Parallel programming

MPI 1

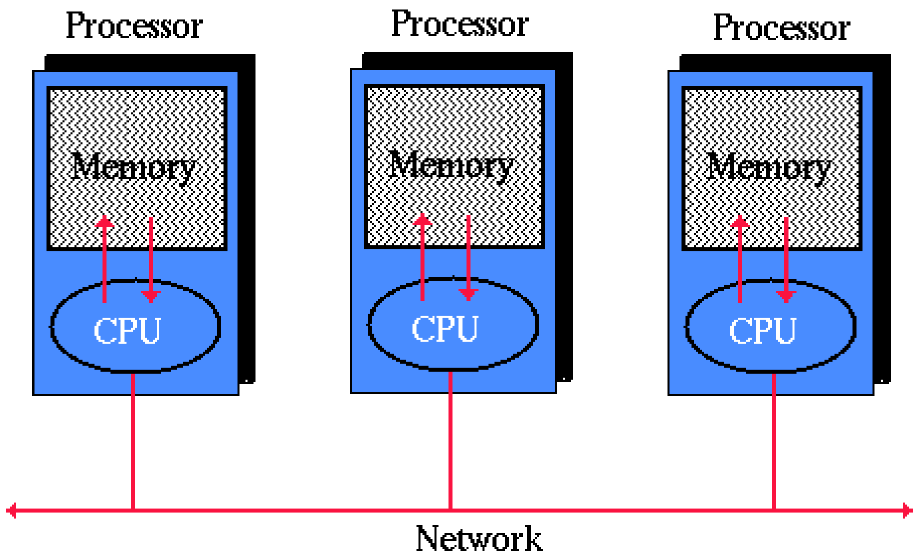
FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

Distributed memory

- Each unit has its **own memory space**
- **Explicit communication** between units (often through a network!) is required
  - point-to-point communication
  - collective communication
- Frequent application: *cluster computing*

Distributed memory

- **Node independence**: each computing node operates independently, with its own local memory
- **Communication challenges**: explicit communication mechanisms to exchange data
- **Scalability**: distribute workloads and manage resources



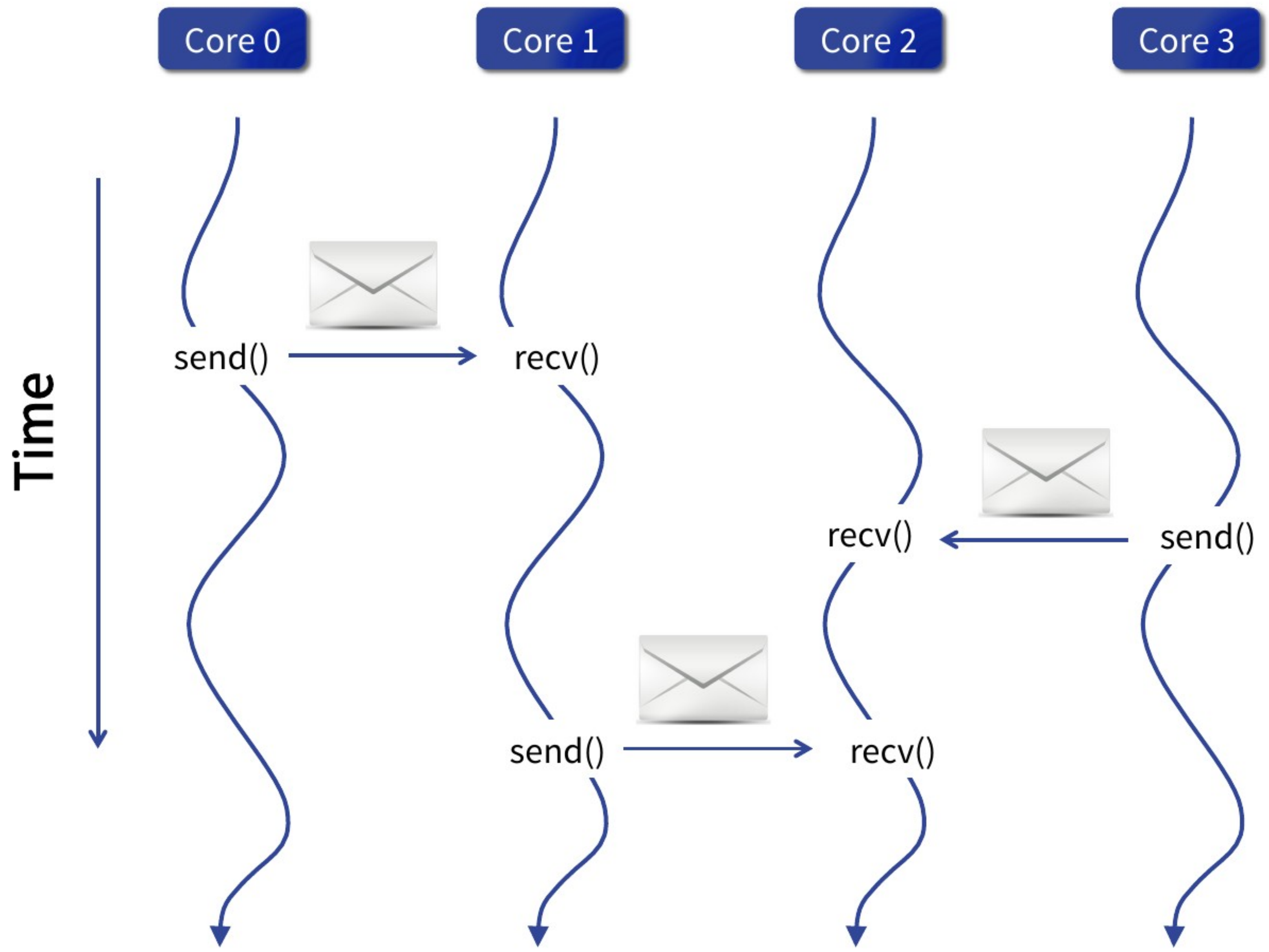
MPI: Message Passing Interface

- A **standard** for developing parallel distributed applications
- MPI is supported by many programming languages and platforms:
  - *C, C++, and Fortran*
  - For JAVA see : *Message Passing for Java Express (MPJ Express)*
  - For .NET see : *https://github.com/mpidotnet/MPI.NET*

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**

Communication example



Example: “Hello, world!”

- Use *HelloWorld.cpp* skeleton
- Write a program that
  - initializes MPI
  - each process print its rank
  - process with rank 0 prints the total number of processes (communicator size)

Basic MPI operations

- ***#include <mpi.h>***
  - Include the header file with MPI functions
- ***int MPI_Init(int *argc, char ***argv)***
  - Initializes MPI runtime environment
- ***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
  - *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

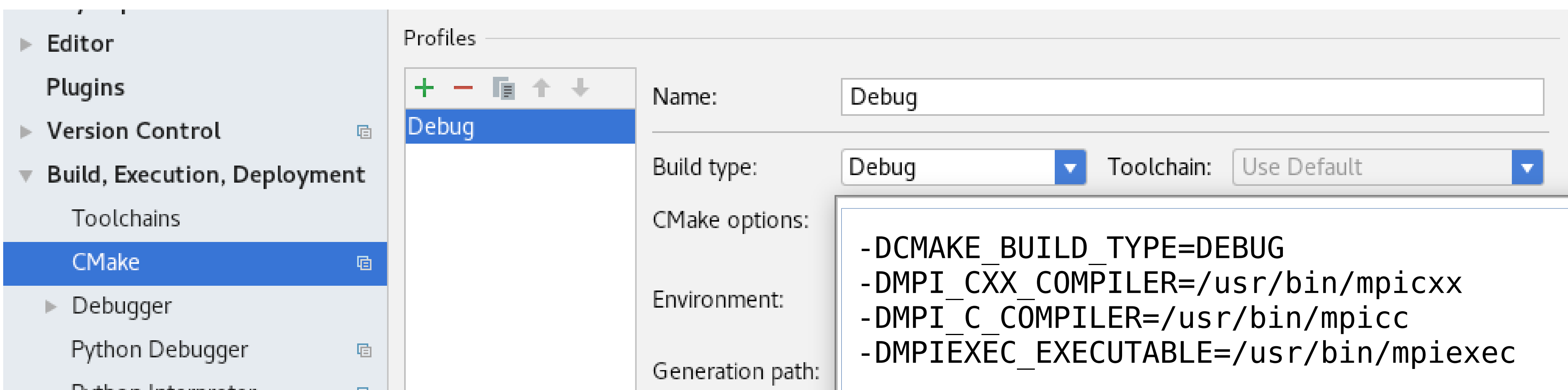
Compilation - CMake

```
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)



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
- Microsoft 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
- MinGW toolchain:
  - to be able reach and run **mpiexec** program from windows command prompt (powershell), it is necessary install library at this link

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


  ```

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.

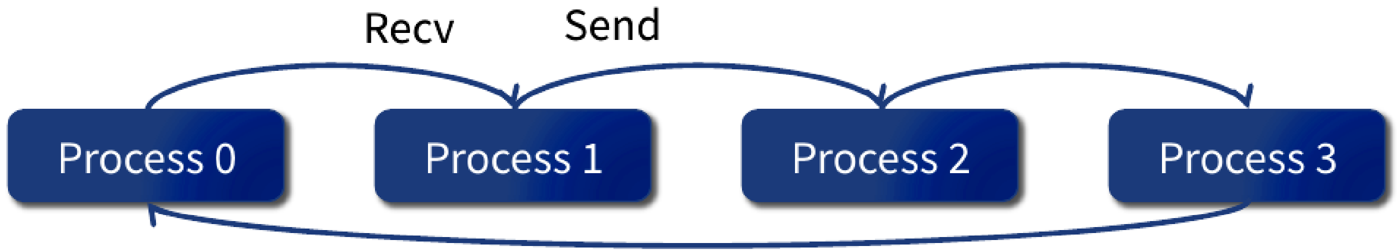

  ```

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

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: “Send me a secret code!”

- Use *SendAndReceive.cpp* skeleton
- Write a program that
  - sends short message “IDDQD” from one process to another one
  - receiving process prints the result

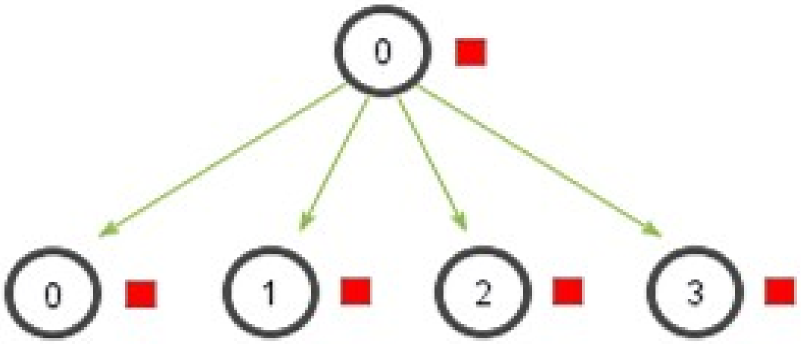
```
 _______
< IDDQD >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Collective communication

- Communication between **all** the processes **inside a communicator group**
- Examples of collective communication:
  - spread common data to all processes
  - gather results from many processes
  - etc.
- MPI provides several functions implementing collective communication patterns
- All these operations have
  - blocking version
  - non-blocking version

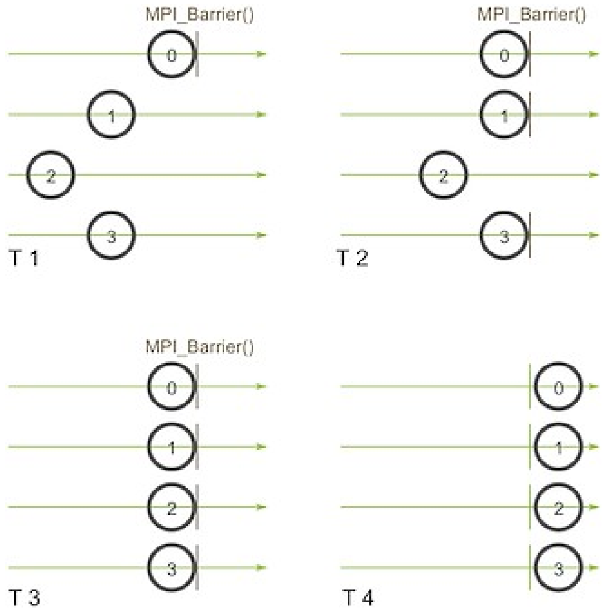
Broadcast message

- ```
  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)


  ```

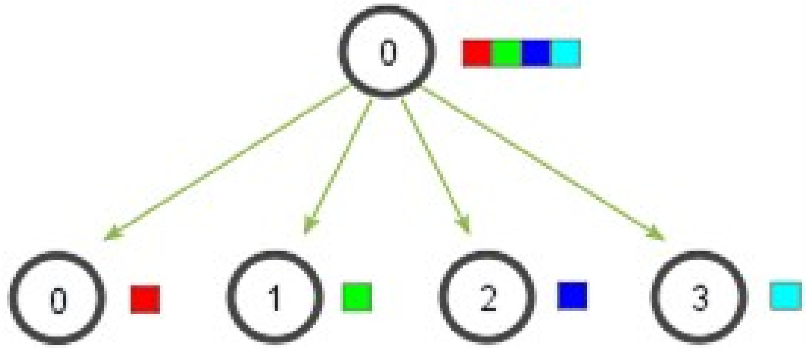
Barrier synchronization

- **`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.



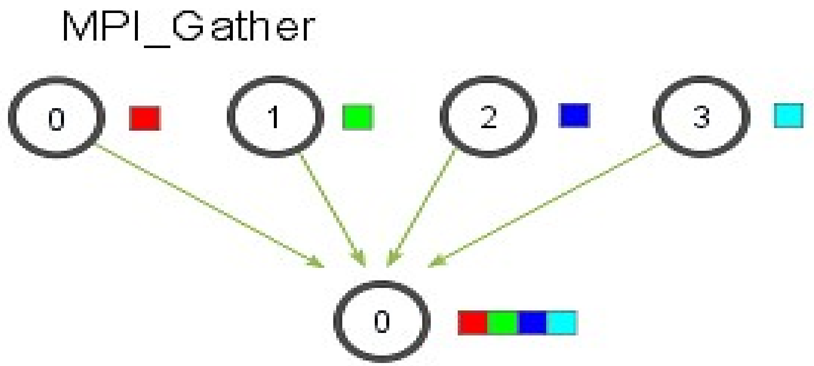
Scatter operation

-
```
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.



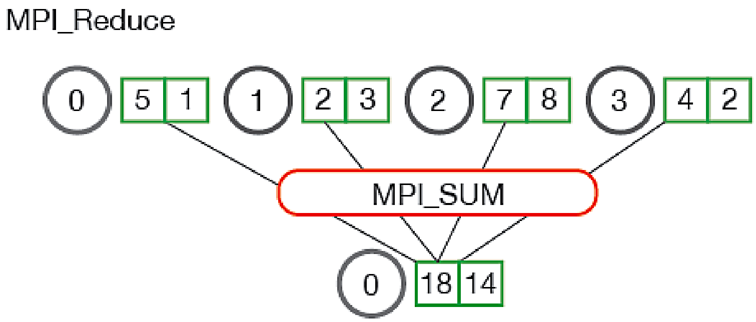
Gather operation

- ```
  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 operation

- ```
  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.



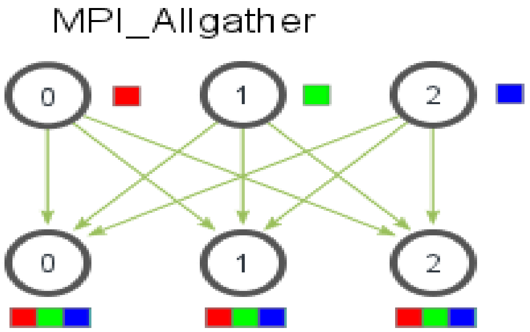

  ```

Operations for reduction

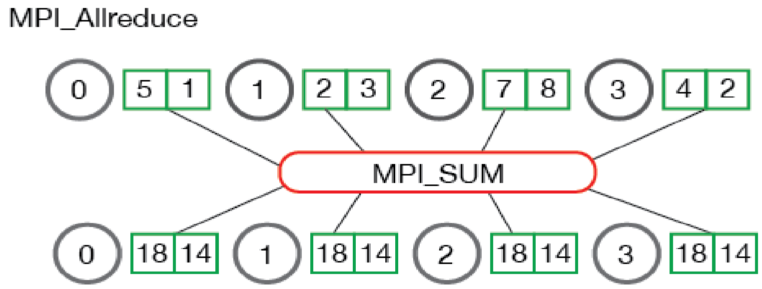
Representation	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bit-wise and
MPI_LOR	Logical or
MPI_BOR	Bit-wise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bit-wise exclusive or
MPI_MAXLOC	Maximum value and corresponding index
MPI_MINLOC	Minimum value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

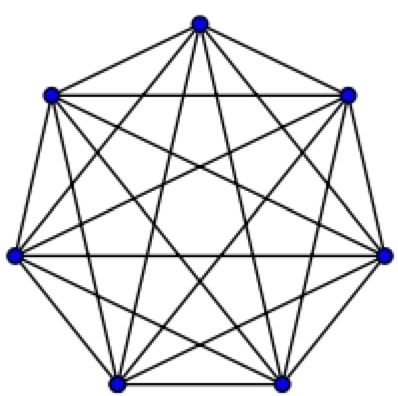


- **Allreduce**
  - `MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`



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: Vector normalization

- Use *VectorNormalization.cpp* skeleton
- Compute 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
  - the normalized vector is gathered in the root process

Visualization of Example 2