EuroCC@Greece

HPC Training Series

HPC for beginners: basic concepts, MPI and OpenMP

Introduction to MPI (Message Passing Interface)

Antony Spyropoulos

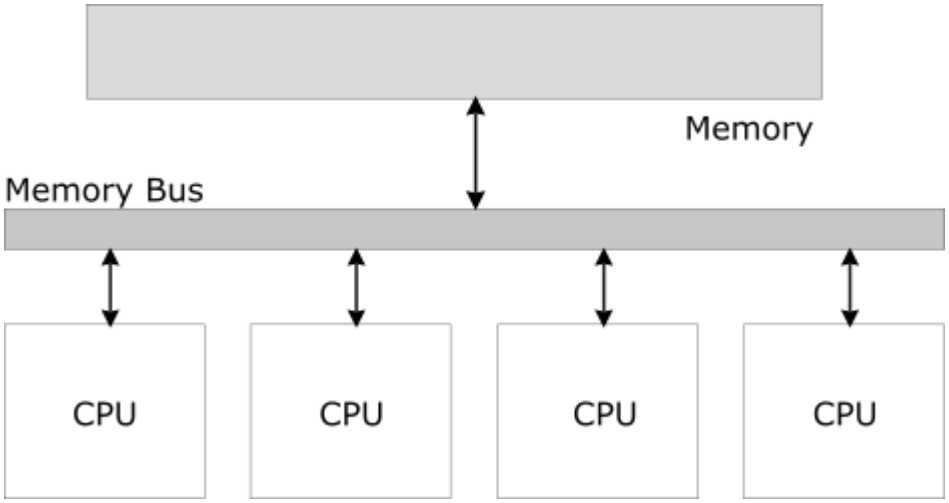
Laboratory Teaching Stuff, Computer Center,

School of Chemical Engineering, NTUA

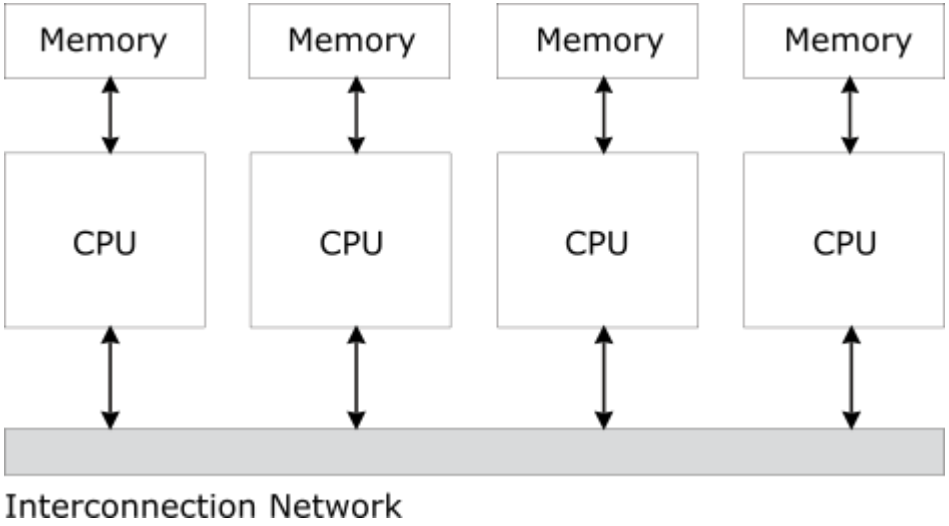
email: aspyr@chemeng.ntua.gr

Parallel Computer Architectures (1/2)

Shared Memory

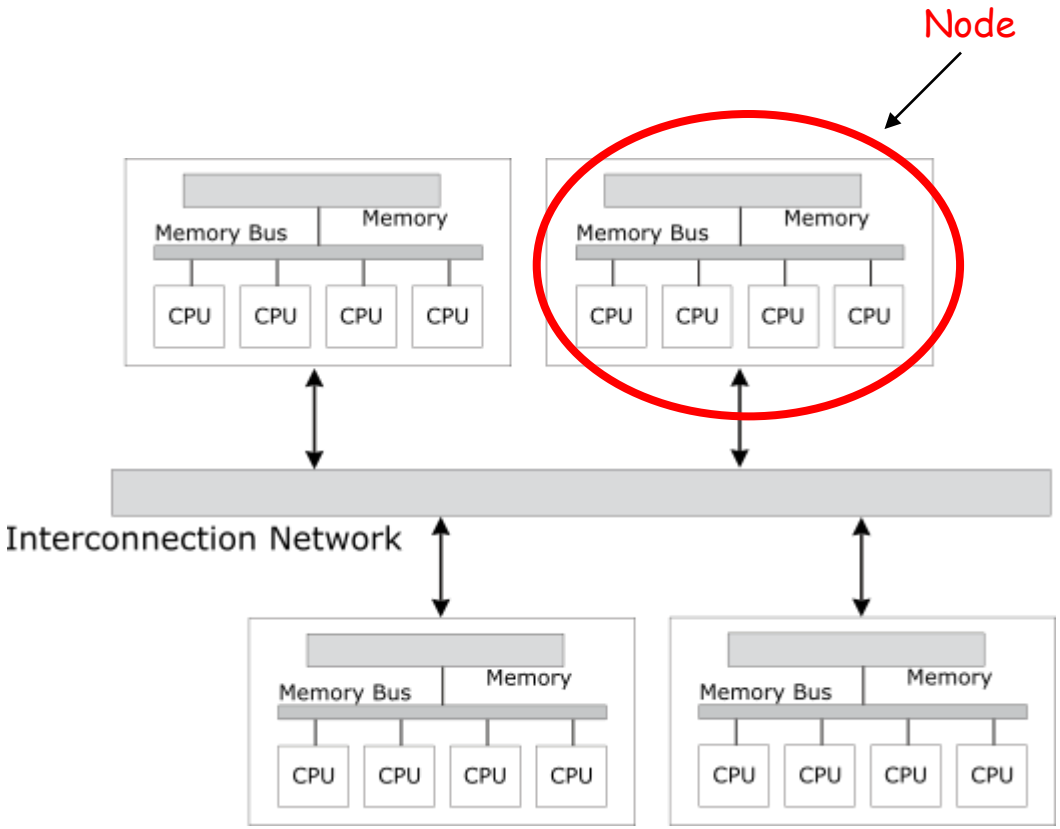


Distributed Memory

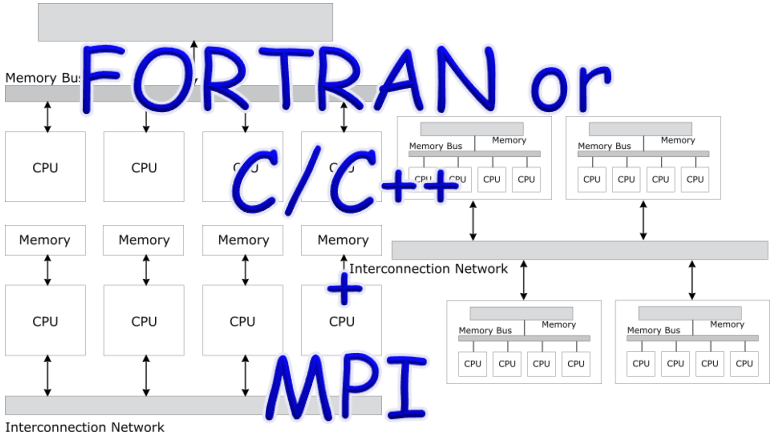


Parallel Computer Architectures (2/2)

Clusters



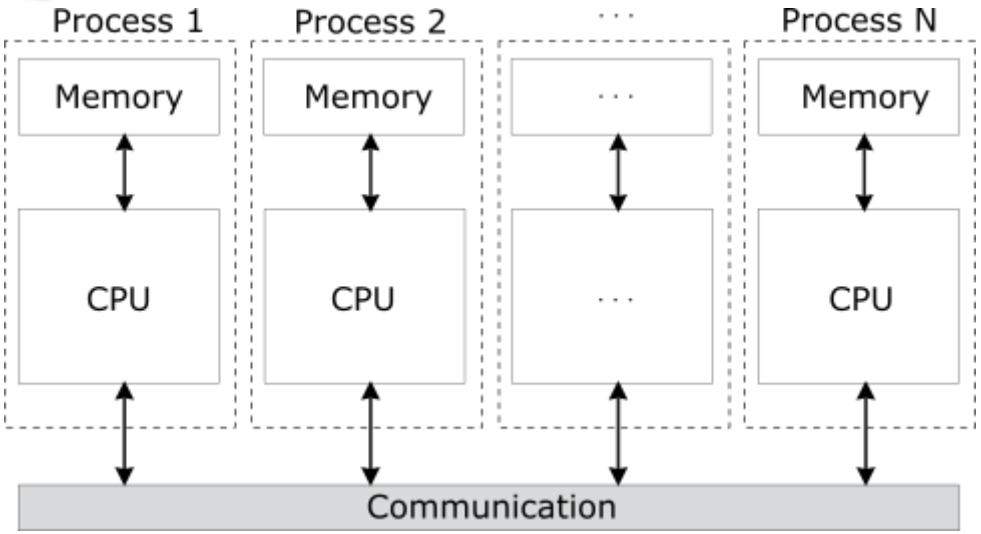
Programming parallel computers with MPI



Separate address space

Messages can be sent over network, shared memory, etc.

Virtual topology



MPI (Message Passing Interface)

MPI is a standard → message-passing library interface specification

We need an implementation of MPI before we can start coding

MPI is a language-independent communication protocol

MPI is used to send messages from one process to another

Messages contain data → primitive types (integers, strings …) or objects

Implementations of MPI

Open MPI → https://www.open-mpi.org/

MPICH → https://www.mpich.org

LAM/MPI

FT-MPI

LA-MPI

MPICH-V

MPI/FT

MetaMPICH

Scali-MPI

MPICH-GM

MPICH-SCore

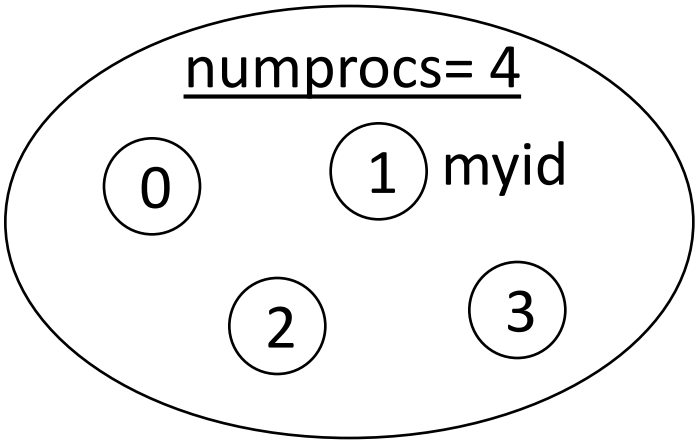
MPI-BIP

MPI/Pro

…

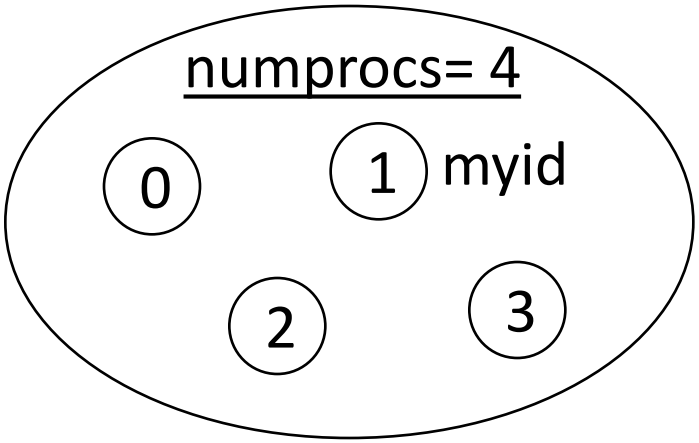
Vendor implementations: SGI-MPI, Sun-MPI, HP-MPI, NEC-MPI, Fujitsu-MPI, Cray-MPI, Hitachi-MPI, IBM-MPI …

MPI : Start - Finish




```
include 'mpif.h'
.
.
.
call MPI_INIT(error)

call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)

call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)
.
.
.
call MPI_FINALIZE(error)
```

MPI : Start - Finish




```
#include <mpi.h>
.
.
.
MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &myid);

MPI_Comm_size(MPI_COMM_WORLD, &numprocs)
.
.
.
MPI_Finalize()
```

MPI : Compilation and Running

Compile: **mpif90** myprogram.f90 (Fortran)
mpic++ myprogram.cpp (C++)

Run: **mpirun** -np **n** ./a.out

This will run **n** copies of `a.out` (`numprocs = n`)

Example 1: My first MPI code

```
program hello
implicit NONE
include 'mpif.h'
integer myid, numprocs, error

call MPI_INIT(error)

call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)

print *, 'Hello world from ', myid

call MPI_FINALIZE(error)

end
```

Example 2: My second code in MPI

```
program hello2
implicit NONE
include 'mpif.h'

integer myid, numprocs, error

call MPI_INIT(error)
if(error /= MPI_SUCCESS) then
  print *, 'Error starting MPI program'
  call MPI_ABORT(error)
  stop
endif

call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)

print *, 'My ID:', myid
print *, 'Number of processes:', numprocs
if(myid==0) print *, 'Hello world'

call MPI_FINALIZE(error)

end
```

MPI communication types

Two types of communication:

Collective
Participation of all processes in a communicator (e.g. MPI_COMM_WORLD)

Point-to-Point
Messages are sent between two processes

Collective communication
REDUCE, ALLREDUCE

```
MPI_REDUCE(sendbuf, recvbuf, count, type, op, root, comm, error)

MPI_ALLREDUCE(sendbuf, recvbuf, count, type, op, comm, error)
```

comm: MPI_COMM_WORLD

type: MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX, MPI_LOGICAL, MPI_CHARACTER

op: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD

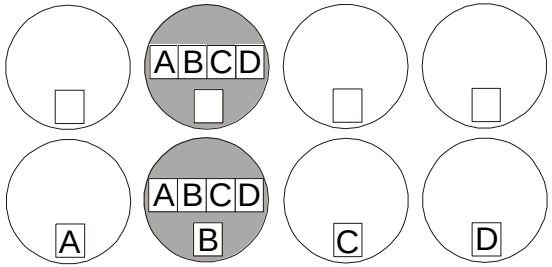
Example 3: MPI_ALLREDUCE

```
program allreduce
implicit NONE
include 'mpif.h'
integer myid, numprocs, error
real a, s

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)

a = myid + 1

call MPI_ALLREDUCE(a, s, 1, MPI_REAL, MPI_SUM, MPI_COMM_WORLD, error)

print *, s

call MPI_FINALIZE(error)

end
```

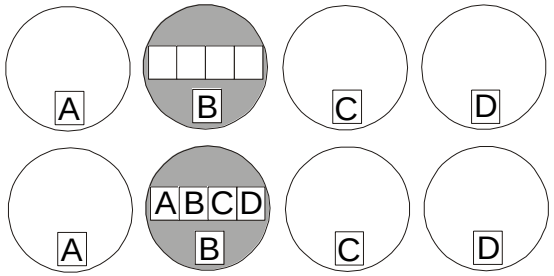
Collective communication
Broadcast, Scatter and Gather

```
MPI_BCAST(buffer, count, type, root, comm, error)
```

```
MPI_SCATTER(sendbuf, sendcount, sendtype,
            recvbuf, recvcount, recvtype,
            root, comm, error)
```




```
MPI_GATHER(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype,
           root, comm, error)
```



Example 4: MPI_SCATTER

```
program scatter
implicit NONE
include 'mpif.h'
integer, parameter :: size=4
integer myid, numprocs, error
integer sendcount, recvcount, source
real, dimension(size,size) :: sendbuf
real, dimension(size)      :: recvbuf

data sendbuf / 1.0,  2.0,  3.0,  4.0,  5.0,  6.0,  7.0,  8.0, &
               9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0 /

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)

if (numprocs==size) then
  source = 1
  sendcount = size
  recvcount = size
  call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, &
                   recvbuf, recvcount, MPI_REAL, &
                   source, MPI_COMM_WORLD, error)
  print *, 'Task ID = ',myid,' recvbuf: ',recvbuf
else
  if (myid==0) print *, 'Error: numprocs/=4'
endif

call MPI_FINALIZE(error)

end
```

Point-to-Point communication

One process sends a message (some data) and the other receives it.

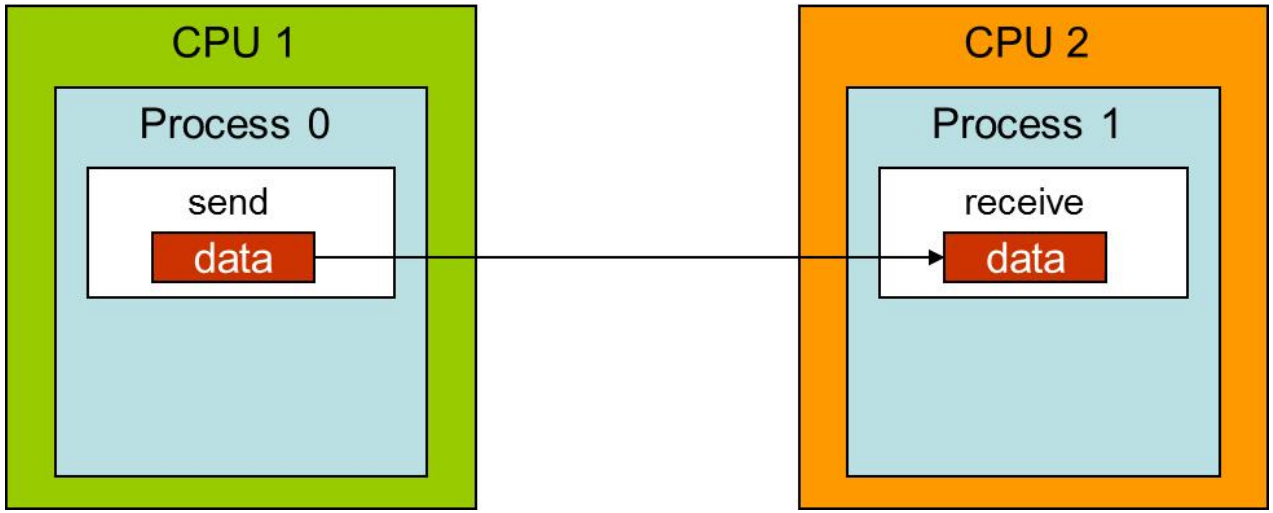
Four communication modes

- synchronous
- buffered
- standard
- ready

All four modes exist in both blocking and non-blocking forms

	Blocking form	non-Blocking form
Synchronous send	MPI_SSEND	MPI_ISSEND
Buffered send	MPI_BSEND	MPI_IBSEND
Standard send	MPI_SEND	MPI_ISEND
Ready send	MPI_RSEND	MPI_IRSEND
Receive	MPI_RECV	MPI_IRECV

Point-to-Point communication

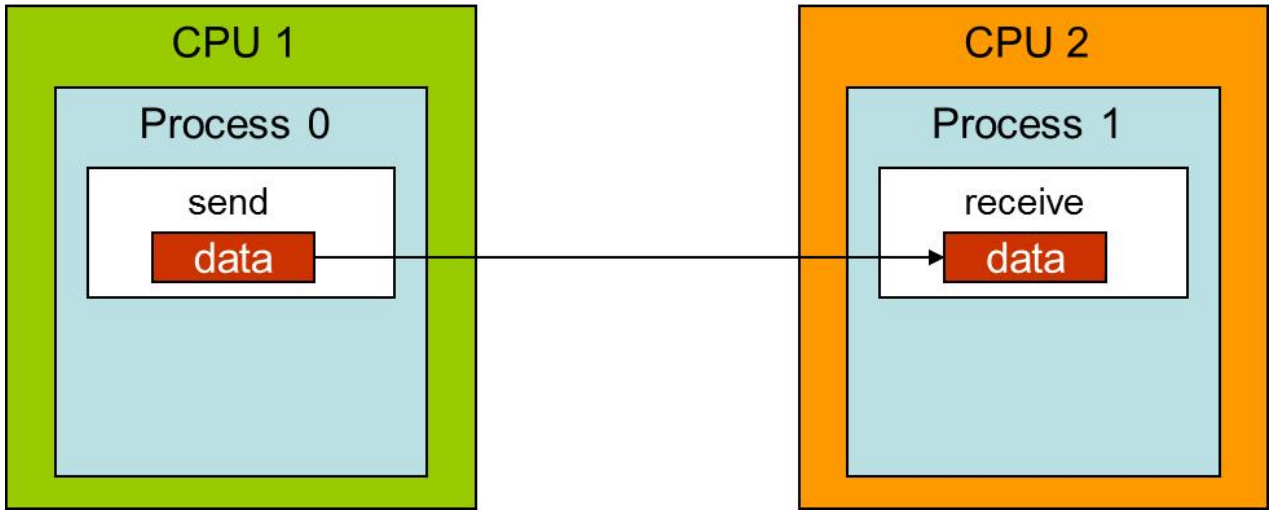


Both processes must know:

1) The source or the destination of the message
2) The message identification tag
3) The type and the amount of data that are being passed

Communication time

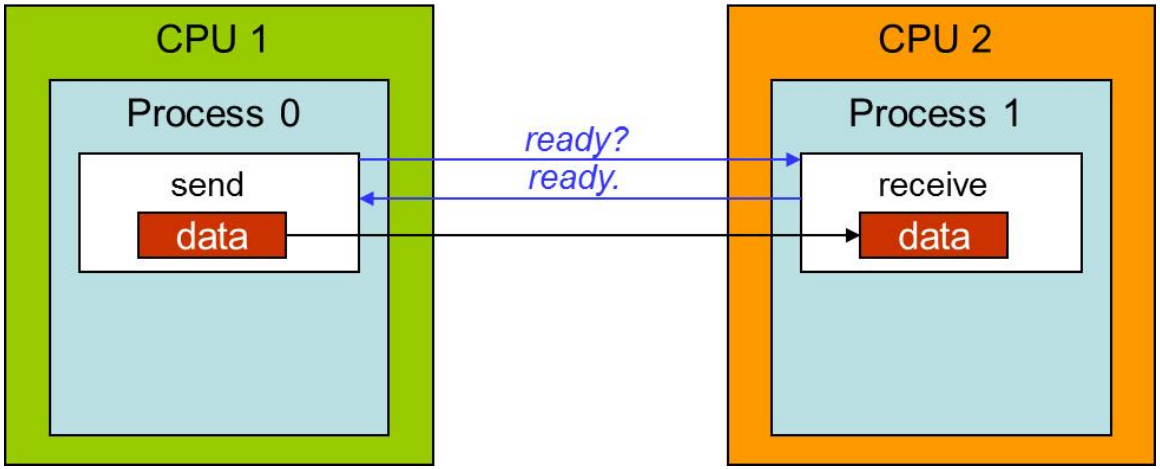
Communication time = System overhead + Synchronization overhead

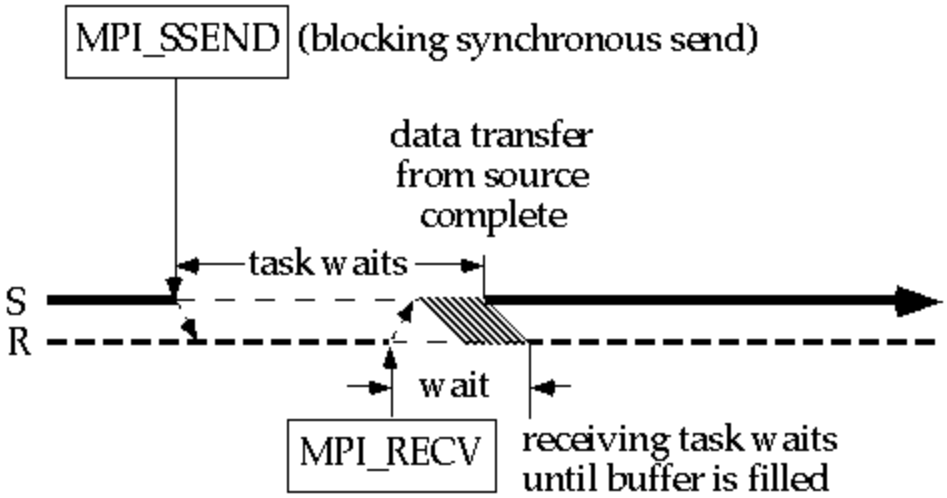


System overhead:
The time spent for data transfer (We need fast network!)

Synchronization overhead:
The time spent waiting for the other process
(We need good programming practices!)

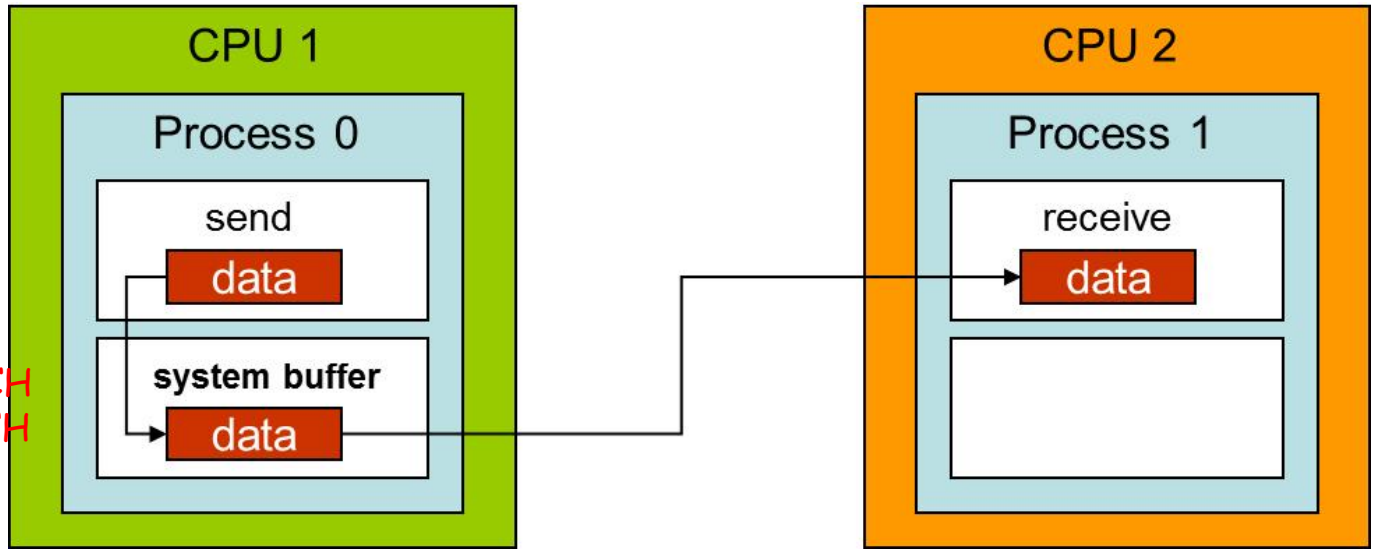
Blocking Synchronous Send (MPI_SSEND)



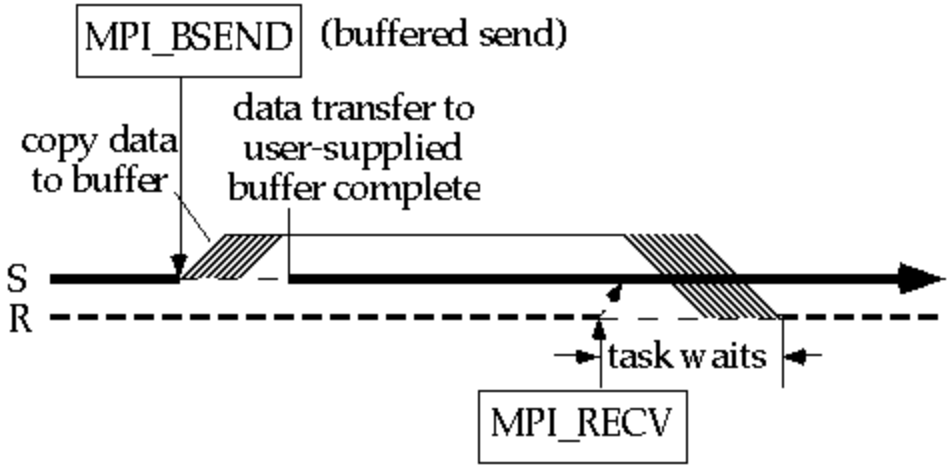


https://www.cac.cornell.edu/VW/MPIP2P/default.aspx

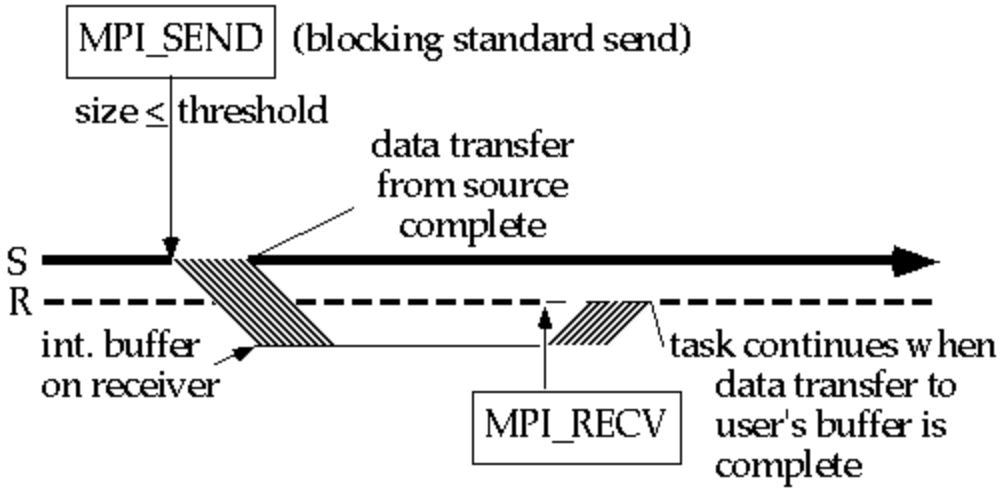
Blocking Buffered Send (MPI_BSEND)



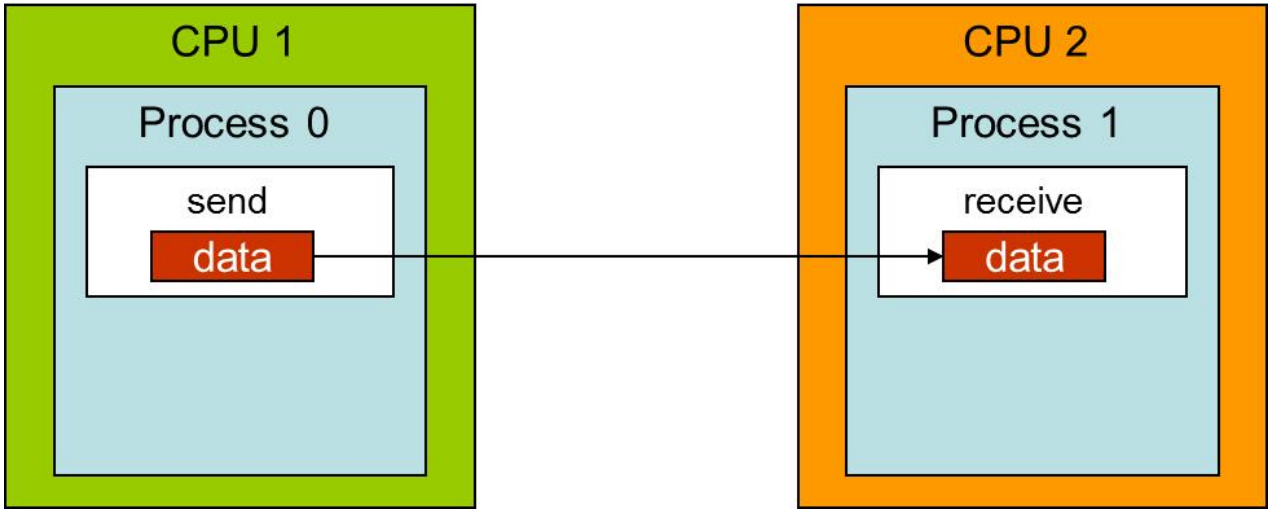
MPI_BUFFER_ATTACH
MPI_BUFFER_DETACH

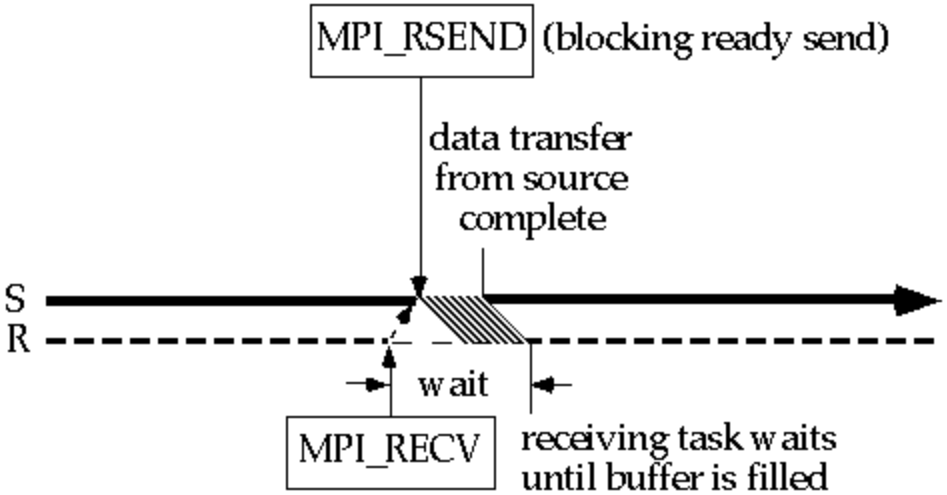


Blocking Standard Send (MPI_SEND)



Blocking Ready Send (MPI_RSEND)





Blocking calls: Developer point of view

The process that calls one of the routines: MPI_SSEND, MPI_BSEND, MPI_RSEND or MPI_SEND to send data, does not return from the routine until the variable that hosts the data can be reused by the developer without affecting them.

The process that calls the MPI_RECV routine to receive data does not return from the routine until the variable that will host the data is ready to use.

Non-blocking calls: Developer point of view

The process that calls one of the routines: MPI_ISSEND, MPI_IBSEND, MPI_IRSEND or MPI_ISEND to send data, simply notifies the MPI of an outgoing message (the actual transfer can take place later) and returns immediately. It is up to the developer to keep the variable, with the sending data, intact.
The developer must check (MPI_WAIT or MPI_TEST) at a later point in the program whether to reuse the variable without affecting the sending data.

The process that calls the MPI_IRECV routine to receive data, simply notifies MPI that it is ready for an incoming message and returns immediately. It is up to the developer to check (MPI_WAIT or MPI_TEST) at a later point in the program (MPI_WAIT or MPI_TEST) whether the variable that will host the data is ready to use.

Point-to-Point communication: Arguments

Blocking *send*	buffer, count, type, dest, tag, comm, error
non-Blocking *send*	buffer, count, type, dest, tag, comm, request, error
Blocking *receive*	buffer, count, type, source, tag, comm, status, error
non-Blocking *receive*	buffer, count, type, source, tag, comm, request, error

Example 5: Blocking message passing (1/2)

```
program ping
implicit NONE
include 'mpif.h'

integer numprocs, myid, error
integer dest, source, tag
integer status(MPI_STATUS_SIZE)
integer, parameter :: count=17
character(LEN=count) send_msg , recv_msg

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)

tag=1

if (myid==0) send_msg='Message from ID 0'
if (myid==1) send_msg='Message from ID 1'
```

Example 5: Blocking message passing (2/2)

```
if (myid==0) then
  dest = 1
  source = 1

  call MPI_SSEND(send_msg, count, MPI_CHARACTER, dest,  tag, &
                 MPI_COMM_WORLD, error)

  call MPI_RECV(recv_msg, count, MPI_CHARACTER, source, tag, &
                 MPI_COMM_WORLD, status, error)
endif

if (myid==1) then
  dest = 0
  source = 0

  call MPI_RECV(recv_msg, count, MPI_CHARACTER, source, tag, &
                 MPI_COMM_WORLD, status, error)

  call MPI_SSEND(send_msg, count, MPI_CHARACTER, dest,  tag, &
                  MPI_COMM_WORLD, error)
endif

print *, myid,'recv_mesg: ',recv_msg

call MPI_FINALIZE(error)

end
```

Example 6: non - Blocking message passing (1/2)

```
program ring
implicit NONE
include 'mpif.h'

integer myid, numprocs, error
integer next, prev
integer, dimension(2) :: buf
integer tag1, tag2
integer statuses(MPI_STATUS_SIZE,4), requests(4)

tag1 = 1
tag2 = 2

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)

prev = myid - 1
next = myid + 1

if (myid==0) prev = numprocs - 1

if (myid==(numprocs - 1)) next = 0
```

Example 6: non - Blocking message passing (2/2)

```
call MPI_IRECV(buf(1), 1, MPI_INTEGER, prev, tag1, &
               MPI_COMM_WORLD, requests(1), error)

call MPI_IRECV(buf(2), 1, MPI_INTEGER, next, tag2, &
               MPI_COMM_WORLD, requests(2), error)



call MPI_ISEND(myid, 1, MPI_INTEGER, prev, tag2,   &
               MPI_COMM_WORLD, requests(3), error)

call MPI_ISEND(myid, 1, MPI_INTEGER, next, tag1,   &
               MPI_COMM_WORLD, requests(4), error)

!         do some work

call MPI_WAITALL(4, requests, statuses, error)

print *, buf(1), myid, buf(2)

call MPI_FINALIZE(error)

end
```

Exercise: Parallel Dot Product

Let $x = [\ x_1\ x_2\ x_3\ \ldots\ x_N\ ]$ be a vector in $\mathbf{R}^N$ and $x_i = i\ /\ (i + 1)$

Write a MPI program to compute the dot product

$$x^T x = x_1^2 + x_2^2 + \ldots + x_N^2$$

on P processes. It is not assumed N is divisible by P

- Distribute x on P processes
- Calculate local dot products
- Global sum of local dot products

Solution

```
program dot_product
implicit NONE
include 'mpif.h'
integer, parameter :: N = 10000
integer myid, numprocs, error, i
integer N_local, temp
real(8), allocatable, dimension(:) :: x
real(8) sum_local, sum_global, shift

call MPI_INIT(error)

call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)
```

```
N_local = N / numprocs
temp=0
if(myid==(numprocs-1)) then
  temp = N - N_local * numprocs
  N_local = N_local + temp
endif
```

Find the local dimension (N_local) of vector x

```
shift = myid*(N_local - temp)
allocate(x(N_local))
do i=1,N_local
  x(i) = (shift+i)/(shift+i+1)
end do
```

Distribution of vector x on processes

```
sum_local=0.
do i=1,N_local
  sum_local=sum_local + x(i)**2
end do
```

Calculate the local dot products (sum_local)

```
call MPI_ALLREDUCE(sum_local, sum_global, 1, MPI_DOUBLE_PRECISION, &
                   MPI_SUM, MPI_COMM_WORLD, error)

print *, 'Dot product=', sum_global

call MPI_FINALIZE(error)

end
```

Global sum of local dot products

https://hpc-workshop.chemeng.ntua.gr

HPC Workshop

NATIONAL TECHNICAL UNIVERSITY OF ATHENS

EuroCC@Greece

Large-Scale Scientific Computations

School of Chemical Engineering

Computer Center

School of Mechanical Engineering

Parallel CFD and Optimization Unit

School of Electrical and Computer Engineering

Computing Systems Laboratory

This workshop is dedicated to students with some programming experience to learn the three parallel programming models **MPI**, **CUDA** and **OpenMP**. It starts on beginners level but also includes some advanced features like the parallelization of a Krylov type solver. **Hands-on sessions** (in C and Fortran) allow students to immediately test and understand the main building blocks of these parallel programming models.

Format: On-Site, Hands-on
When: Every year in July
Where: NTUA School of Chemical Engineering (PC-Lab)