www.bsc.es


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

# Introduction to the MPI programming model

Dr. Janko Strassburg

PATC Parallel Programming Workshop 30th June 2015

# Motivation

High Performance Computing

**Processing speed**

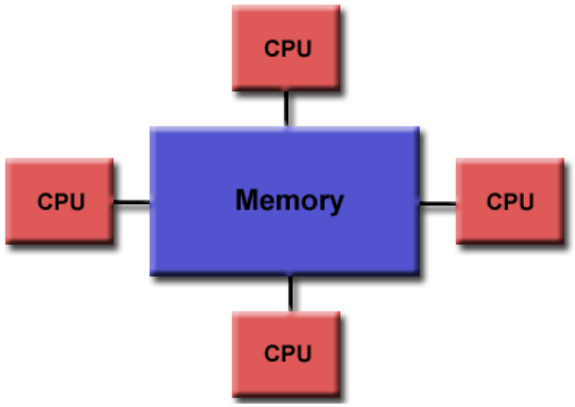
**+**

**Memory capacity**

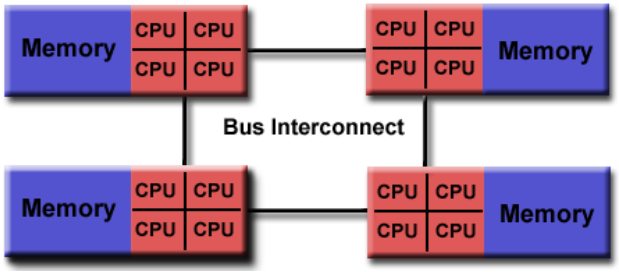
Software development for:
Supercomputers, Clusters, Grids

# 1 - Parallel computing

## SMP Machine

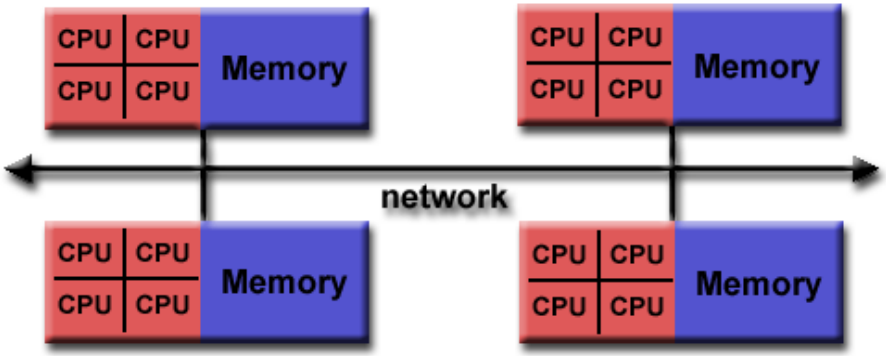



UMA




NUMA

## Cluster machine



# Programming Parallel Systems

## Possibilities

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

## Approach

- Use Fortran/C
- Function calls to message passing library

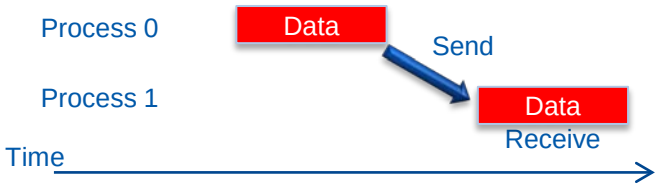
**Explicit parallelism** - User defines:

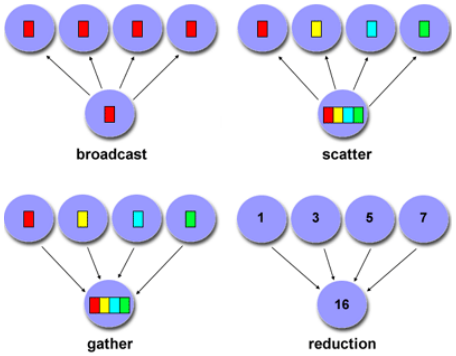
- which processes to execute,
- when and how to exchange messages, und
- which data to exchange within messages.

# Distributed memory parallelism

## Distributed memory parallelism

- MPI (Message Passing Interface) is the most used mechanism in HPC distributed memory clusters
  - http://www.mpi-forum.org/
  - http://www.mcs.anl.gov/mpi - MPI stuff, talks, tutorials, FAQ
- Processes can have *threads* in the same node
  - Sharing the same address space
- MPI is used for the communication
  - Which have separated address space
- Interprocess communication consists of
  - Synchronization
  - Moving data (P2P and Collective)



# 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

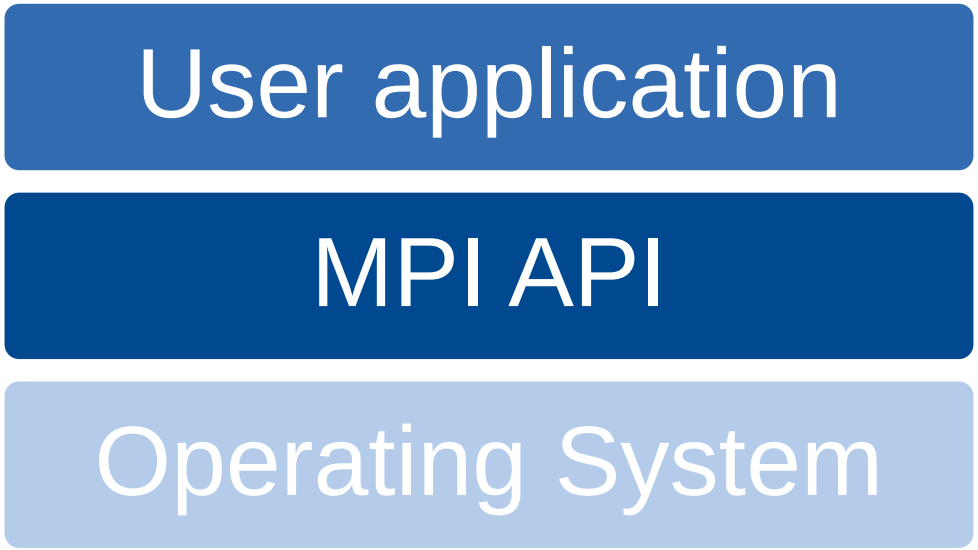
# Purpose

- 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

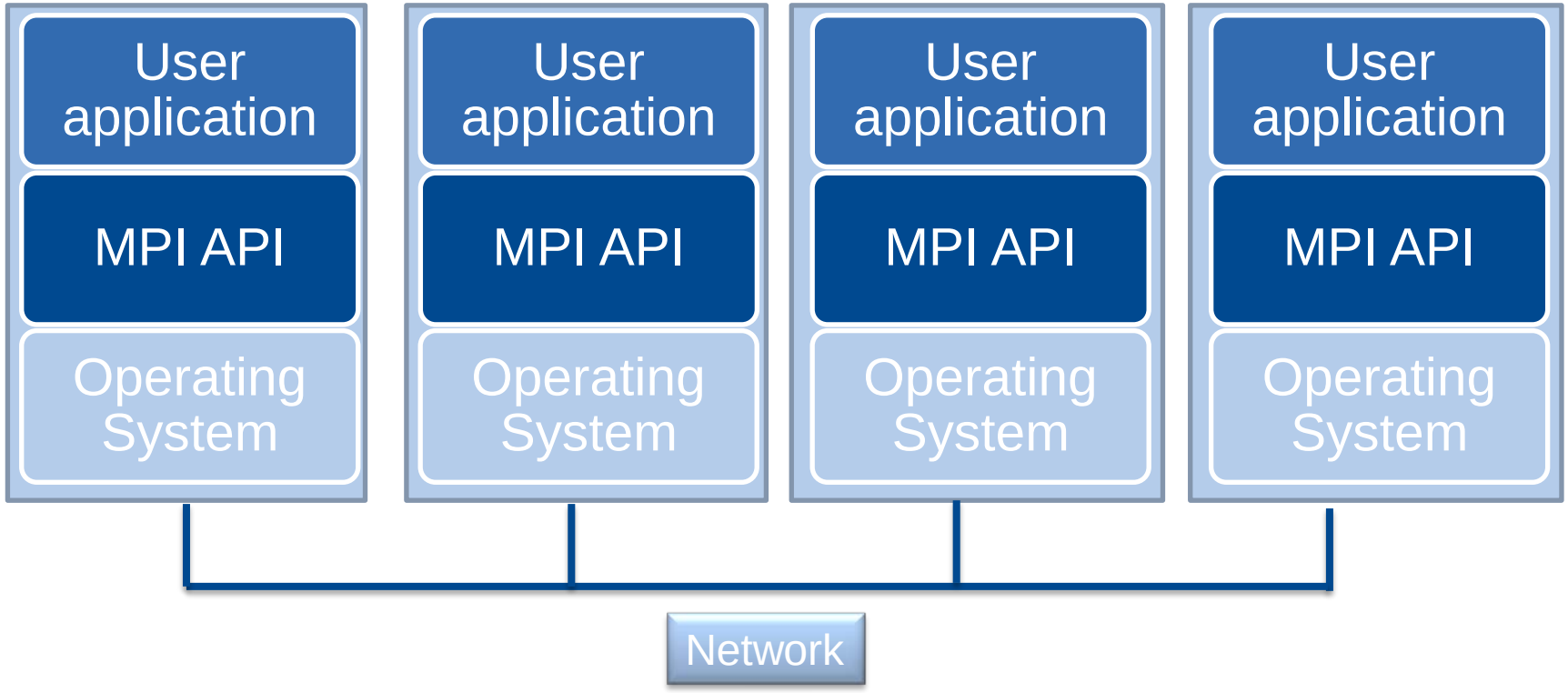
- Abstracts view of the network
  - Shared memory / Sockets
  - Ethernet / Infiniband
  - High speed communication network / High throughput data network
  - …
- Application communicates using simple commands
  - MPI_SEND / MPI_RECV
  - Implementation handles connections automatically

# MPI High-Level View

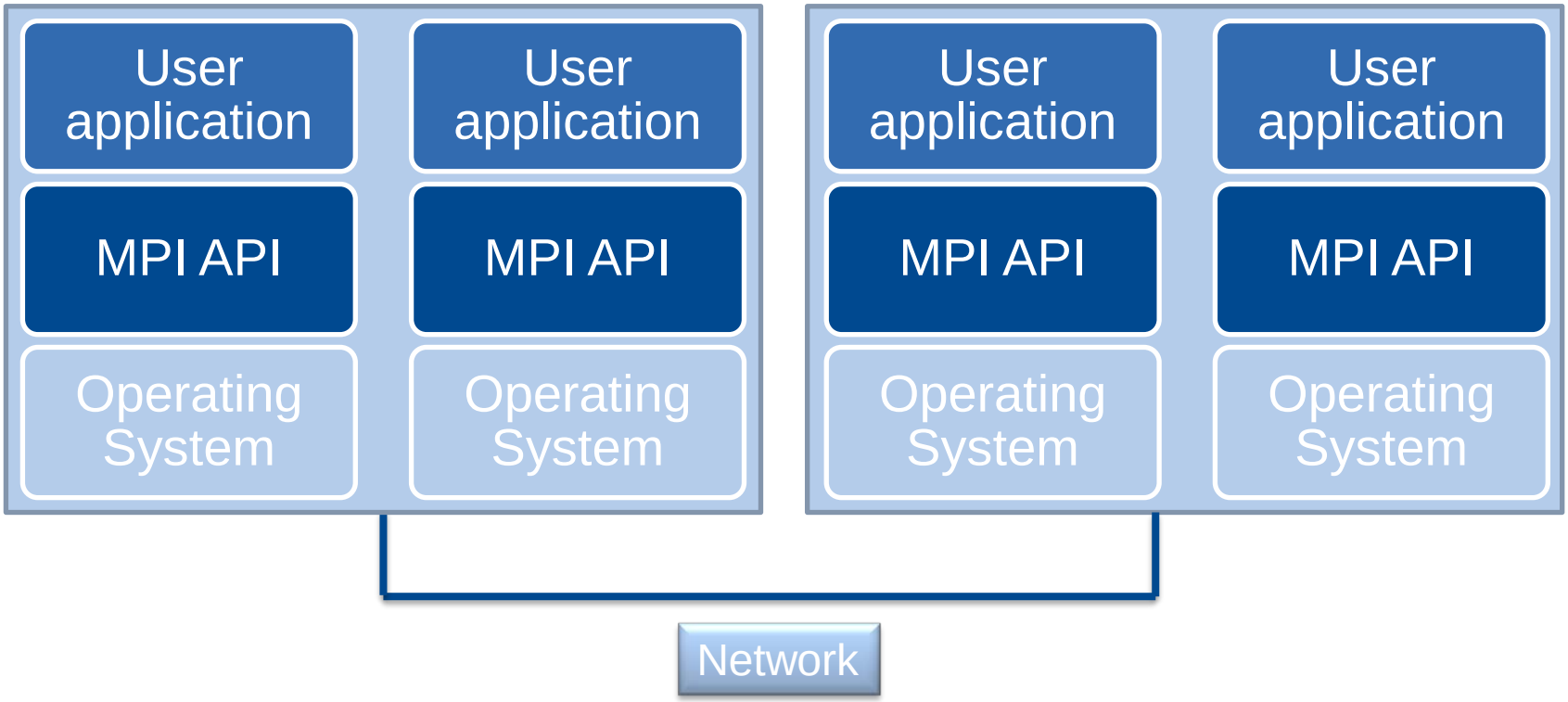



- Layered system
- Abstracting hardware from the programmer

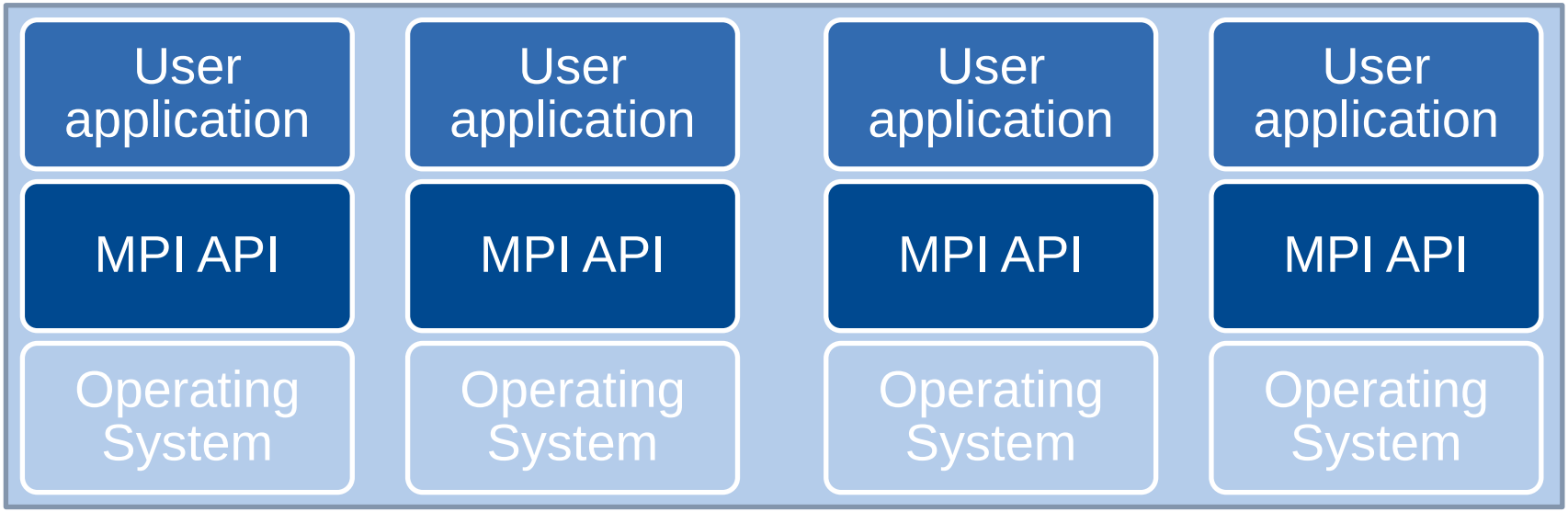
# Example: 4 Nodes



# Example: 2 Nodes



# Example: 1 Node




All processes running on the same machine

# Parallel Programming

**Possibilities**:

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

# Usage of Existing Sequential Programming Languages

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

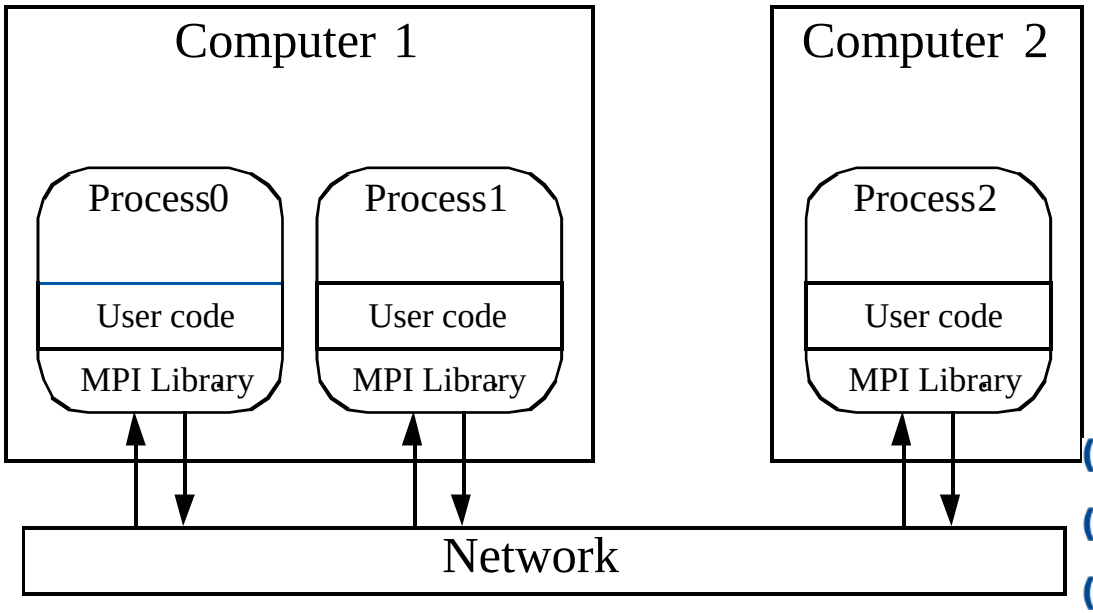
- Open MPI:
  - Combined effort from FT-MPI, LA-MPI, LAM/MPI, PACX-MPI
  - De facto standard; used on many TOP500 systems
- MPICH
- CHIMP
- LAM
- FT-MPI
- Vendor specific implementations:
  - Bull, Fujitsu, Cray, IBM, SGI, DEC, Parsytec, HP, ...

# 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

```
void main()
{
  int i,j;
  char a;
  for(i=0;
  ...
```


**Execution:**
n Executables

```
void main()
{
  int i,j;
  char a;
  for(i=0;
  ...
```

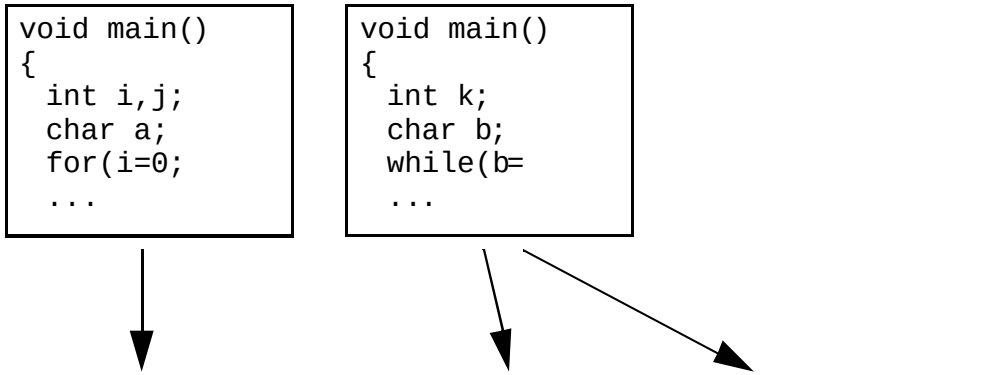
```
void main()
{
  int i,j;
  char a;
  for(i=0;
  ...
```

```
void main()
{
  int i,j;
  char a;
  for(i=0;
  ...
```

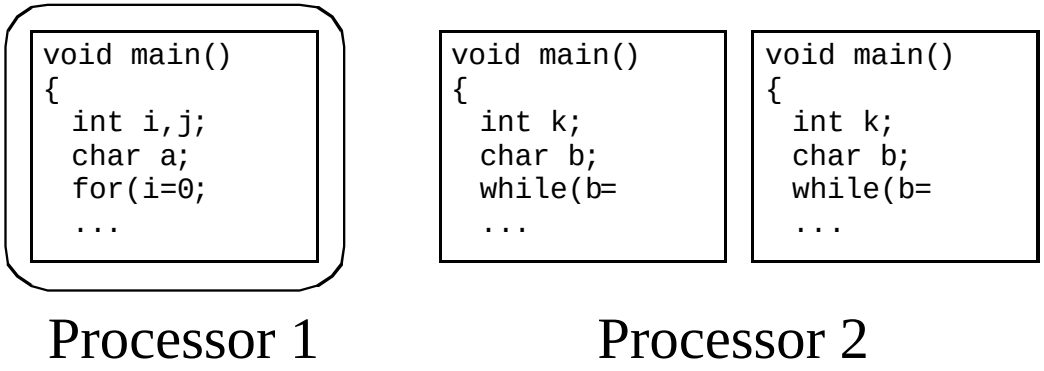
Processor 1 Processor 2

# 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`

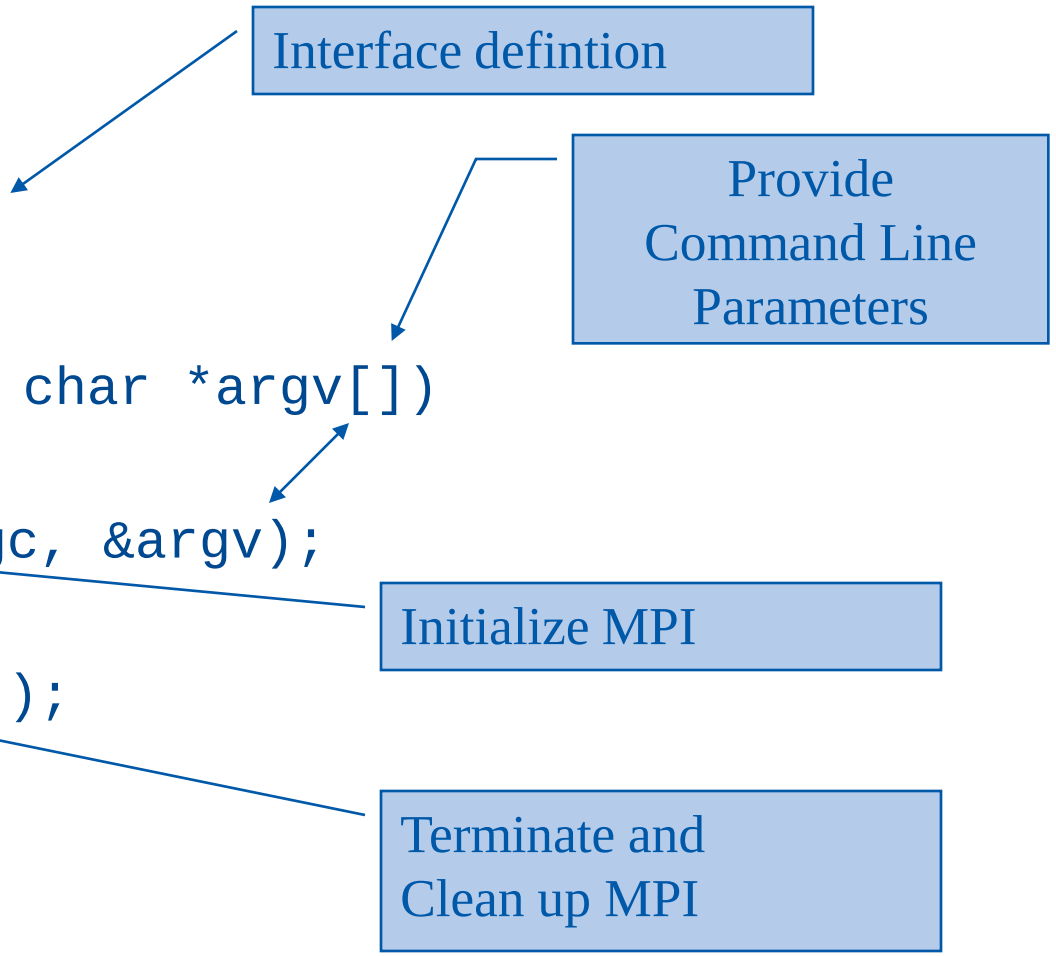
  Execute: `mpirun -np <proc> <exec>`

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

# Basic-Code-Fragment

## Initialization and Exit:

```
1.  #include <mpi.h>
2.  ...
3.  int main(int argc, char *argv[])
4.  {
5.        MPI_Init(&argc, &argv);
6.  ...
7.        MPI_Finalize();
8.  }
```



# Structure of MPI Functions

**General**:

```
1.  result = MPI_Xxx(…);
```

**Example**:

```
1.  result = MPI_Init(&argc, &argv);
2.  if(result!=MPI_SUCCESS) {
3.     fprintf(stderr,”Problem”);
4.     fflush(stderr);
5.     MPI_Abort(MPI_COMM_WORLD,result);
6.  }
```

# 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.

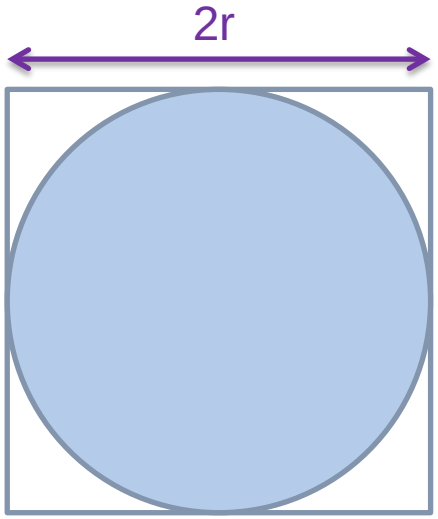
- Preliminaries:
  Statements to distinguish processes and the subsequent code execution

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

# 1.1 – Example

- MonteCarlo MPI approximation
  - Inscribe a circle inside a square
  - Generate random points within the square
  - Determine the number of points inside circle
  - If N is
    - The number of points in the circle divided by
    - The number of points in the square
  - Then $\pi$ = 4*N
  - The more number of generated points the

Better approximation




$$As = (2r)^2$$
$$Ac = \pi r^2$$
$$\pi = 4 * \frac{Ac}{As}$$

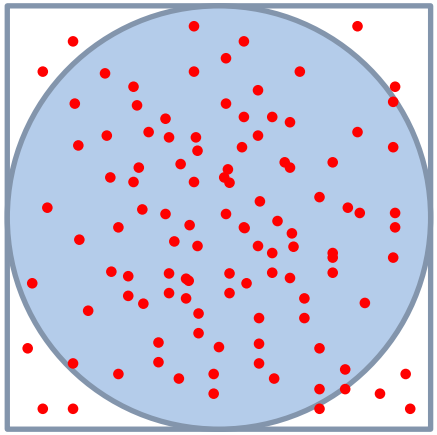
# 1.1 – Example

## MonteCarlo sequential pseudocode

```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```



$$As = (2r)^2$$
$$Ac = \pi r^2$$
$$\pi = 4 * \frac{Ac}{As}$$

# 1.1 – Example

- MonteCarlo “embarrassingly parallel” solution
  - Divide the number of iterations into chunks that can be executed by different tasks simultaneously.
  - Each task executes its portion of the loop a number of times.
  - Each task can do its work without requiring any information from the other tasks (there are no data dependencies).
  - Master task receives results from other tasks.

# 1.1 – Example

- MonteCarlo parallel pseudocode
  - Parts in yellow are the changes

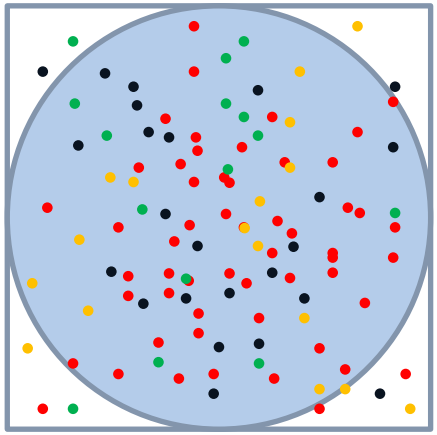
```
npoints = 10000
circle_count = 0

p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER
  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)
else if I am WORKER
  send to MASTER circle_count
endif
```



# 1.1 – Example

## Small MPI code example

```
#include “mpi.h”
#include <stdio.h>

int main (int argc, char **argv)
{
  int rank, size;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank ( MPI_COMM_WORLD, &rank ); /* Who am I? */
  MPI_Comm_size ( MPI_COMM_WORLD, &size ); /* How many are we? */
  printf ( “I am %d of %d\n”, rank, size );
  MPI_Finalize();
  return 0;
}
```

```
bsc99002@login3-mn3:~/examples> mpicc -o hello-world mpi-hello-world.c
bsc99002@login3-mn3:~/examples> mpirun -np 4 ./hello-world
I am 0 of 4
I am 3 of 4
I am 1 of 4
I am 2 of 4
```

# Master-Slave Program

```
1.  int main(int argc, char *argv[])
2.  {
3.     MPI_Init(&argc, &argv);
4.     ...
5.     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6.     if(myrank == 0)
7.           master();
8.     else
9.           slave();
10.    ...
11.    MPI_Finalize();
12. }
```

# Global Variables

**Problem**:

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

**Solution**:

```
1.  int main(int argc, char *argv[])
2.  {
3.     if(myrank == 0) {
4.           float big_array[10000];
```

Allocate large arrays only on the ranks needed to save memory

# Global Variables

**Problem**:

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

**Solution**:

```
1.  int main(int argc, char *argv[])
2.  {
3.    float *big_array;
4.    if(myrank == 0) {
5.          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.

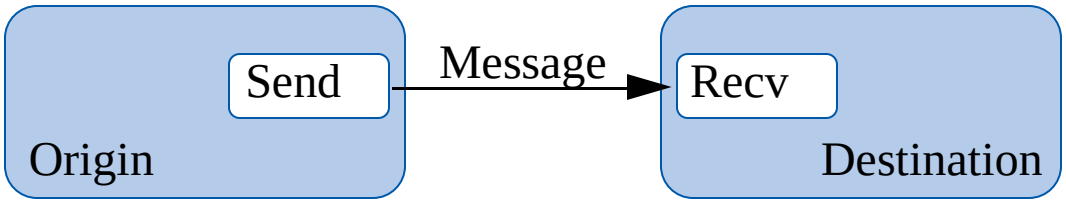
# Guidelines for Using Communication

- 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

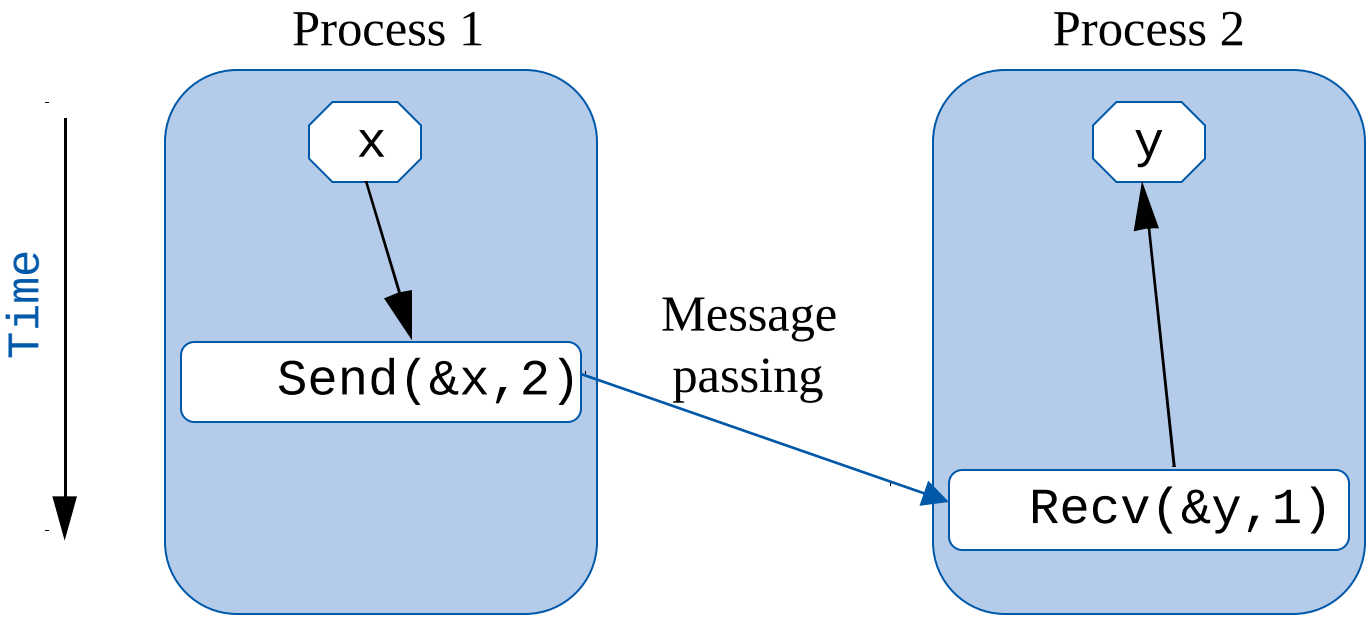
# Simple Functions

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

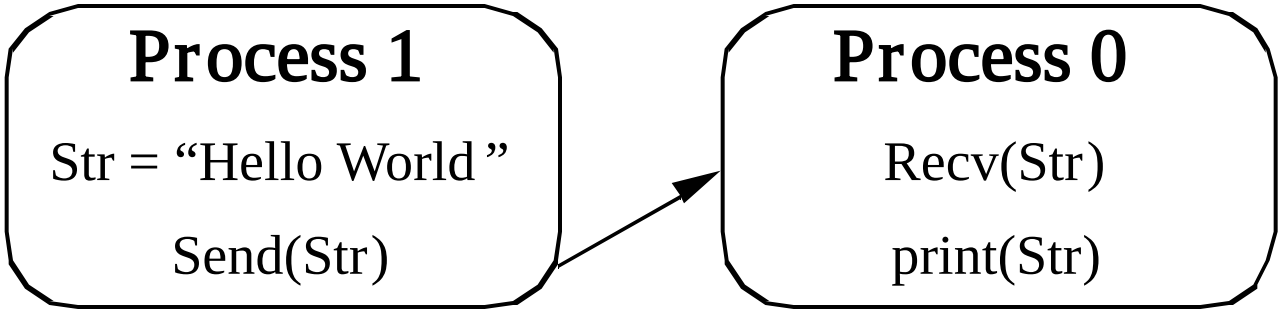
```
MPI_Datatype   MPI_CHAR,MPI_INT,MPI_FLOAT,...
MPI_Comm       MPI_COMM_WORLD
```

# 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

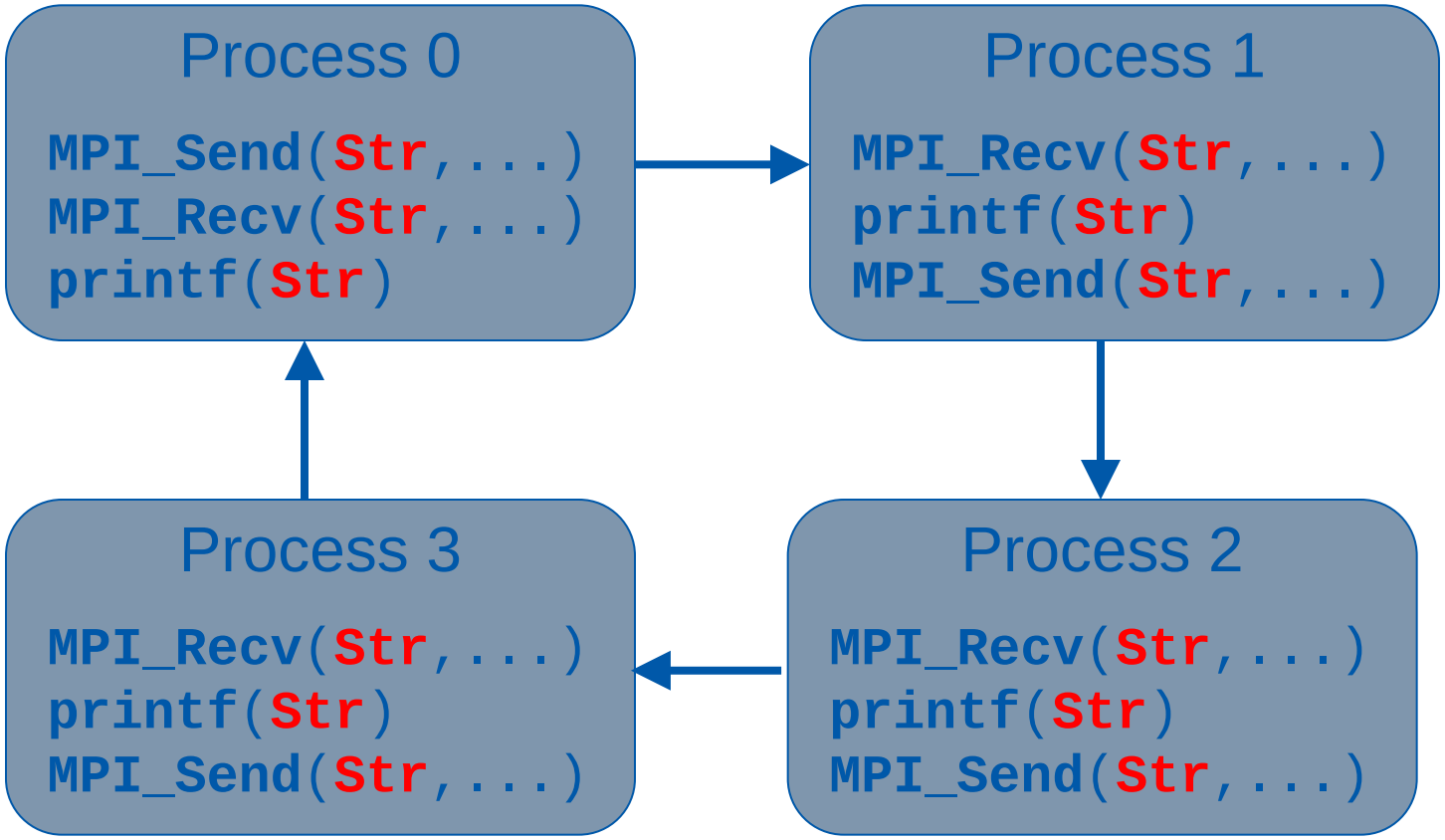
# Example: *Hello World*



# Example: *Hello World*

```
1.  if (myrank == 1) {
2.     char sendStr[] = “Hello World”;
3.     MPI_Send(sendStr, strlen(sendStr)+1, MPI_CHAR,
4.            0 ,3, MPI_COMM_WORLD );
5.     }
6.  else {
7.     char recvStr[20];
8.     MPI_Recv(recvStr, 20, MPI_CHAR, 1, 3,
9.            MPI_COMM_WORLD, &stat );
10.    printf(“%s\n”,recvStr);
11. }
```

# 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

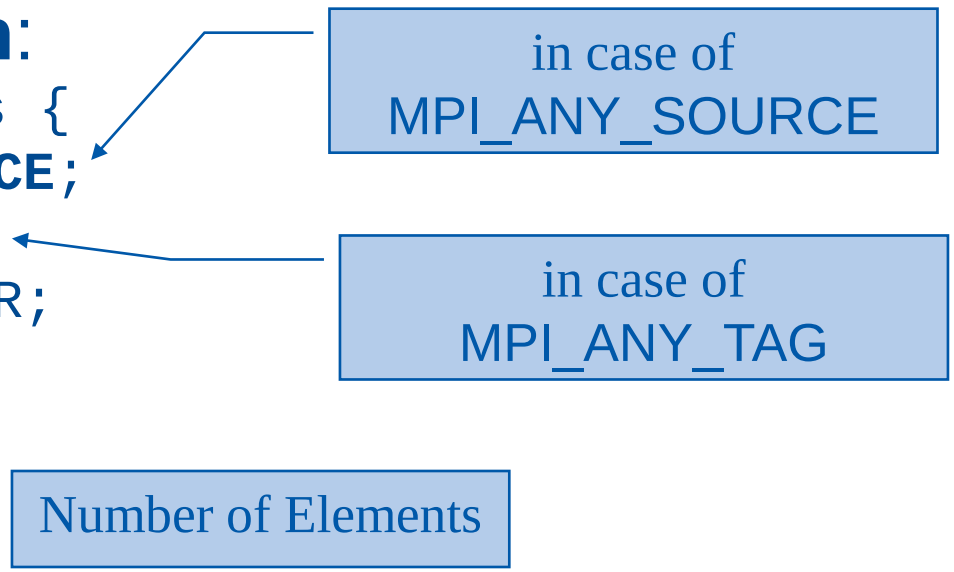
# Standard Receive

## How many elements have been received?

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

## Status-Information:

```
1.  struct MPI_Status {
2.     int   MPI_SOURCE;
3.     int   MPI_TAG;
4.     int   MPI_ERROR;
5.     int   count;
6.     ...
7.  };
```



# Minimum Set of Functions?

*For an arbitrary MPI program,*
*only **6 Functions** are needed*

- `MPI_Init(...)`
- `MPI_Finalize(...)`
- `MPI_Comm_rank(...)`
- `MPI_Comm_size(...)`
- `MPI_Send(...)`
- `MPI_Recv(...)`

# 1.2 – MPI Implementations @ BSC

- How to compile MPI programs

| LANGUAGE | Commad |
|---|---|
| C | mpicc |
| C++ | mpiCC/mpicxx/mpic++ |
| Fortran 77 | mpif77/mpifort |
| Fortran 90 | mpif90/mpifort |

- All the backend compilers are managed by environment variables (modules intel and gnu)
- Several MPI implementations available
  - Intel MPI
  - OpenMPI
  - MVAPICH2
  - IBM Parallel Environment (PE)

# 1.2 – INTEL MPI

| **Version** | **4.1.3.049** |
|---|---|
| Module | module load impi |
| Description | Intel® MPI Library 4.1 focuses on making applications perform better on Intel® architecture-based clusters—implementing the high performance Message Passing Interface Version 2.2 specification on multiple fabrics. |
| Web | http://software.intel.com/en-us/intel-mpi-library |
| Location | /apps/INTEL/impi/4.1.3.049/bin64 |
| Features | Fully compatible with LSF + POE |
| Benchmarks | /apps/INTEL/mkl/benchmarks/mp_linpack/bin_intel/intel64/ |
| Launcher | mpirun / poe |

# 1.2 – OpenMPI

| | |
|---|---|
| **Versions** | **1.4.6 (/usr/mpi/gcc/openmpi-1.4.6)**<br>**1.6.4**<br>**1.6.4-valgrind**<br>**1.7.2**<br>**1.8.1 (default)**<br>**1.8.1-multithread**<br>**…** |
| Module | module load openmpi[/version] |
| Description | The Open MPI Project is an open source MPI-2 implementation that is developed and maintained by a consortium of academic, research, and industry partners. |
| Web | http://www.open-mpi.org/ |
| Location | /apps/OPENMPI/<version>/bin |
| Features | Fully compatible with LSF |
| Launcher | mpirun |

# 1.2 – IBM Parallel Environment (POE)

| **Version** | **1303** |
|---|---|
| Module | module load poe |
| Description | IBM PE Developer Edition provides an Integrated Development Environment (IDE) that combines open source tools, and IBM-developed tools to fulfill the programmer requirements. |
| Web | http://www.redbooks.ibm.com/abstracts/sg248075.html?Open |
| Location | /opt/ibmhpc/pecurrent/base//bin |
| Features | Full compatible with LSF |
| Launcher | poe |

# 1.2 – MVAPICH2

| **Version** | **1.8.1 (default)**<br>**2.0.1**<br>**2.0.1-multithread** |
|---|---|
| Module | module load mvapich2[/version] |
| Description | MPI over Infiniband. It implements MPI 3.0 standard. |
| Web | http://mvapich.cse.ohio-state.edu/ |
| Location | /apps/MVAPICH2/<version>/ |
| Features | Compatible LSF |
| Launcher | mpirun |

# 1.3 – How to switch MPI implementations

- OpenMPI 1.8.1 is loaded by default
- You can change your environment easily using ‘module’ commands
- OpenMPI -> IntelMPI

```
bsc99002@login3-mn3:~> module unload openmpi
remove openmpi/1.8.1 (PATH, MANPATH, LD_LIBRARY_PATH)
bsc99002@login3-mn3:~> module load impi
load impi/4.1.3.049 (PATH, MANPATH, LD_LIBRARY_PATH)
bsc99002@login3-mn3:~> module list
Currently Loaded Modulefiles:
  1) intel/13.0.1     2) transfer/1.0     3) impi/4.1.3.049
bsc99002@login3-mn3:~> mpicc -show
icc -m64 -D__64BIT__ -Wl,--allow-shlib-undefined -Wl,--enable-new-dtags -Wl,-
rpath,/opt/ibmhpc/pecurrent/mpich2/intel/lib64  -I/opt/ibmhpc/pecurrent/mpich2/intel/include64 -
I/opt/ibmhpc/pecurrent/base/include -L/opt/ibmhpc/pecurrent/mpich2/intel/lib64 -lmpi -ldl -L -lirc -
lpthread -lrt
```

- You can not have 2 MPI implementations loaded at the same time (command will not permit loading conflicting modules)

# 1.3 – How to switch MPI implementations

## Switch between OpenMPI versions

```
bsc99002@login3-mn3:~> module switch openmpi/1.8.1 openmpi/1.6.4
switch1 openmpi/1.8.1 (PATH, MANPATH, LD_LIBRARY_PATH)
switch2 openmpi/1.6.4 (PATH, MANPATH, LD_LIBRARY_PATH)
switch3 openmpi/1.8.1 (PATH, MANPATH, LD_LIBRARY_PATH)
ModuleCmd_Switch.c(243):VERB:4: done
Bsc99002@login3-mn3:~> module list
Currently Loaded Modulefiles:
  1) intel/13.0.1    2) transfer/1.0    3) openmpi/1.6.4
bsc99002@login3-mn3:~> mpicc -show
icc -I/apps/OPENMPI/1.6.4/include -pthread -L/apps/OPENMPI/1.6.4/lib -lmpi -ldl -lm -lnuma -Wl,--
export-dynamic -lrt -lnsl -lutil -lm -ldl
```

## Remember to use the same module to compile and execute

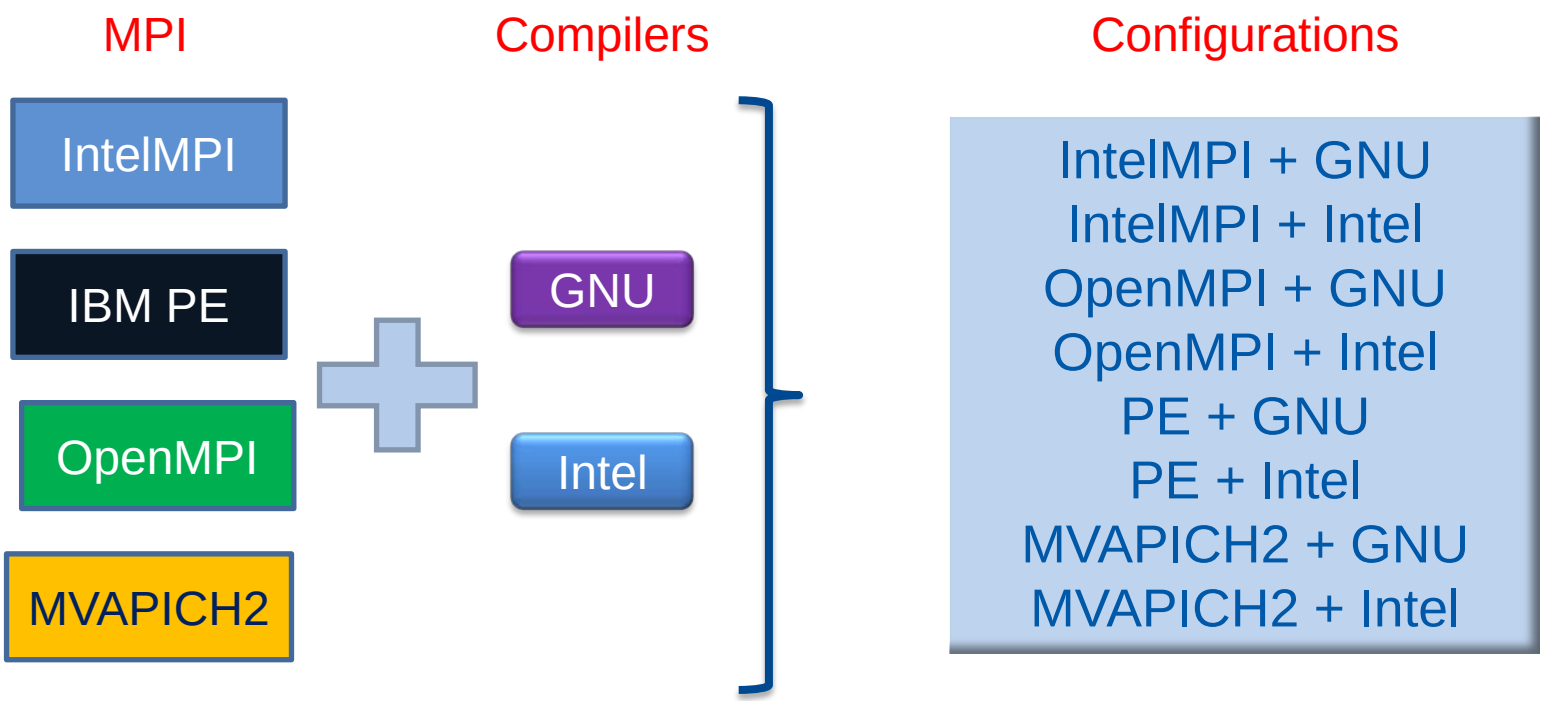
- Otherwise the job will fail or not behave properly

```
./hello_world: error while loading shared libraries: libmpi.so.1: cannot open shared object file:
No such file or directory

bsc99002@login3-mn3:~> mpirun -np 4 ./hello_world
Hello world! I'm process 0 of 1 on login3
Hello world! I'm process 0 of 1 on login3
Hello world! I'm process 0 of 1 on login3
Hello world! I'm process 0 of 1 on login3
```

# 1.3 – Application advice

- There is no ‘best’ MPI implementation in MareNostrum
  - If you encounter problems with > 2048 cores it is recommend to launch with POE using IntelMPI or POE



# 1.4 - Possible issues

- Memory problems
  - Be careful with the memory consumption of your job
  - 28GB per node
- If your program tries to use more memory, then LSF will send SIGKILL to your processes
  - You will see: mpirun: killing job…
- You can avoid this problem:
  - Set system limit on the memory

```
ulimit -m 1700000
bsub < job.lsf
```

  - Trying to allocate less tasks per node using -R”span[ptile=N]”
- For executions up to 64 nodes, there are medium/fat nodes:
  - #BSUB –M 3000
  - #BSUB –M 7000

# 1.4 - Running jobs

- How to decide which number of cores to use?
  - Depends on the scalability of your code
  - Avoid wasting resources!! (+70%)