Barcelona Supercomputing Center
Centro Nacional de Supercomputación

# Introduction to the MPI programming model

Janko Strassburg

PATC Parallel Programming Workshop October 2013

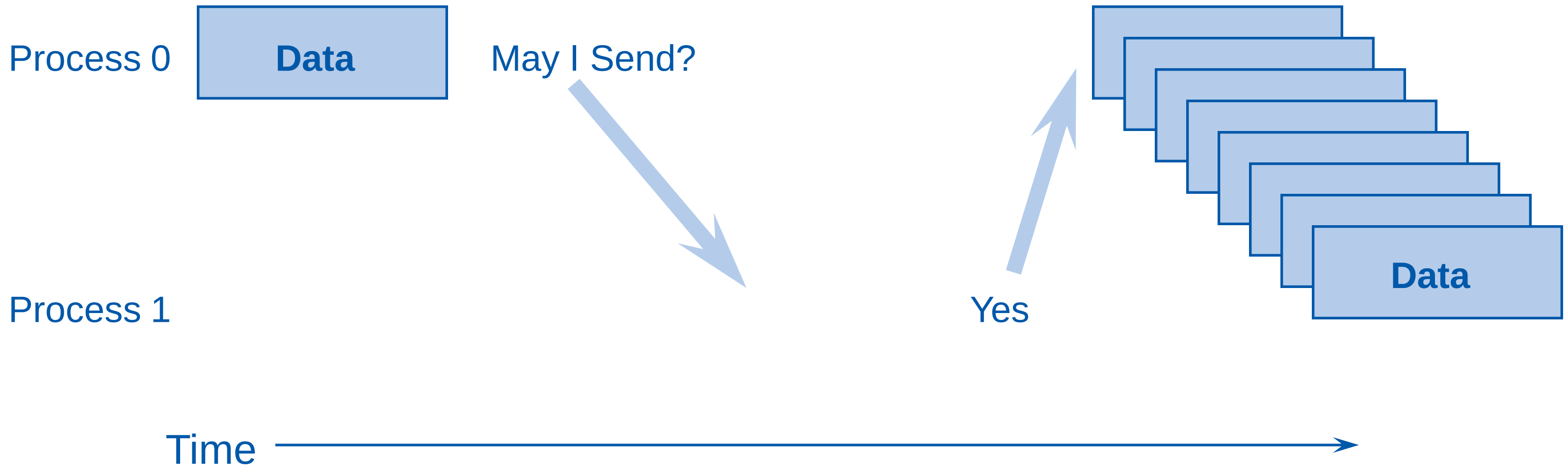
# MPI

- Message Passing Interface – MPI
  - De facto standard
  - Although not an official standard (IEEE, …)

- Maintained by the MPI forum
  - Writes and ratifies the standard
  - Consisting of academic, research and industry representatives

# What is message passing?

- Data transfer plus synchronization




- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

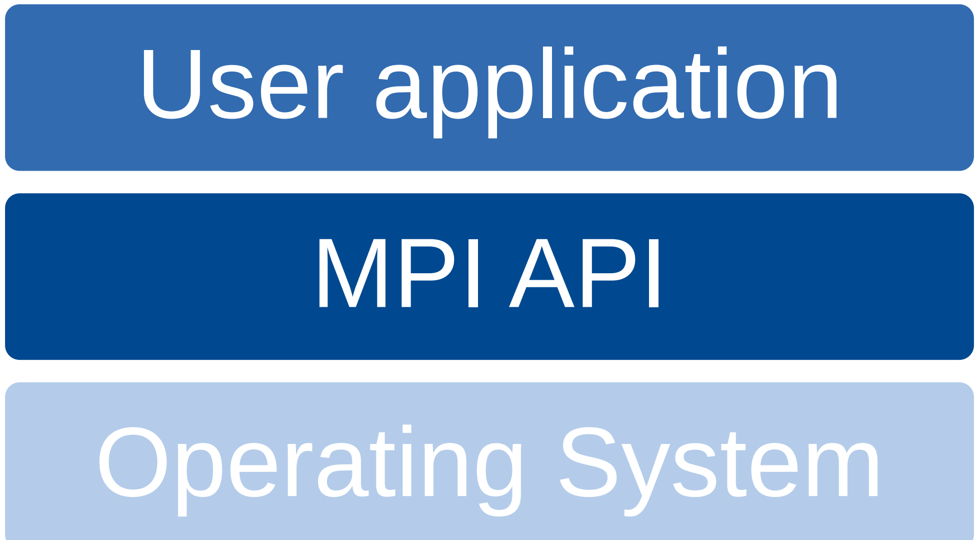
# Purpose

- Divide problems in parallel computing
- Use multiple processors, hundreds or thousands at a time
- Spread job across multiple nodes
  - Computations too big for one server
  - Memory requirements
  - Splitting the problem, divide and conquer approach

# MPI Use

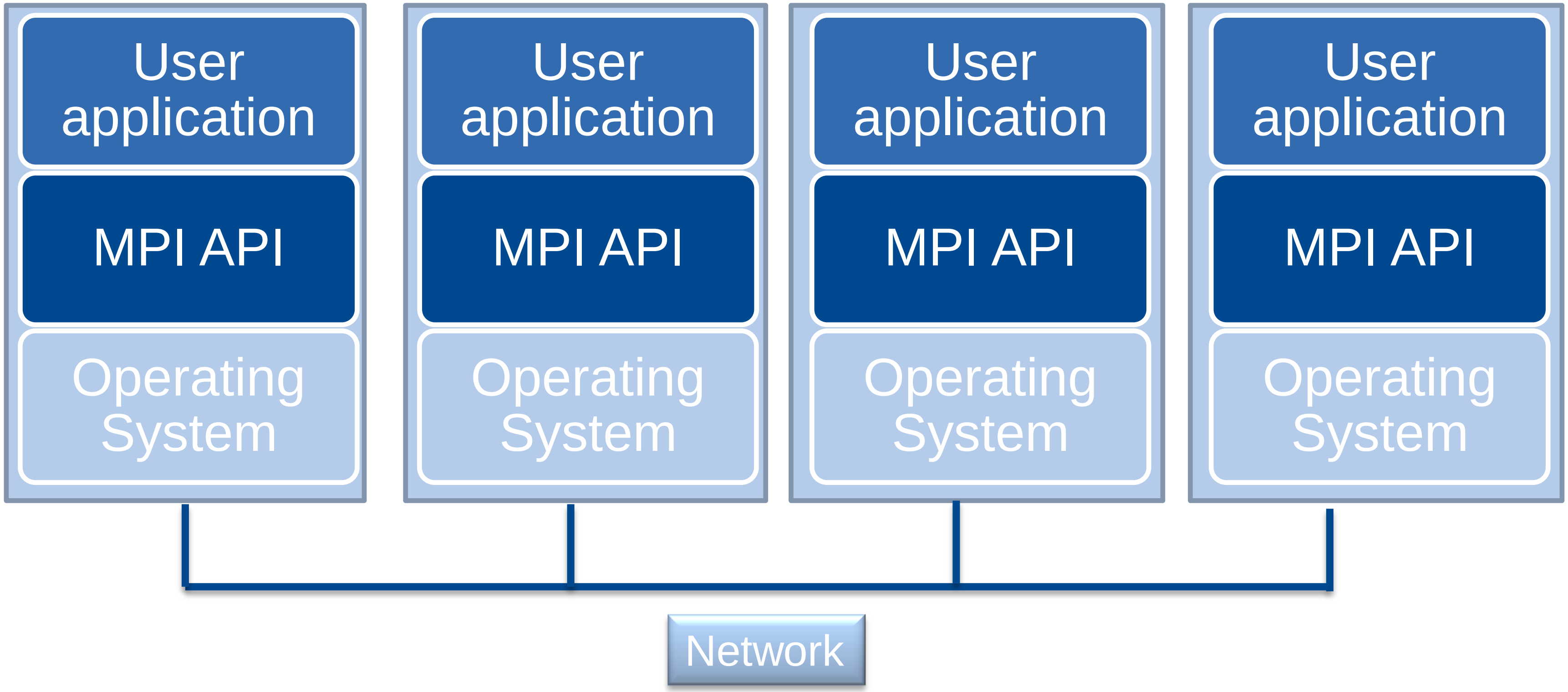
- Abstracts view of the network
  - Shared memory / Sockets
  - Ethernet / Infiniband
  - High speed communication network / High throughput data network
  - …
- Application communicates using simple commands
  - MPI_SEND / MPI_RECV
  - Implementation handles connections automatically

# MPI High-Level View

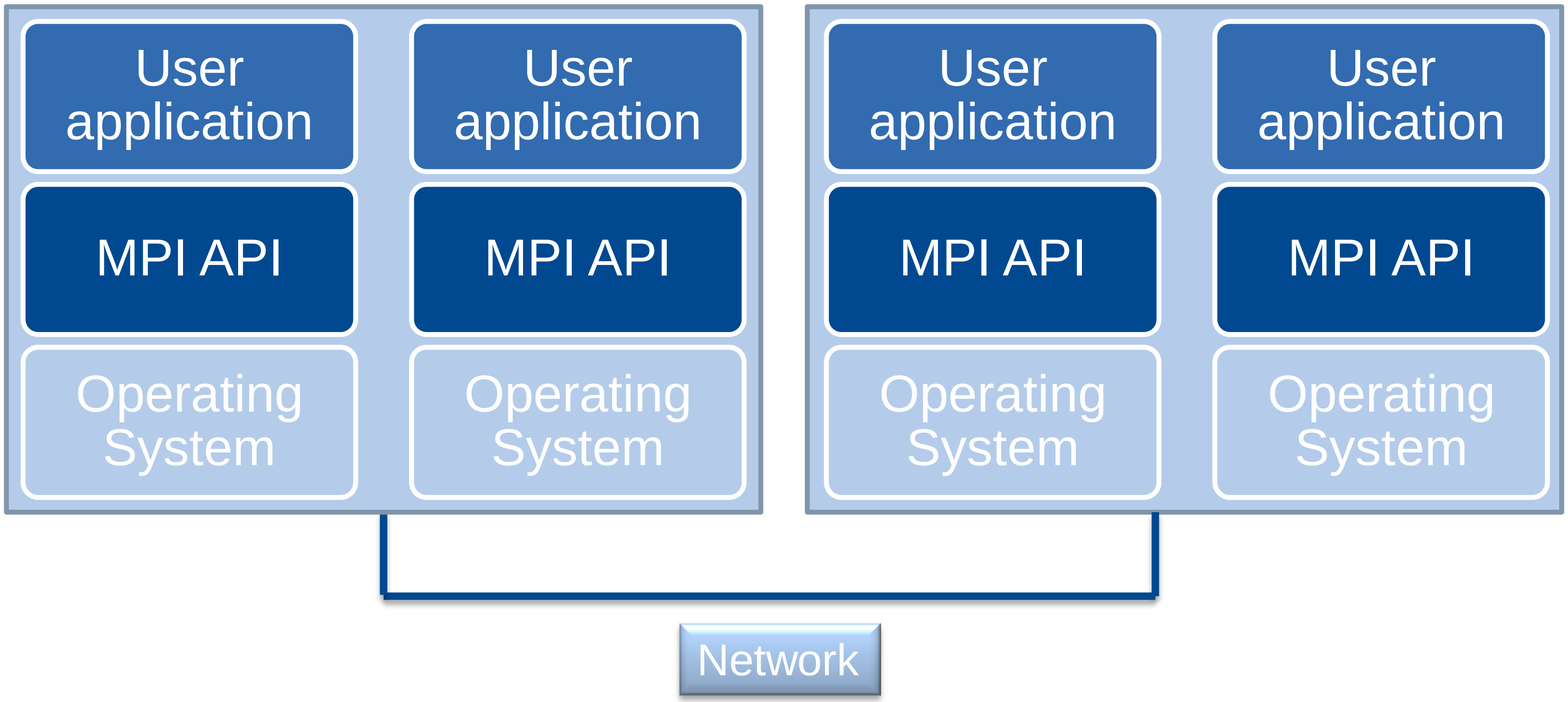



- Layered system
- Abstracting hardware from the programmer

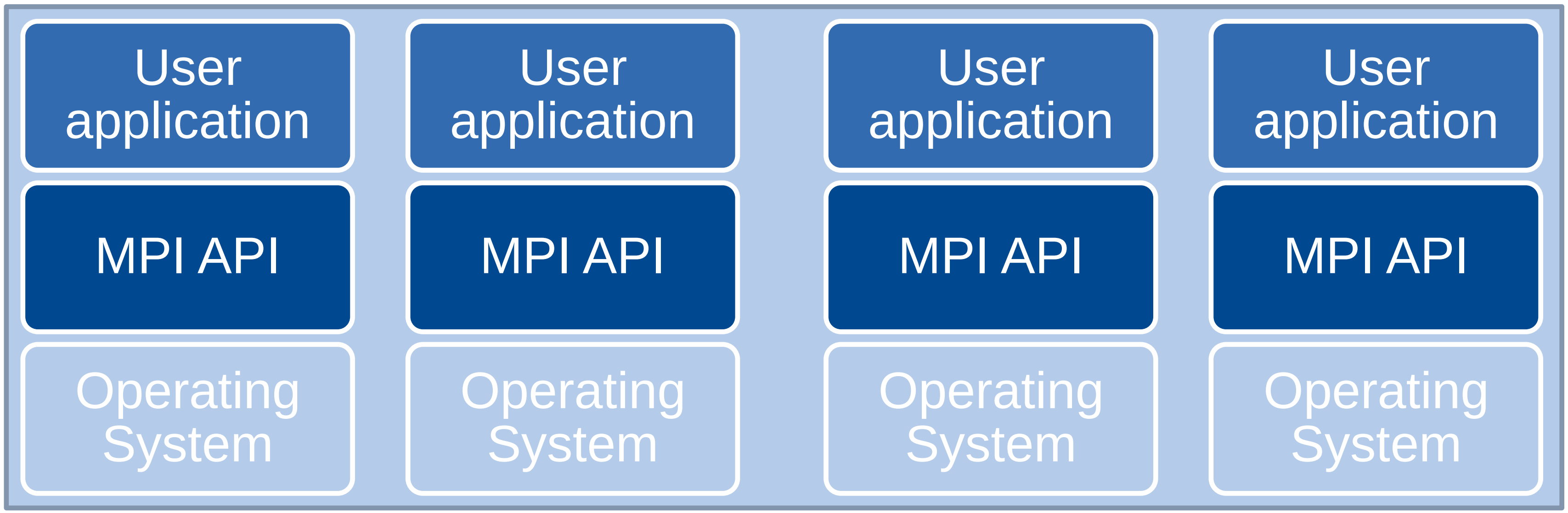
# Example: 4 Nodes



# Example: 2 Nodes



# Example: 1 Node




All processes running on the same machine

**Possibilities**:

- Dedicated parallel programming languages
- Extensions of existing sequential programming languages
- Usage of existing sequential programming languages + libraries with external functions for message passing

# Approach:

– Use existing FORTRAN/C code

– Function calls to message passing library

# **Explicit parallelism**:

User defines

– which processes to execute,

– when and how to exchange messages, und

– which data to exchange within messages.

# MPI Intention

- Specification of a standard library for programming message passing systems
- Interface: practical, portable, efficient, and flexible
- ⇨ *Easy to use*
- For vendors, programmers, and users

# MPI Goals

- Design of a standardized API
- Possibilities for efficient communication (Hardware-Specialities, ...)
- Implementations for heterogeneous environments
- Definition of an interface in a traditional way (comparable to other systems)
- Availability of extensions for increased flexibility

# MPI Forum

## Collaboration of 40 Organisations world-wide:

- IBM T.J. Watson Research Center
- Intels NX/2
- Express
- nCUBE’s VERTEX
- p4 - Portable Programs for Parallel Processors
- PARMACS
- Zipcode
- PVM
- Chameleon
- PICL
- ...

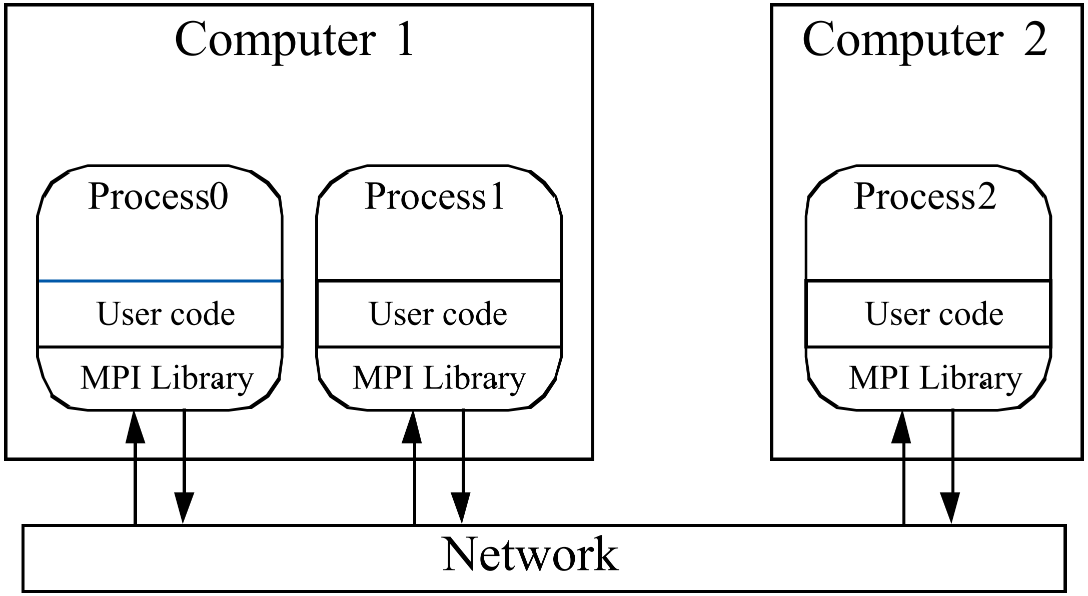
# Available Implementations

- Open MPI:
  - Combined effort from FT-MPI, LA-MPI, LAM/MPI, PACX-MPI
  - De facto standard; used on many TOP500 systems
- MPICH
- CHIMP
- LAM
- FT-MPI
- Vendor specific implementations:
  - Bull, Fujitsu, Cray, IBM, SGI, DEC, Parsytec, HP, ...

# MPI Programming Model

- **Parallelization**:
  - Explicit parallel language constructs (for communication)
  - Library with communication functions
- **Programming Model**:
  - SPMD (single program multiple data)
  - All processes load the same source code
  - Distinction through process number

# MPI Program




**2 Parts**:

- User code
- MPI Functionality (from MPI Library)

# MPI Functionality

- Process Creation and Execution
- Queries for system environment
- Point-to-point communication (Send/Receive)
- Collective Operations (Broadcast, ...)
- Process groups
- Communication context
- Process topologies
- Profiling Interface

# Processes

**Characteristics**:

- For *Parallelism*, computation must be partitioned into multiple processes (or tasks)

- Processes are assigned to processors ⇨ **mapping**
  - 1 Process = 1 Processor
  - n Processes = 1 Processor

- *Multitasking* on one processor:
  - Disadvantage: Longer execution time due to time-sharing
  - Advantage: Overlapping of communication latency

# Granularity

- The size of a process defines its granularity
- Coarse Granularity
  - each process contains many sequential execution blocks
- Fine Granularity
  - each process contains only few (sometimes one) instructions

# Granularity

- Granularity =
  Size of computational blocks between communication and synchronization operations

- The higher the granularity, the
  - smaller the costs for process creation
  - smaller the number of possible processes and the achievable parallelism

# Parallelization

- Data Partitioning:
  **SPMD** = Single Program Multiple Data

- Task Partitioning:
  **MPMD** = Multiple Program Multiple Data

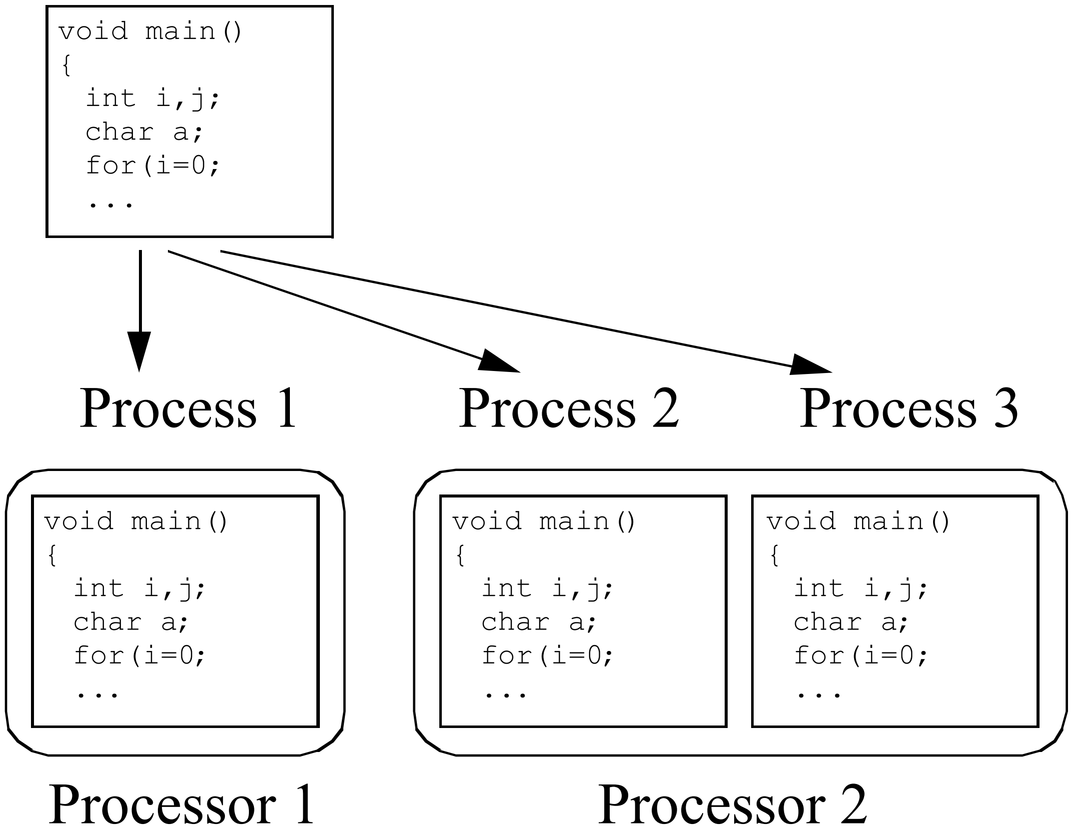
**Types of Process-Creation**:

- Static
- Dynamic

# Data Partitioning (SPMD)

**Implementation:**
1 Source code

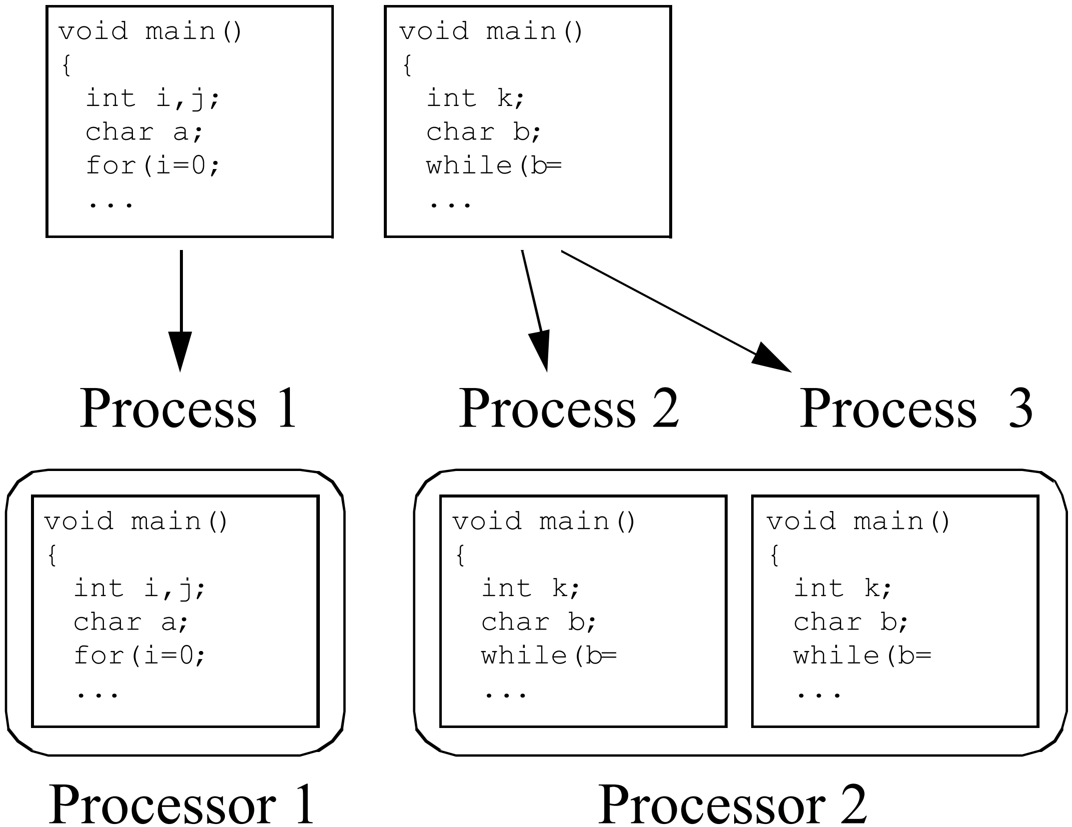
**Execution:**
n Executables



# Task-Partitioning (MPMD)

**Implementation:**
m Source codes

**Execution:**
n Executables



# Comparison: SPMD/MPMD

**SPMD**:

- One source code for all processes

- Distinction in the source code through control statements

```
if (pid() == MASTER) { ... }
else { ... }
```

- Widely used

# Comparison: SPMD/MPMD

**MPMD**:

- One source for each process
- Higher flexibility and modularity
- Administration of source codes difficult
- Additional effort during process creation
- Dynamic process creation possible

# Process Creation

## **Static**:

- All processes are defined before execution
- System starts a fixed number of processes
- Each process receives same copy of the code

# Process Creation

## **Dynamic**:

- Processes can creation/execute/terminate other processes during execution
- Number of processes changes during execution
- Special constructs or functions are needed

- Advantage
  higher flexibility than SPMD
- Disadvantage
  process creation expensive ⇨ overhead

# Process Creation/Execution

**Commands**:

- Creation and execution of processes is not part of the standard, but instead depends on the chosen implementation:

  Compile: `mpicc -o <exec> <file>.c`

  Execute: `mpirun -np <proc> <exec>`

- Process Creation: only static (before MPI-2)
- SPMD programming model

# Basic-Code-Fragment

## Initialization and Exit:

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

Interface defintion

Provide Command Line Parameters

Initialize MPI

Terminate and Clean up MPI

# Structure of MPI Functions

## **General**:

```
result = MPI_Xxx(…);
```

## **Example**:

```
result = MPI_Init(&argc, &argv);
if(result!=MPI_SUCCESS) {
   fprintf(stderr,”Problem”);
   fflush(stderr);
   MPI_Abort(MPI_COMM_WORLD,result);
}
```

# Query Functions

**Identification**:

- *Who am I*?
- Which process number has the current process?

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank)
```

- *Who else is there*?
- How many processes have been started?

```
MPI_Comm_size(MPI_COMM_WORLD, &mysize)
```

- Characteristics: 0 <= myrank < mysize

# MPI & SPMD Restrictions

- *Ideally*: Each process executes the same code.
- *Usually*: One (or a few) processes execute slightly different codes.

- Preliminaries:
  Statements to distinguish processes and the subsequent code execution

- Example: Master-slave program
  ⇨ complete code within one program/executable

# Master-Slave Program

```
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    ...
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if(myrank == 0)
          master();
    else
          slave();
    ...
    MPI_Finalize();
}
```

# Global Variables

## **Problem**:

```
int main(int argc, char *argv[])
{  float big_array[10000];
```

## **Solution**:

```
int main(int argc, char *argv[])
{
    if(myrank == 0) {
            float big_array[10000];
```

Allocate large arrays only on the ranks needed to save memory

# Global Variables

## **Problem**:

```
int main(int argc, char *argv[])
{ float big_array[10000];
```

## **Solution**:

```
int main(int argc, char *argv[])
{
  float *big_array;
  if(myrank == 0) {
        big_array = (float *)malloc(...)
```

If the other ranks need to know details about a variable, pointers can be created. The memory can be allocated dynamically within the correct rank.

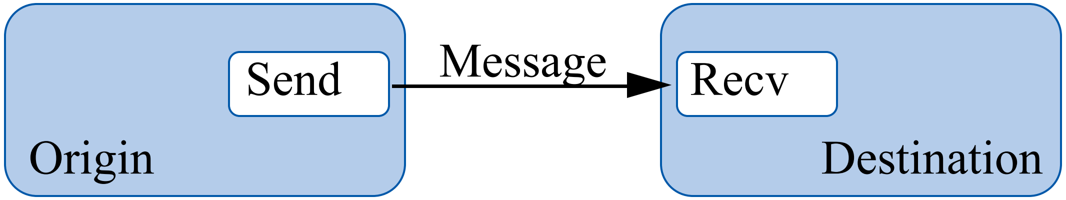
# Guidelines for Using Communication

- Try to avoid communication as much as possible: more than a factor of 100/1000 between transporting a byte and doing a multiplication
  - Often it is faster to replicate computation than to compute results on one process and communicate them to other processes.
- Try to combine messages before sending.
  - It is better to send one large message than several small ones.

# Message Passing

**Basic Functions**:

- send(parameter_list)
- recv(parameter_list)




**Send Function**:

- In origin process
- Creates message

**Receive Function**:

- In destination process
- Receives transmitted message

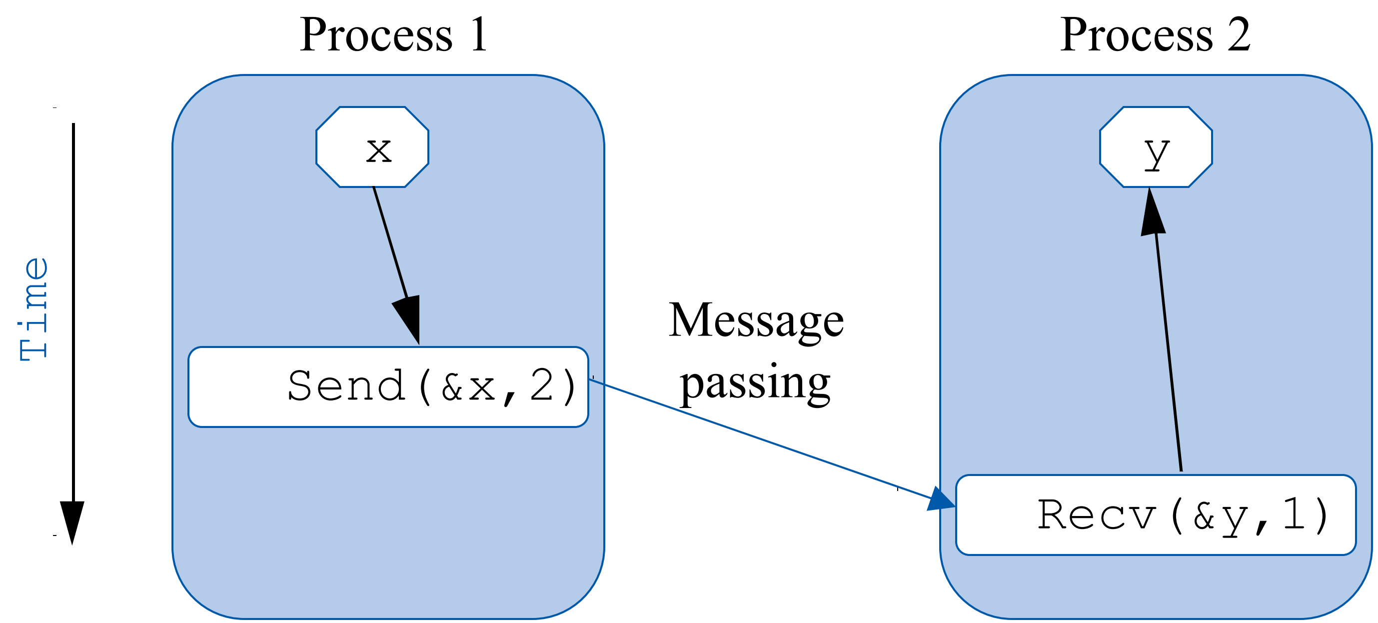
# Simple Functions

## On the origin process:

```
send(&x,destination_id)
```

## On the destination process:

```
recv(&y, source_id)
```



# Standard Send

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

- `buf` Address of message in memory
- `count` Number of elements in message
- `datatype` Data type of message
- **`dest`** Destination process of message
- `tag` Generic message tag
- `comm` Communication handler

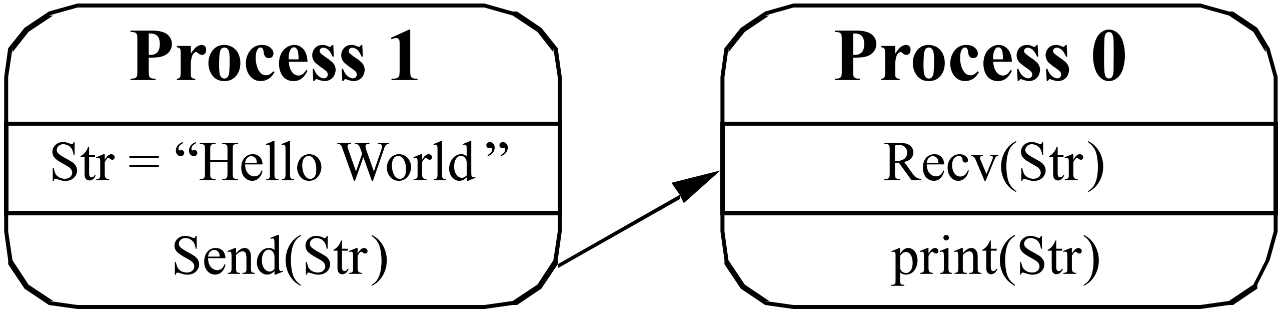
| | |
|---|---|
| `MPI_Datatype` | `MPI_CHAR,MPI_INT,MPI_FLOAT,...` |
| `MPI_Comm` | `MPI_COMM_WORLD` |

# Standard Receive

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

- `buf` Address of message im memory
- `count` *Expected* number of elements in message
- `datatype` Data type of message
- **`source`** Origin process of message
- `tag` Generic message tag
- `comm` Communication handler
- `status` Status-Information

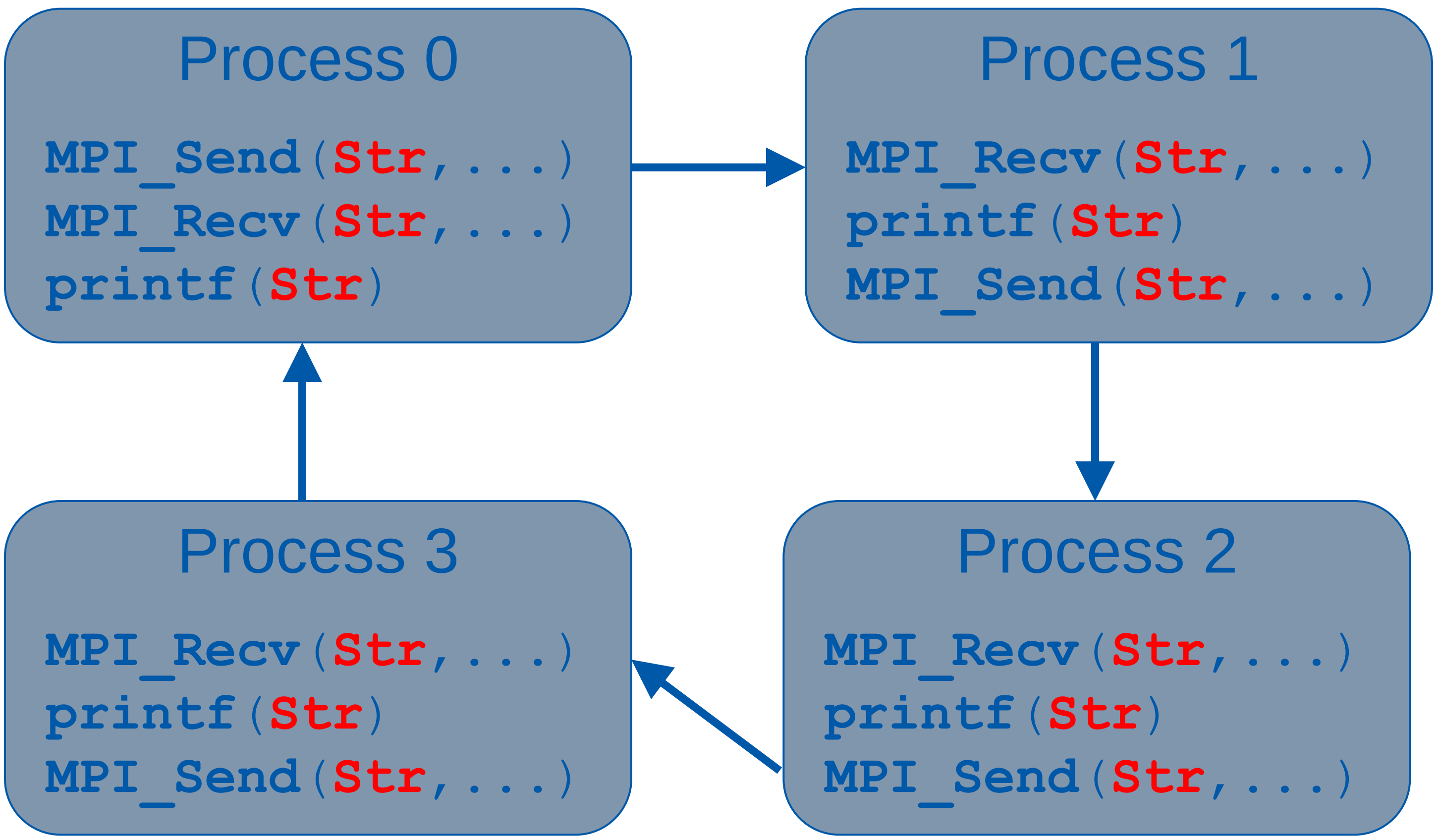
# Example: *Hello World*



# Example: *Hello World*

```
if (myrank == 1) {
    char sendStr[] = "Hello World";
    MPI_Send(sendStr, strlen(sendStr)+1, MPI_CHAR,
             0 ,3, MPI_COMM_WORLD );
    }
else {
    char recvStr[20];
    MPI_Recv(recvStr, 20, MPI_CHAR, 1, 3,
             MPI_COMM_WORLD, &stat );
    printf("%s\n",recvStr);
}
```

# Example: *Round Robin*



# Standard Receive

**Remark:**

Maximum message length is fixed:

- If message is bigger → overflow error
- If message is smaller → unused memory

→ Allocate sufficient space before calling MPI_Recv

# Standard Receive

## How many elements have been received?

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

## Status-Information:

```
struct MPI_Status {
    int    MPI_SOURCE;
    int    MPI_TAG;
    int    MPI_ERROR;
    int    count;
    ...
};
```

in case of MPI_ANY_SOURCE

in case of MPI_ANY_TAG

Number of Elements

# MPI Specifics

- Communicators: Scope of communication operations
- Structure of messages: complex data types
- Data transfer:
  - Synchronous/asynchronous
  - Blocking/non-blocking
- Message tags/identifiers
- Communication partners:
  - Point-to-point
  - Wild card process and message tags

# Communicators

## Scope of processes

- Communicator group processes
- A group defines the set of processes,
  that can communicate with each other
- Used in point-to-point and collective communication
- After starting a program, its processes subscribe to the "Universe" ==> `MPI_COMM_WORLD`
- Each program has its own "Universe"

# Usage of Communicators

- Fence off communication environment
- Example: Communication in library
  *What happens, if a program uses a parallel library that uses MPI itself?*
- 2 Kinds of communicators:
  - Intra-communicator: inside a group
  - Inter-communicator: between groups
- Processes in each group are always numbered *0* to *m-1* for *m* processes in a group

# MPI Specifics

- Communicators: Scope of communication operations
- Structure of messages: complex data types
- Data transfer:
  - Synchronous/asynchronous
  - Blocking/non-blocking
- Message tags/identifiers
- Communication partners:
  - Point-to-point
  - Wild card process and message tags

# Structure of Messages

- Standard data types:
  - Integer, Float, Character, Byte, ...
  - (Continuous) arrays
- Complex data types:
  - Messages including different data: counter + elements
  - Non-continuous data types: sparse matrices
- Solutions:
  - Pack/unpack functions
  - Special (common) data types:
    - Array of data types
    - Array of memory displacements
  - Managed by the message-passing library

# Point-to-Point Communication

## **MPI**:

- Data types for message contents:
  - Standard types:
    - `MPI_INT`
    - `MPI_FLOAT`
    - `MPI_CHAR`
    - `MPI_DOUBLE`
    - ...
  - User defined types: derived from standard types