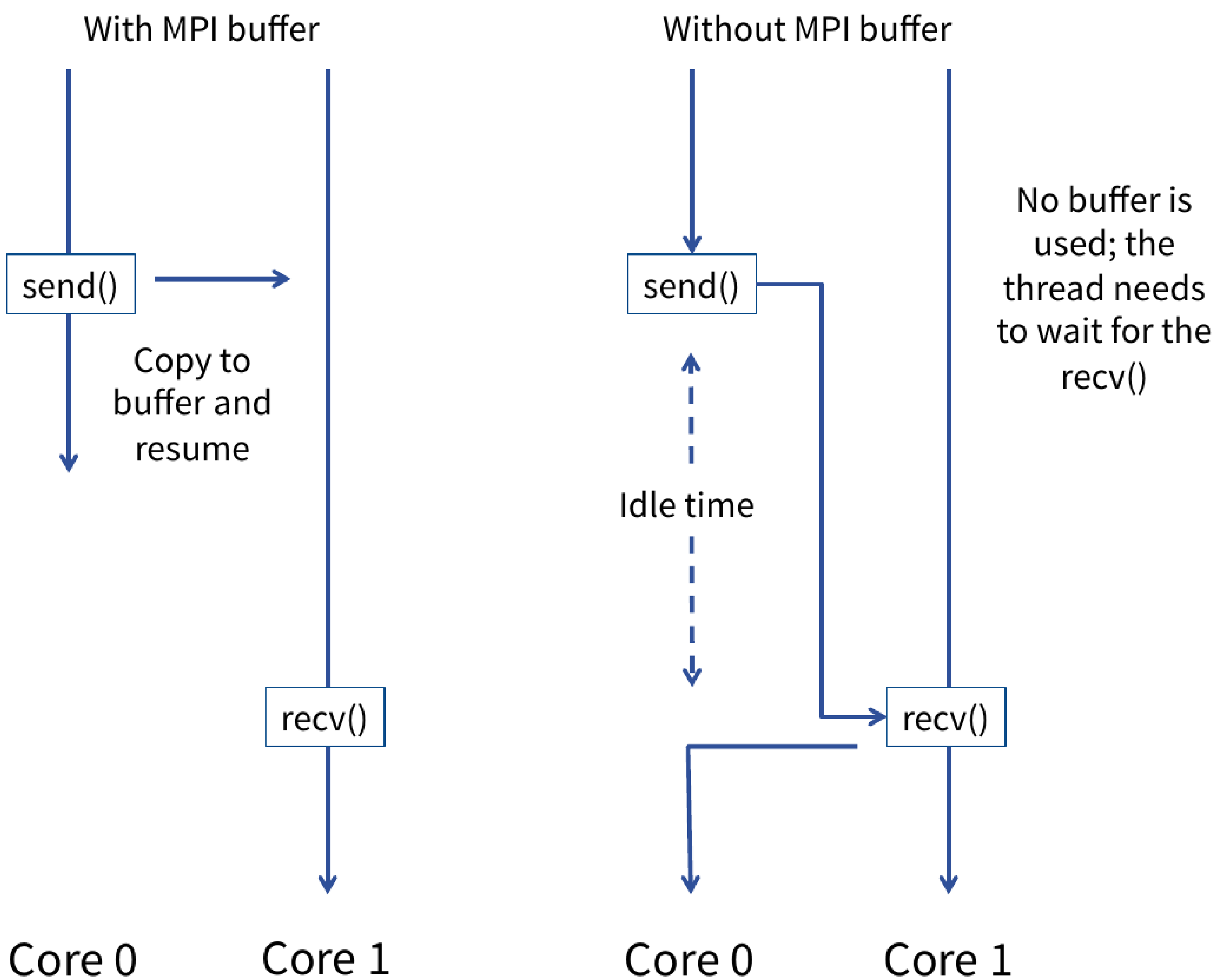
- Write a program which sends short message “IDDQD” from one process to another one which prints the result.

```
 ________
< IDDQD >
 --------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Blocking and Non-blocking

- Send and Recv are **blocking** operations:
  - The call does not return until the user buffer can be used again.
- **Send**
  - If MPI uses a separate system buffer, the data in ***buf*** (user buffer space) is copied to it; then the main thread resumes (fast).
  - If MPI does not use a separate system buffer, the main thread must wait until the communication over the network is complete.
- **Recv**
  - If communication happens before the call, the data is stored in an MPI system buffer and then simply copied into the user provided ***buf*** when ***MPI_Recv()*** is called.
- **Note**:
  - The user cannot enforce whether a buffer is used or not
  - The MPI library makes that decision based on the resources available and other factors.
  - However, calling different functions may alter the buffering behavior, see https://www.mcs.anl.gov/research/projects/mpi/sendmode.html

Blocking and Non-blocking



Non-blocking Send

- Replace: **`MPI_Send`** → **`MPI_Isend`**
-
```
int MPI_Isend(void* buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request)
```

- Parameters
  - ***request*** - use to get information later on about the status of that operation.
- I stand for Immediate, meaning that it does not wait on the matching receive. It may or may wait not for user buffer to be copied!
  - Call **`MPI_Wait`** to be able to use the user buffer again.

Non-blocking receive

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

- Test the status of the request using:
  - ```
    int MPI_Test(MPI_Request *request,
                 int *flag,
                 MPI_Status *status)
    ```
  - ***flag*** is 1 if request has been completed, 0 otherwise.

- Wait until request completes:
  - `int MPI_Wait(MPI_Request *request, MPI_Status *status)`

Example 2 – Send me a secret code

- Write a program which sends short message “IDKFA” in **non-blocking way** from one process to another one and prints the result.

```
 _______
< IDKFA >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Collective communication

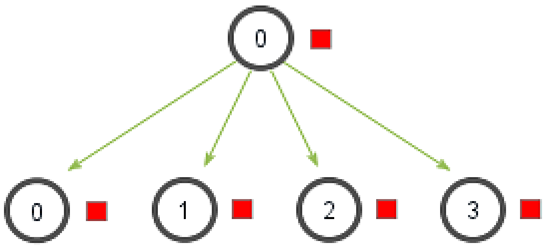
- Communication where **more than just two processes** are involved in.
- There are many instances where collective communications are required. For example:
  - Spread common data to all processes
  - Gather results from many processes
  - etc.
- Since these are typical operations, MPI provides several functions that implement these operations.
- All these operations have
  - blocking version
  - non-blocking version

Collective communication

- Always remember that every collective function call you make is **synchronized**.
  - If you try to call collective functions (e.g., **`MPI_Barrier`**, **`MPI_Bcast`**, etc.) without ensuring all processes in the communicator will also call it, your program will idle => **deadlock**.

Broadcast message

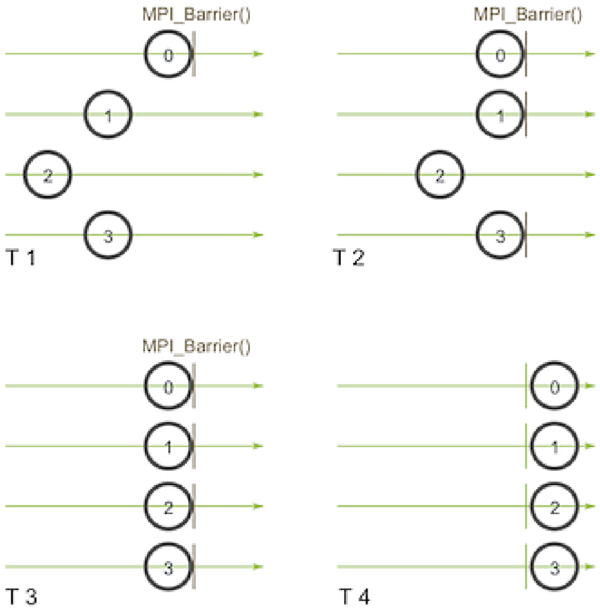
- ```
  int MPI_Bcast(void *buf,
                int count,
                MPI_Datatype datatype,
                int root,
                MPI_Comm comm)
  ```
- The simplest communication: one process sends a piece of data to all other processes.
- Parameters:
  - ***root*** – rank of the process that provides data (all other receive it)


  ```

Barrier

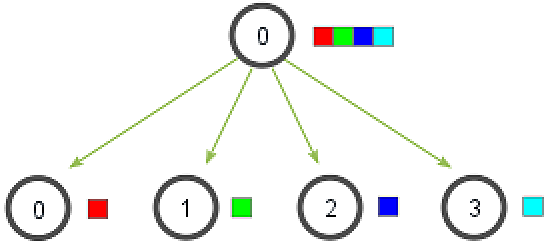
- **`int MPI_Barrier(MPI_Comm *comm*)`**

- Synchronization point among processes.
  - All **processes must reach a point** in their code before they can all begin executing again.



Scatter

- ```
  int MPI_Scatter(const void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  int root,
                  MPI_Comm comm)
  ```
- Sends personalized data from one root process to all other processes in a communicator group.
- The primary difference between **`MPI_Bcast`** and **`MPI_Scatter`** is that MPI_Bcast sends **the same piece** of data to all processes while **`MPI_Scatter`** sends **chunks of an array** to different processes.
- Parameters:
  - **`sendcount`** - dictate how many elements of a **`sendtype`** will be sent to **each** process.

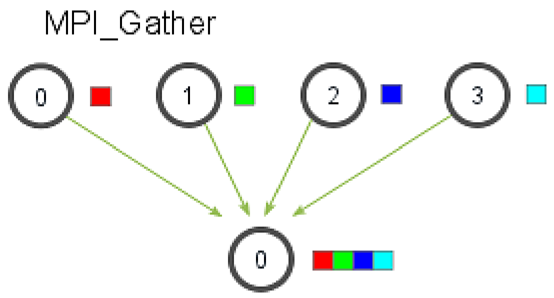

  ```

Scatterv

- ```
  int MPI_Scatterv(const void *sendbuf,
                   const int *sendcounts,
                    const int *displs,
                   MPI_Datatype sendtype,
                   void *recvbuf,
                   int recvcount,
                   MPI_Datatype recvtype,
                   int root,
                   MPI_Comm comm)
  ```
- Like scatter, but the programmer can say which parts of the buffer will be send to processes (similar function exists for other collective communications)
- Parameters:
  - **sendcounts** – array of integers representing the number of elements sent to each process
  - **displs** – array of integers, each specifying the displacement (relative to sendbuf) from which to take the outgoing data to process i
  ```

Gather

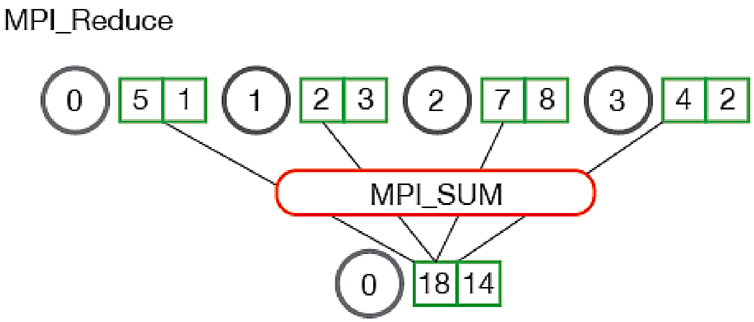
- ```
  int MPI_Gather(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm)
  ```
- **MPI_Gather** is the inverse of **MPI_Scatter**
- **MPI_Gather** takes elements from many processes and gathers them to one single root process (ordered by rank)




  ```

Reduce

- ```
  int MPI_Reduce(const void *sendbuf,
      void *recvbuf,
      int count,
      MPI_Datatype datatype,
      MPI_Op op,
      int root,
      MPI_Comm comm)
  ```
- Takes an array of input elements on each process and returns an array of output elements to the root process (similarly to Gather).
- The output elements contain the reduced result.



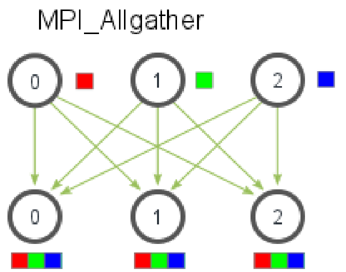

  ```

Operations for reduction

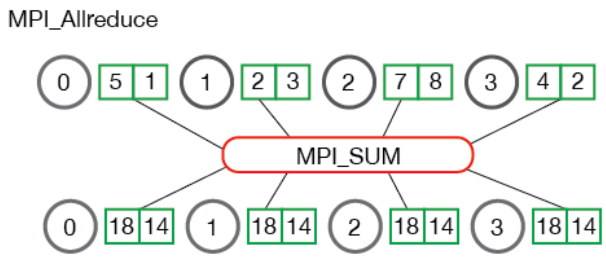
Representation	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bit-wise and
MPI_LOR	Logical or
MPI_BOR	Bit-wise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bit-wise exclusive or
MPI_MAXLOC	Maximum value and corresponding index
MPI_MINLOC	Minimum value and corresponding index

All-versions of operations

- Works exactly as the basic operation followed by broadcasting (everyone has the same results at the end)
- **Allgather**
  - `int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`

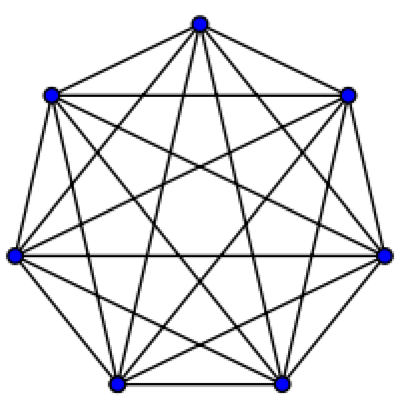


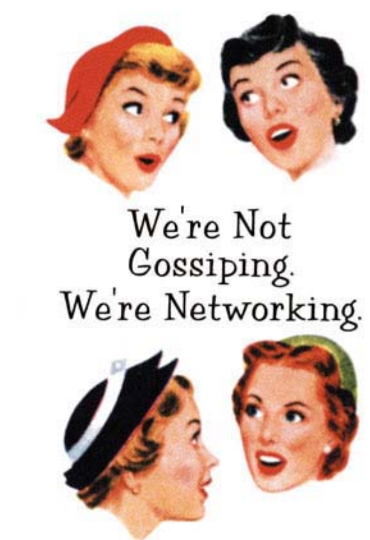
- **Allreduce**
  - `MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`



All to All communication - Gossiping

- ```
  int MPI_Alltoall(const void *sendbuf,
          int sendcount,
          MPI_Datatype sendtype,
          void *recvbuf,
          int recvcount,
          MPI_Datatype recvtype,
          MPI_Comm comm)
  ```
- All processes send data personalized data to all processes
- Total exchange of information




  ```

Example 2 – Vector normalization

- Write function for computing vector normalization using MPI.
  - Root process generates random vector, splits it into chunks and distribute the corresponding chunks to processes
  - Each process works with its chunk
  - In the end, the normalized vector is gathered in the root process