Parallel programming

OpenMPI

Libor Bukata a Jan Dvořák

FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

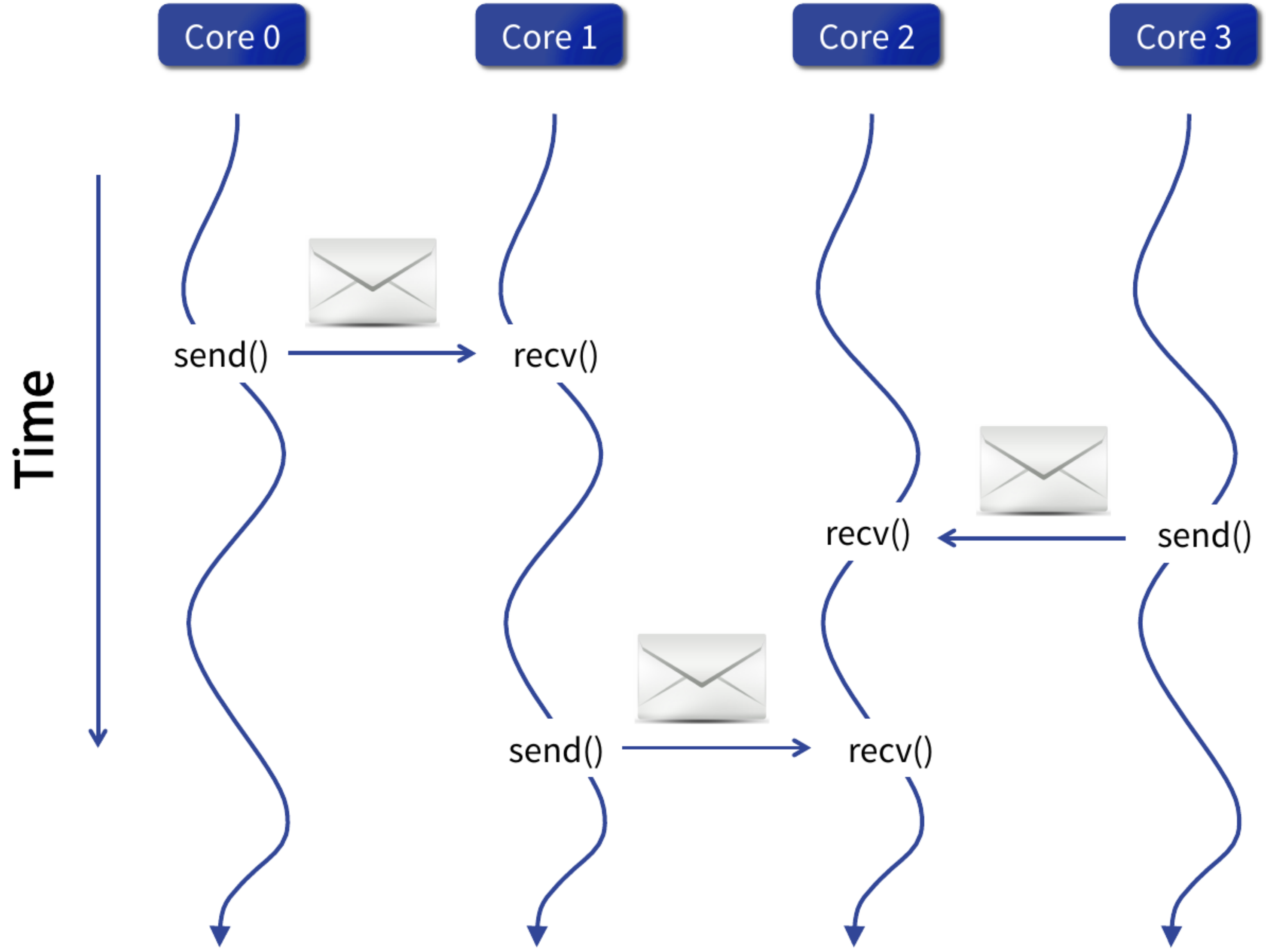
Distributed memory

- 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 communication model
- It is similar to the cooperation of people without a flip chart
  - However, electronic units have more reliable memory w.r.t. human
- Cluster computing

MPI

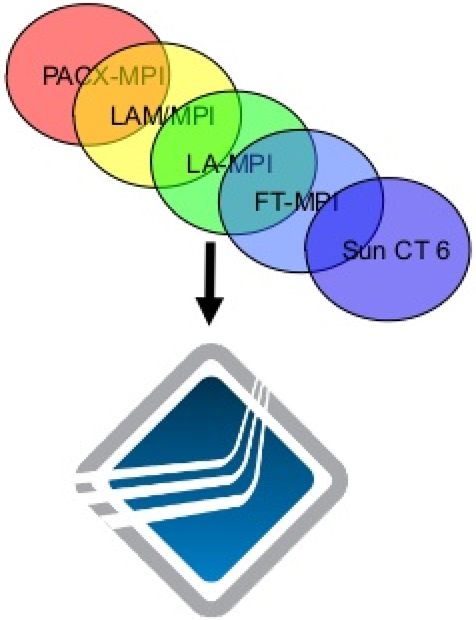
- MPI: **Message passing interface**
- All processes run the **same program**.
- Processes have assigned a **rank**.
- Based on the rank, processes can differ in an execution.
- Processes communicate by **sending and receiving** messages.
- Message passing:
  - Data transfer requires cooperative operations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

Communication example



OpenMPI

- Open source Message Passing Interface implementation
- Project founded in 2003 after intense discussion between multiple open source MPI implementations.
- Developed by a consortium of research, academic, and industry partners
- New BSD license



Basic openMPI operations

- `#include <mpi.h>`
  - Include header file with openMPI functions.
- `int MPI_Init(int *argc, char ***argv)`
  - Initializes openMPI runtime environment and process the arguments (trim the openMPI arguments/options from argument list)
- `int MPI_Finalize()`
  - Terminates MPI execution environment.
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Returns the ***rank*** (identifier) of the process in communicator ***comm***.
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Returns the ***size*** of the group associated with communicator ***comm***.

Send a message

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

- ***buff*** - buffer which contains the data elements to be sent
- ***count*** - number of elements to be sent
- ***datatype*** - data type of entries
- ***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 (more on this later)


  ```

Datatypes in openMPI

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

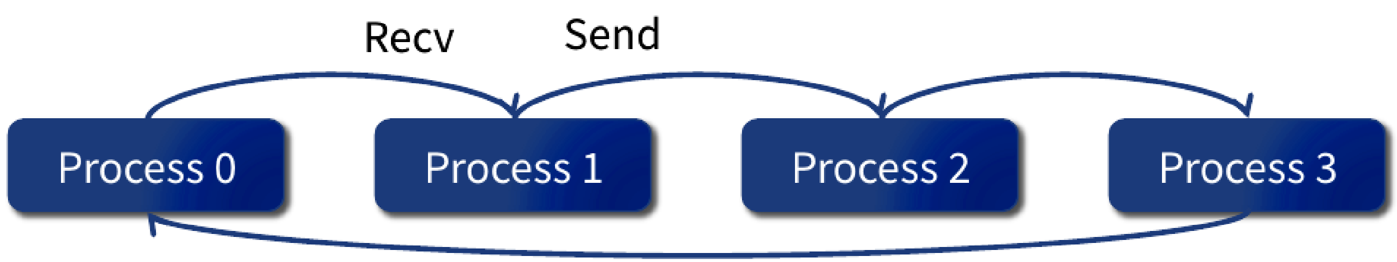
Receive a message

- ```
  int MPI_Recv(void *buff,
               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, length of the message) 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.


  ```

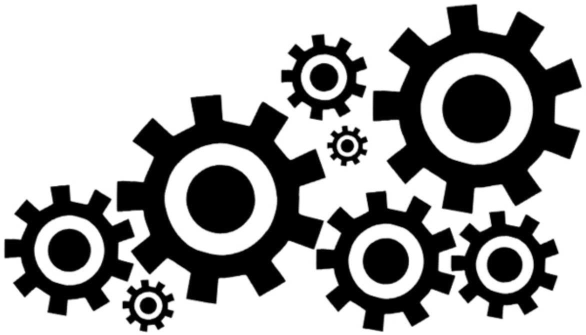
Simultaneous Send and receive

- ```
  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:


  ```

Preparation of an openMPI program

- Compilation:
  - C – use mpicc compilator
  - C++ – use mpiCC/mpic++/mpicxx compilator
  - mpicc and mpiCC are just wrappers that call compiler with necessary options for openMPI
- Run:
  - You will need one of openMPI implementations:
    - MVAPICH, MPICH, LAM/MPI or OpenMPI
  - **`mpirun -np 4 program arguments`**
    - **`np`** – number of used processes
    - **`hostfile`** – file with a list of hosts on which to launch MPI processes
    - This will run the code 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.



Structure of basic openMPI program

```
#include <mpi.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    int world_rank;
    // What is my rank (process ID).
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
                  int world_size;
    // How many processes do we have in total?
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_rank%2 == 0) {
        // If we have even rank, we do something
    } else {
        // If we have odd rank, we do something else
    }

    // Always call at the end
    MPI_Finalize();
}
```

- `MPI_COMM_WORLD` – Common communicator representing the whole world

Example 1 – Send me a secret code

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

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

OpenMPI - API

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

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

- ```
  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)
  ```

```
#include <mpi.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    int world_rank;
    // What is my rank (process ID).
    MPI_Comm_rank(MPI_COMM_WORLD,
                  &world_rank);
    int world_size;
    // How many processes do we have in total?
    MPI_Comm_size(MPI_COMM_WORLD,
                  &world_size);

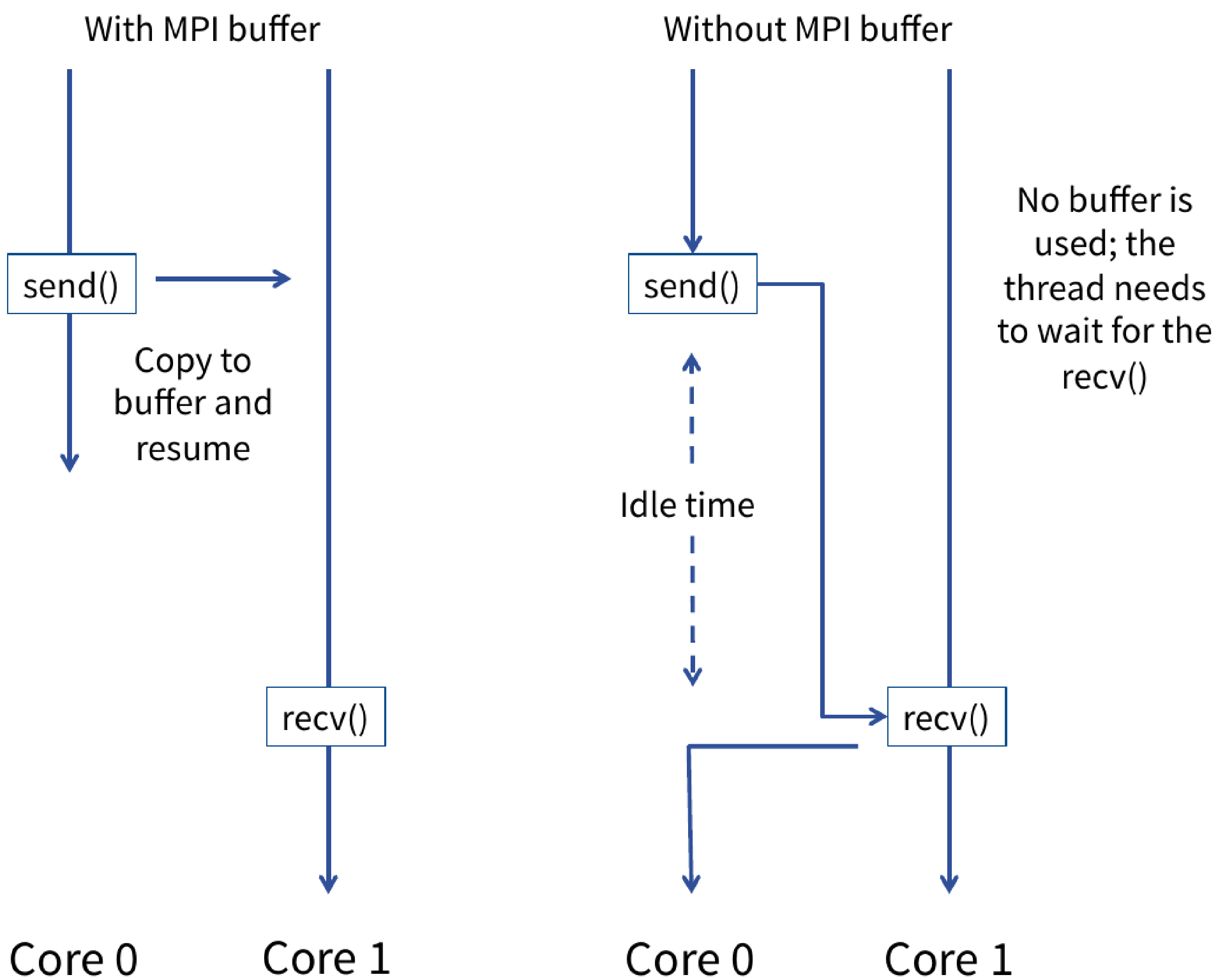
    if (world_rank%2 == 0) {
        // If we have even rank, we do something
    } else {
        // If we have odd rank, we do something else
    }

    // Always call at the end
    MPI_Finalize();
}
```
```

Blocking and Non-blocking

- Send and Recv are **blocking** operations:
  - The call does not return until the resources become available again
- **Send**
  - If MPI uses a separate system buffer, the data in ***`buff`*** (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 ***`buff`*** when ***`MPI_Recv()`*** is called.
- **Note**:
  - The user cannot decide whether a buffer is used or not
  - The MPI library makes that decision based on the resources available and other factors.

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)`

```
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

Non-blocking receive

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

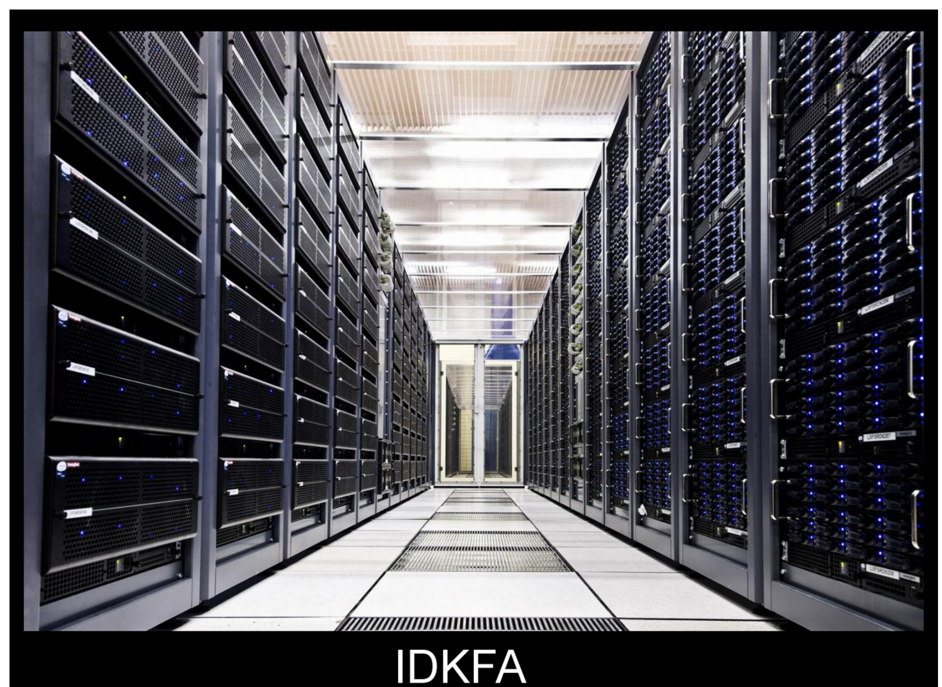
- Test the status of the request using:
  - ```
    int MPI_Test(MPI_Request *request,
                 int *flag,
                 MPI_Status *status)
    ```
  - ***flag*** is 1 if request has been completed, 0 otherwise.

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

Example 1.5 – Send me a secret code

- Task:
  - 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 |
                ||     ||
```



OpenMPI - API

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

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

• MPI_Test(MPI_Request *request, int *flag,
           MPI_Status *status)

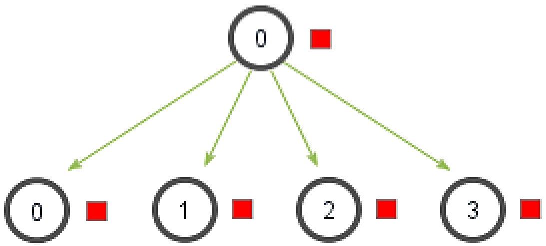
• int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

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 functionalities that implement these operations.
- All these operations have
  - blocking version
  - non-blocking version

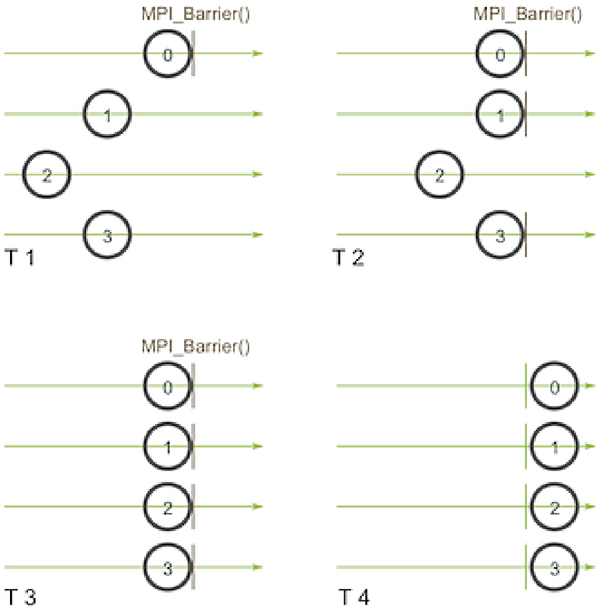
Broadcast message

- **`int MPI_Bcast(void *buffer,`**
```
int MPI_Bcast(void *buffer,
              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)



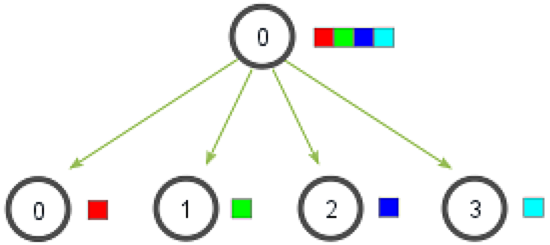
Barrier

- **`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.
- Always remember that every collective call you make is **synchronized**.
  - If you try to call MPI_Barrier or other collective routines without ensuring all processes in the communicator will also call it, your program will idle => **deadlock**.



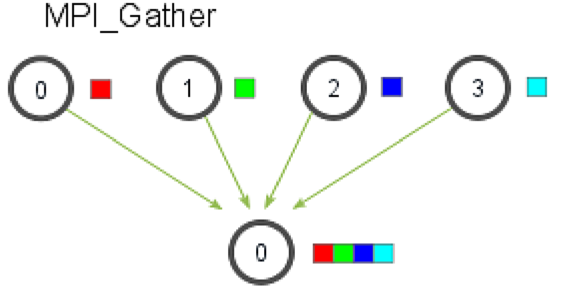
Scatter

-
```
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.



Gather

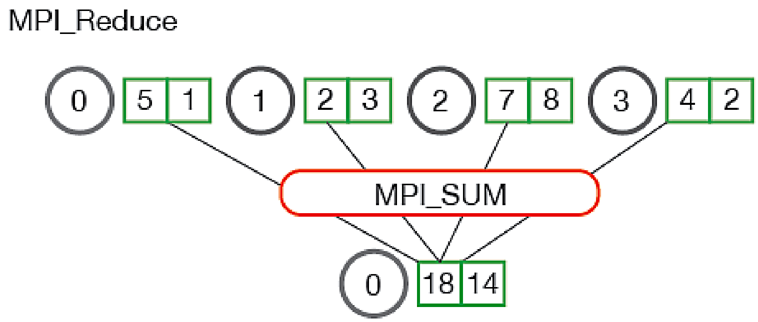
- ```
  int MPI_Gather(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm)
  ```
- **`MPI_Gather`**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




  ```

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.



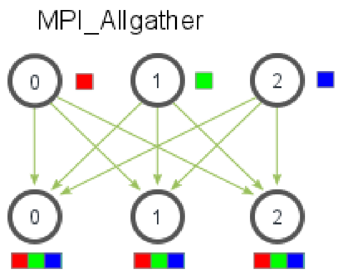

  ```

Operations for reduction

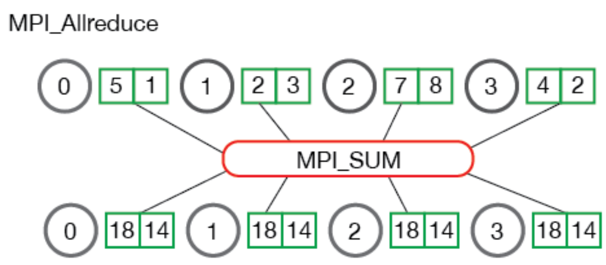
Representation	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bit-wise and
MPI_LOR	Logical or
MPI_BOR	Bit-wise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bit-wise exclusive or
MPI_MAXLOC	Maximum value and corresponding index
MPI_MINLOC	Minimum value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

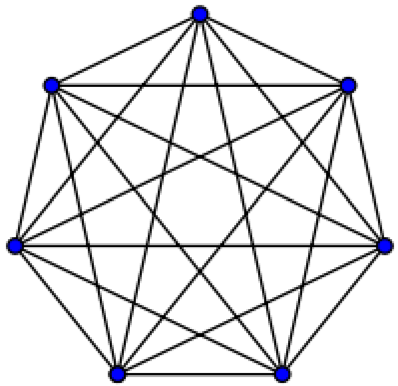


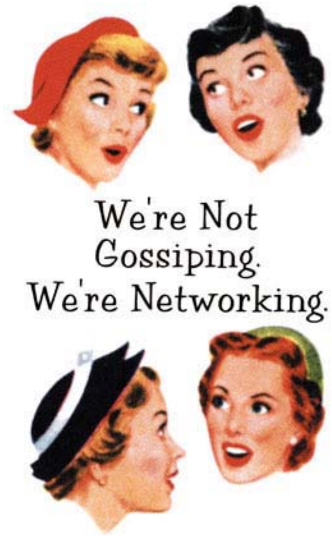
- **Allreduce**
  - `MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`



All to All communication - Gossiping

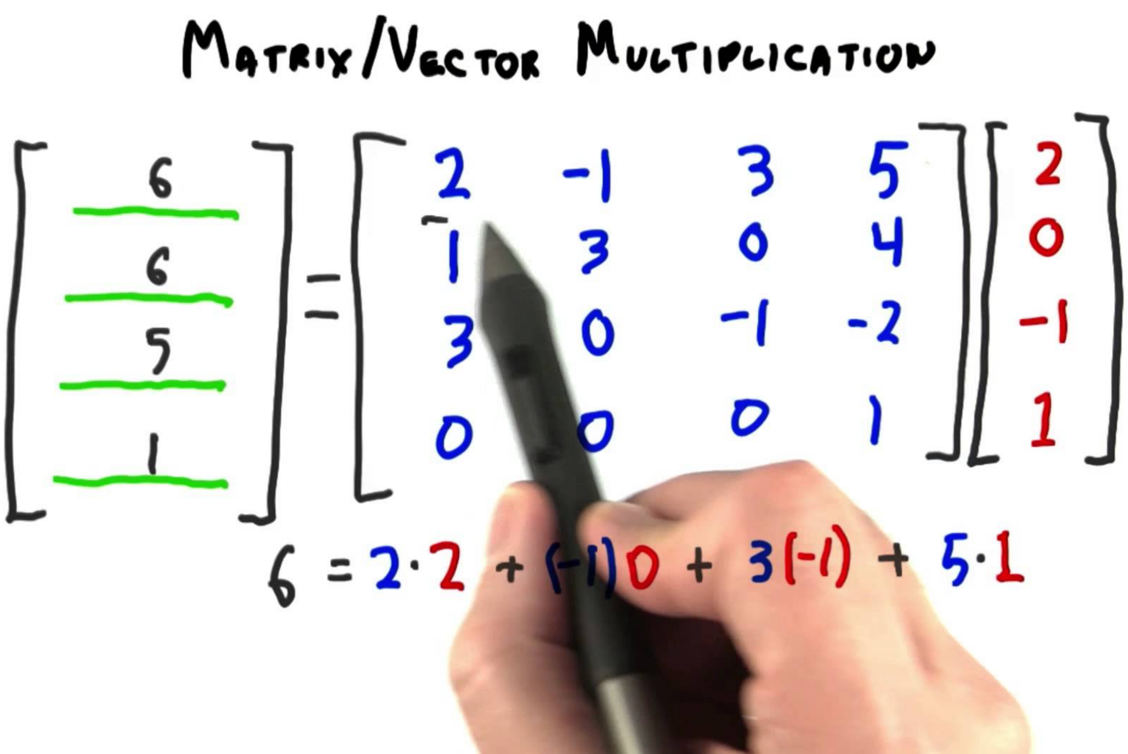
- ```
  int MPI_Alltoall(const void *sendbuf,
          int sendcount,
          MPI_Datatype sendtype,
          void *recvbuf,
          int recvcount,
          MPI_Datatype recvtype,
          MPI_Comm comm)
  ```
- All processes send data personalized data to all processes
- Total exchange of information




  ```

Example 2 – Matrix-vector multiplication

- Task
  - Implement an parallel algorithm for matrix-vector multiplication in openMPI.
  - Small hint:



OpenMPI - API

- `int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
- `int MPI_Barrier(MPI_Comm comm)`
- ```
  int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                  int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
  ```
- ```
  int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
  ```
- ```
  int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
                 MPI_Op op, int root, MPI_Comm comm)
  ```
- ```
  int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                    void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
  ```
- ```
  MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  ```
- ```
  int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                   void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
  ```