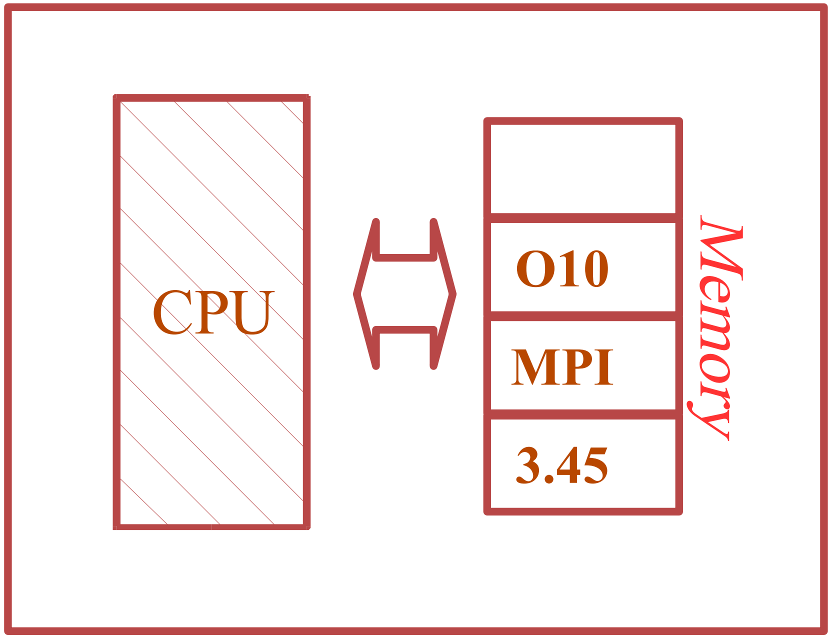
MPI

Massage Passing Interface

HPC08

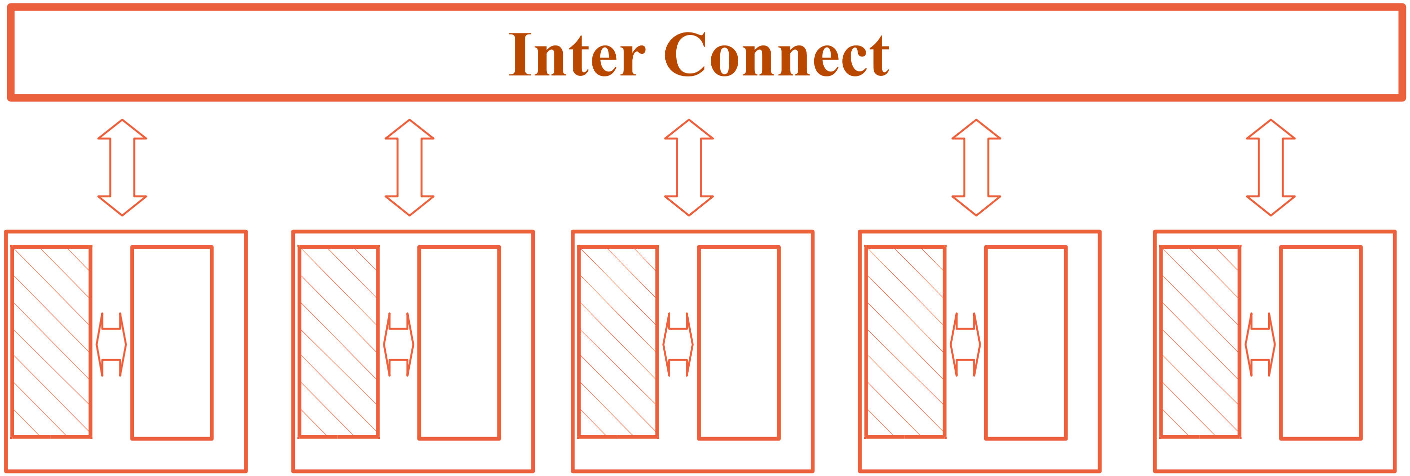
By: Ehsan Nedaaee Oskoee

The von Neumann computer



The Multicomputer

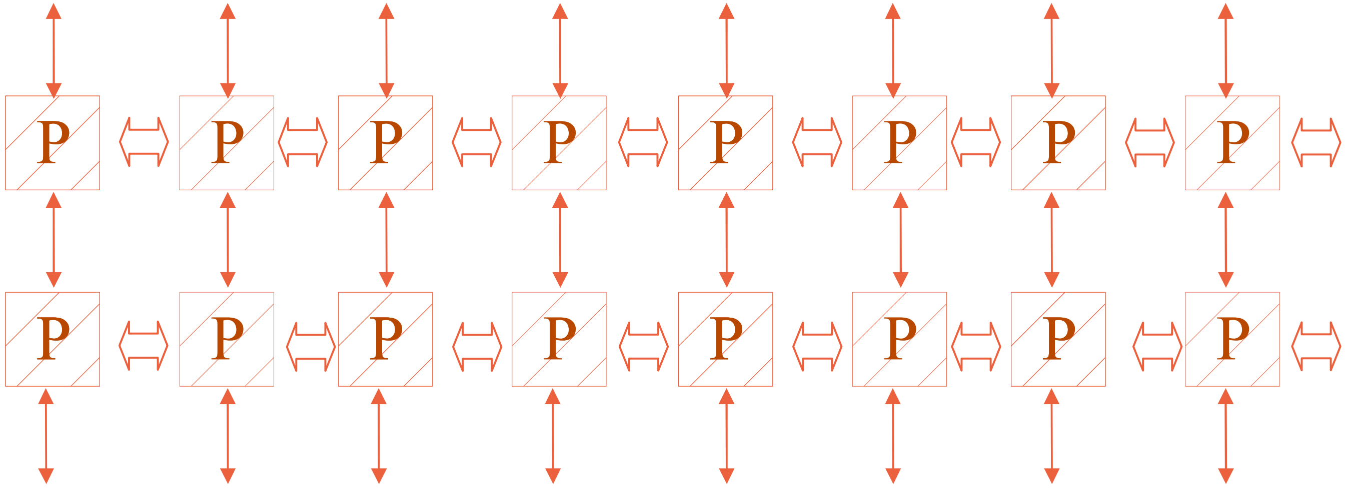
an idealized parallel computer model



Other Machine Models

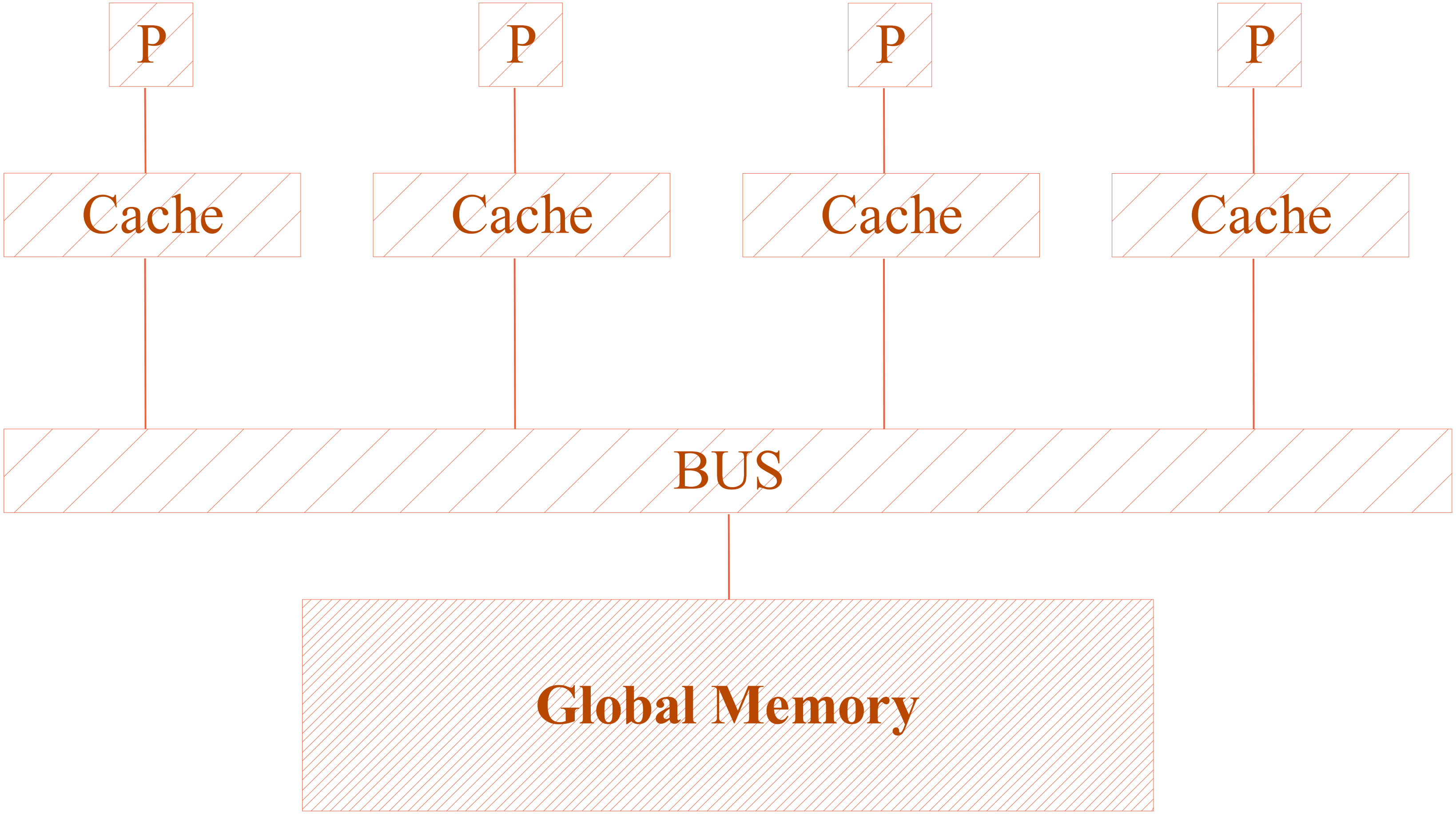
A Distributed-Memory **MIMD** *computer*

MIMD: M*ultiple Instruction Multiple Data*



also **SIMD: *Single Instruction Multiple Data***

Shared-Memory Multiprocessor



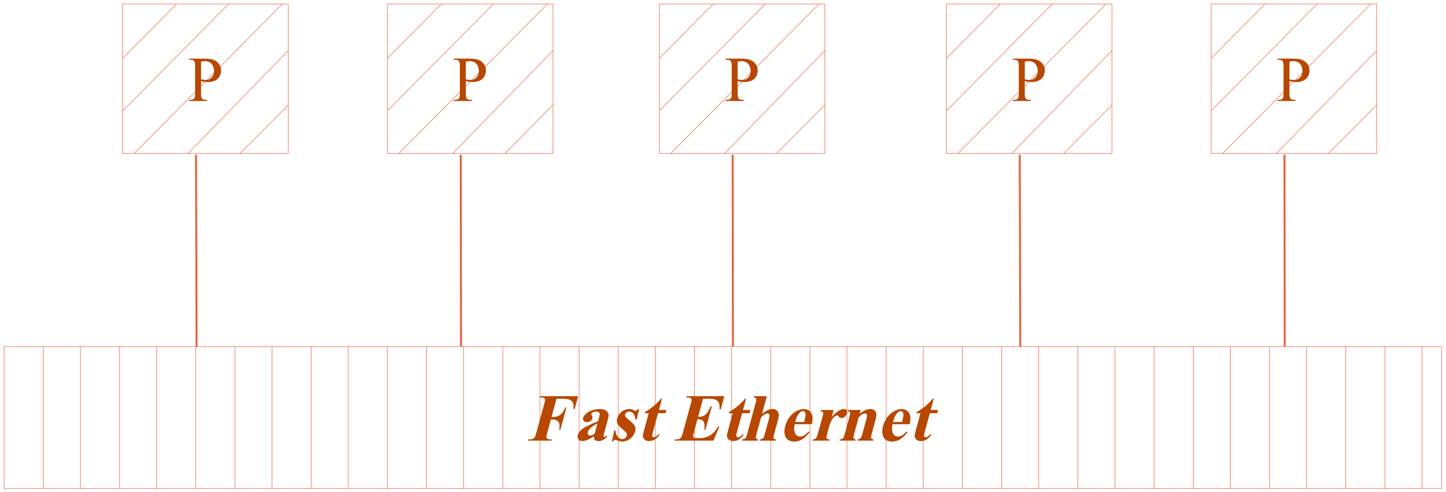
P P P P

Cache Cache Cache Cache

BUS

Global Memory

Local Area Network (LAN)

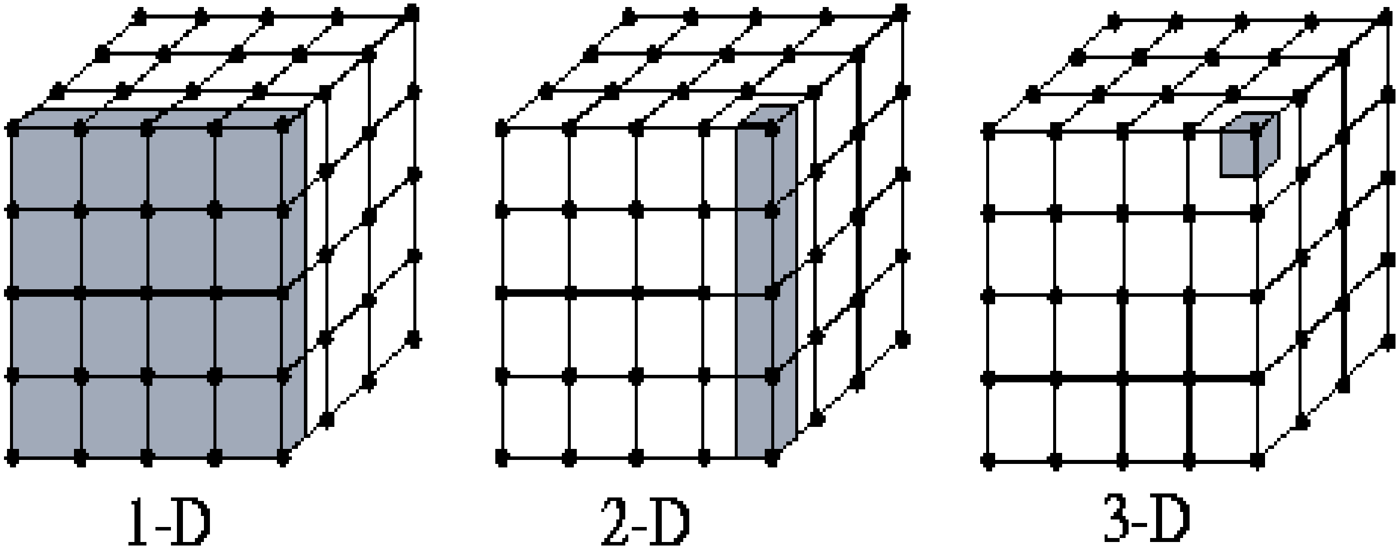


Wide Area Network (WAN)

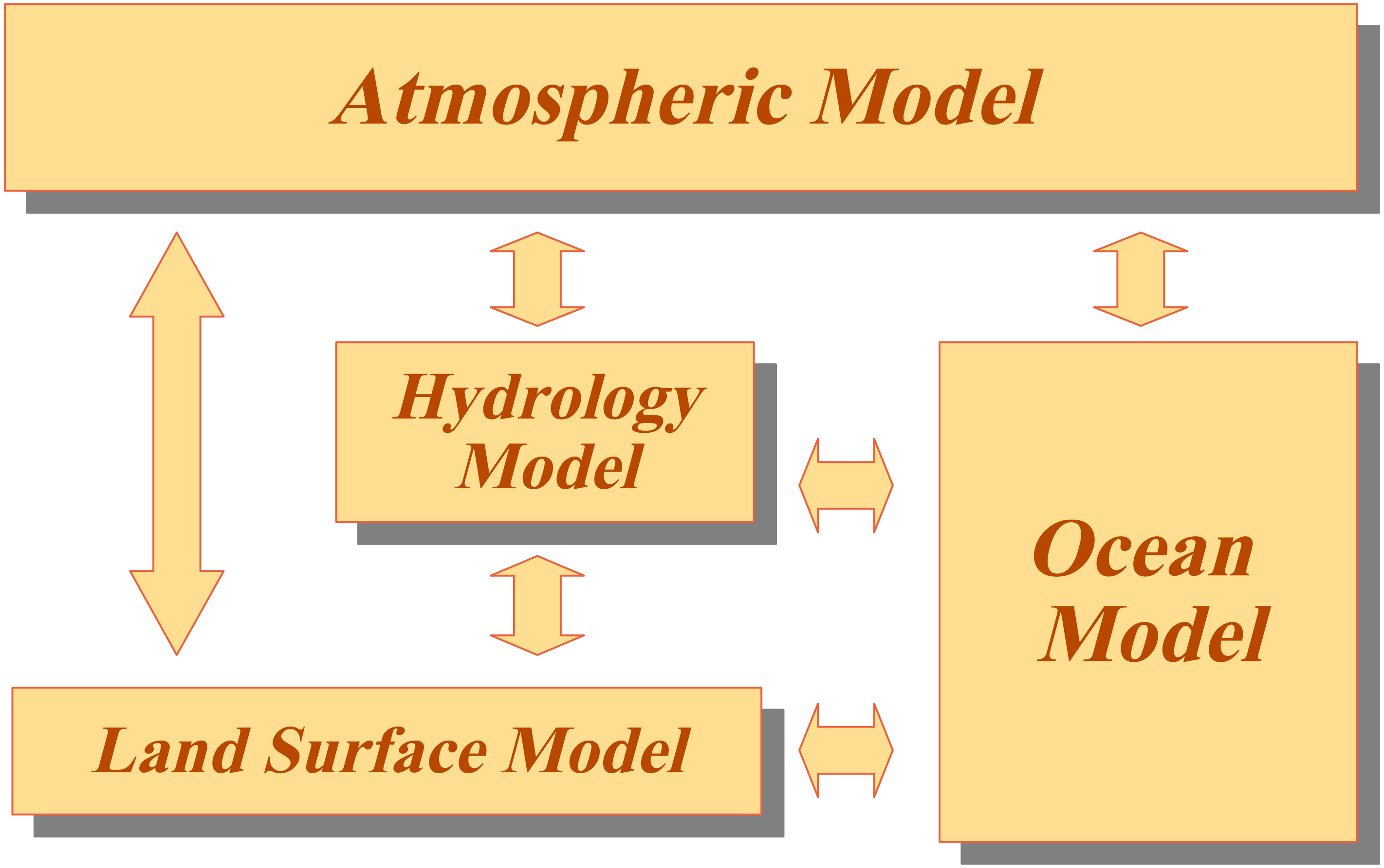
Methodical Design

Partitioning

Domain Decomposition



Functional Decomposition



Atmospheric Model

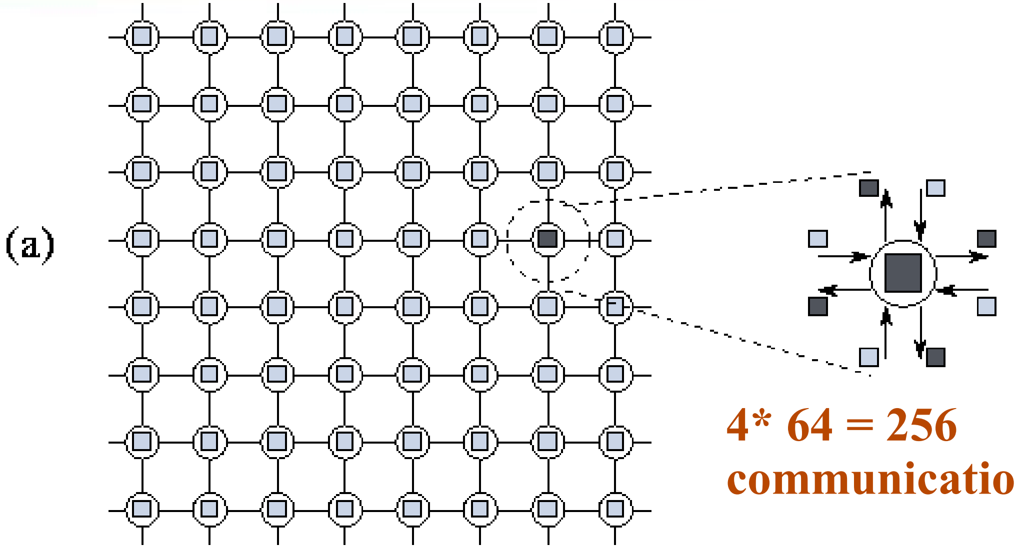
Hydrology Model

Ocean Model

Land Surface Model

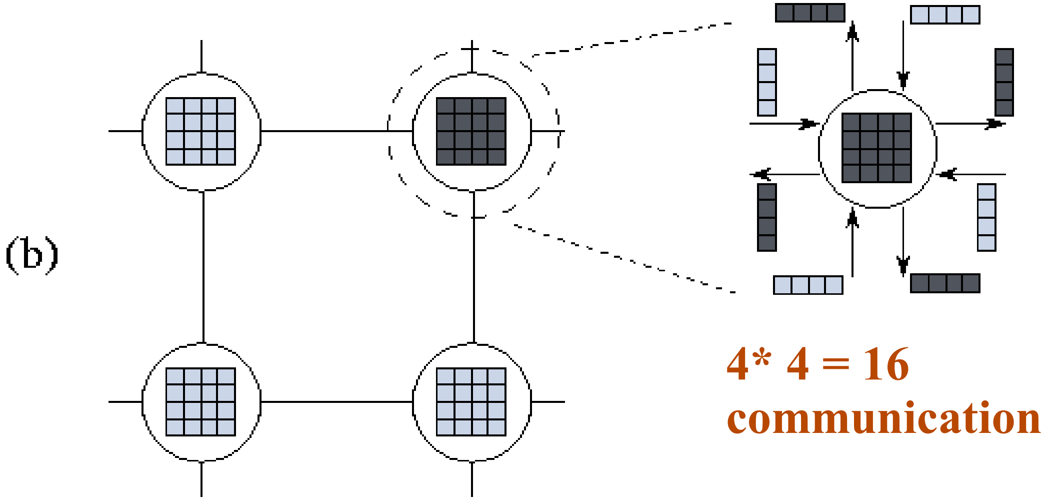
Communication

Agglomeration



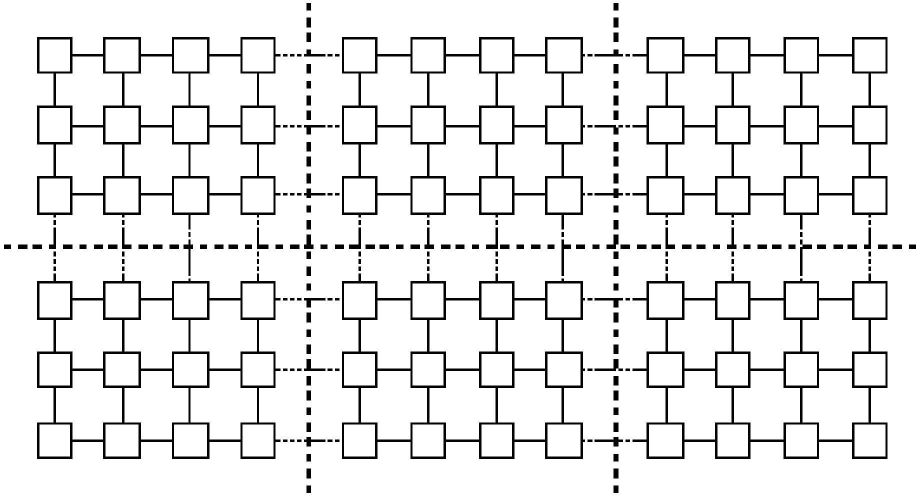
8*8 = 64 task

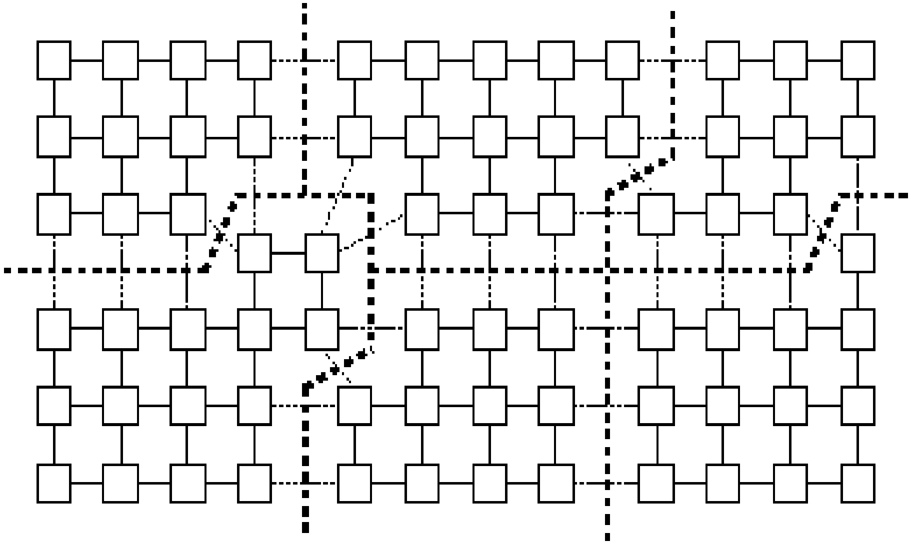
4* 64 = 256
communication



4* 4 = 16
communication

Mapping





MPI An Introduction

Applications:

Scalable Parallel Computers (SPCs) with distributed memory

Network Of Workstations (NOWs)

Some Goals of MPI:

Design an application programming interface

Allow efficient communication

Allow for implementations that can be used in a heterogeneous environment

Allow convenient C and Fortran 77 binding for the interface

Provide a reliable communication interface

Define an interface not too different from current practice, such as PVM, NX, etc.

What is Included in MPI

- Point-to-point communication
- Collective operations
- Process groups
- Communication domains
- Process topologies
- Environmental Management and inquiry
- Profiling interface
- Binding for Fortran 77 and C (Also for C++ and F90 in MPI-2)
- I/O functions (in MPI-2)

Versions of MPI

- Version 1.0 (was made in June 1994)
- Version 1.1 (was made in June 1995)
- Version 2

Procedure Specification

- The call uses but does not update an argument marked **IN**
- The call may update an argument marked **OUT**
- The call both uses and updates an argument marked **INOUT**

Types of MPI Calls

- Local
- Non-local

- Blocking
- Non-blocking

Opaque Objects

Language Binding

Point-to-point Communication

Blocking PTP Communication

The simplest program

main.for

```
Program main

  implicit none
  include 'mpif.h'
  integer ierr,rc
  call MPI_INIT( ierr )

  print*,' HI There'

  call MPI_FINALIZE(rc)
End
```

main.cpp

```
#include <iostream.h>
#include <mpi.h>

int main(int argc, char **argv)
{

  MPI_Init(&argc, &argv);

  cout<<"HI There\n";

  MPI_Finalize();
  return 0;
}
```

Compiling a Program

```
%more hostfile
192.168.189.197      1
Naft         1
Oil                       2
```

```
%more hostfile
HPCLAB
ws01
ws02
ws03
```

```
Lamboot –v hostfile

mpicc code_name.c –o code_exe_name

mpiCC code_name.cpp –o code_exe_name

mpif77 code_name.for –o code_exe_name

mpif90 code_name.f90 –o code_exe_name

mpirun –v –np 9 code_exe_name

mpirun N code_exe_name
```

A More Complex Program

```
#include <iostream.h>
#include <mpi.h>

int main(int argc, char **argv)
{
  int npes, myrank;
  MPI_Init(&argc, &argv);
```

Adding new functions to getting number of processors which is running the code and rank of the local processor, as below:

```
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  cout<<"HI There, I am node "<<myrank<<"  and the total worker"
      <<" which you are using now is: "<<npes<<'\n';

  MPI_Finalize();
  return 0;
}
```

A More Complex Program

```
Program size_rank

         implicit none
         include 'mpif.h'
         integer ierr,npes,myrank
         call MPI_INIT( ierr )
```

Adding new functions to getting number of processors which is running the code and rank of the local processor, as below:

```
         call MPI_COMM_SIZE( MPI_COMM_WORLD, npes, ierr )
         call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )

         print*,"HI There, I am node ",myrank," and the total",
     *"number of workers which you are using now is: ",npes

         call MPI_FINALIZE(ierr)
       End
```

Point to Point (PTP) *Communication*

Blocking Send Operation

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer
IN	count	number of entries to send
IN	datatype	datatype of each entry
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator

C version

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

Fortran version

```
MPI_SEND(BUF, COUNT, DATATYPE,  DEST, TAG, COMM, IERROR)

<type> BUF(*)

INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Blocking Receive Operation

MPI_RECV(buf, count, datatype, source, tag, comm, status)

OUT	buf	
IN	count	
IN	datatype	
IN	source	rank of source
IN	tag	
IN	comm	
OUT	status	return status

C version

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

Fortran version

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
<type> BUF(*)  INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
STATUS(MPI_STATUS_SIZE), IERROR
```

Data Types

MPI Data Type	Fortran Data Type
MPI_INTEGER	**INTEGER**
MPI_REAL	**REAL**
MPI_DOUBLE_PRECISION	**BOUBLE PRECISION**
MPI_COMPLEX	**COMPLEX**
MPI_LOGICAL	**LOGICAL**
MPI_CHARACTER	**CHARACTER(1)**
MPI_BYTE	
MPI_PACKED	

Data Types

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_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

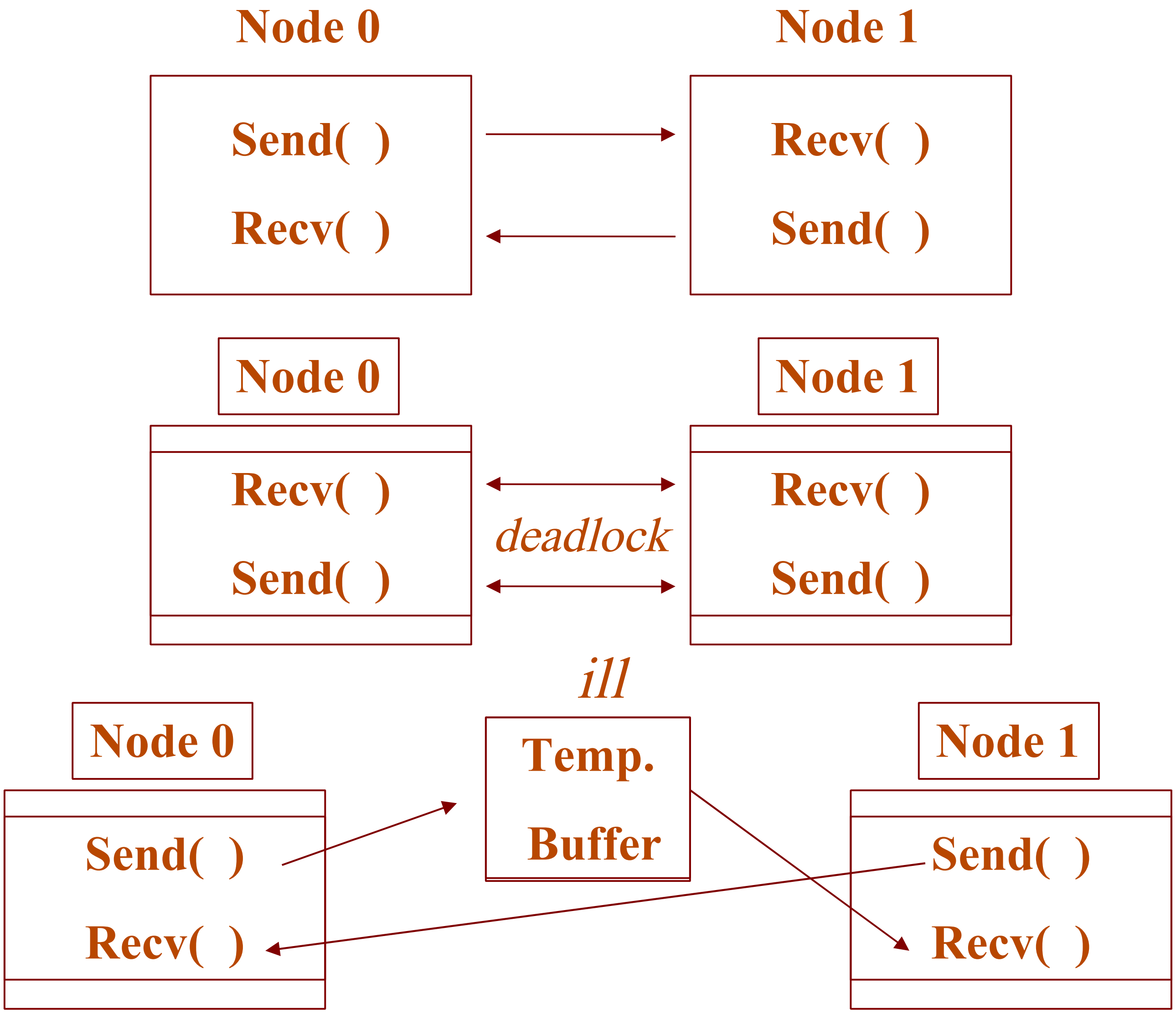
Point to Point (PTP) *Communication*

MPI_GET_PROCESSOR_NAME(name, resultlen)

OUT	**name**	**A unique specifier for the current physical node**
OUT	**resultlen**	**Length (in printable characters) of the result returned in name**

```
int MPI_Get_processor_name(char* name, int* resultlen)
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
CHARACTER*(*) NAME
INTEGER RESULTLEN, IERROR
```

Safety



Node 0

Send( )

Recv( )

Node 1

Recv( )

Send( )

Node 0

Recv( )

Send( )

deadlock

Node 1

Recv( )

Send( )

ill

Node 0

Send( )

Recv( )

Temp.

Buffer

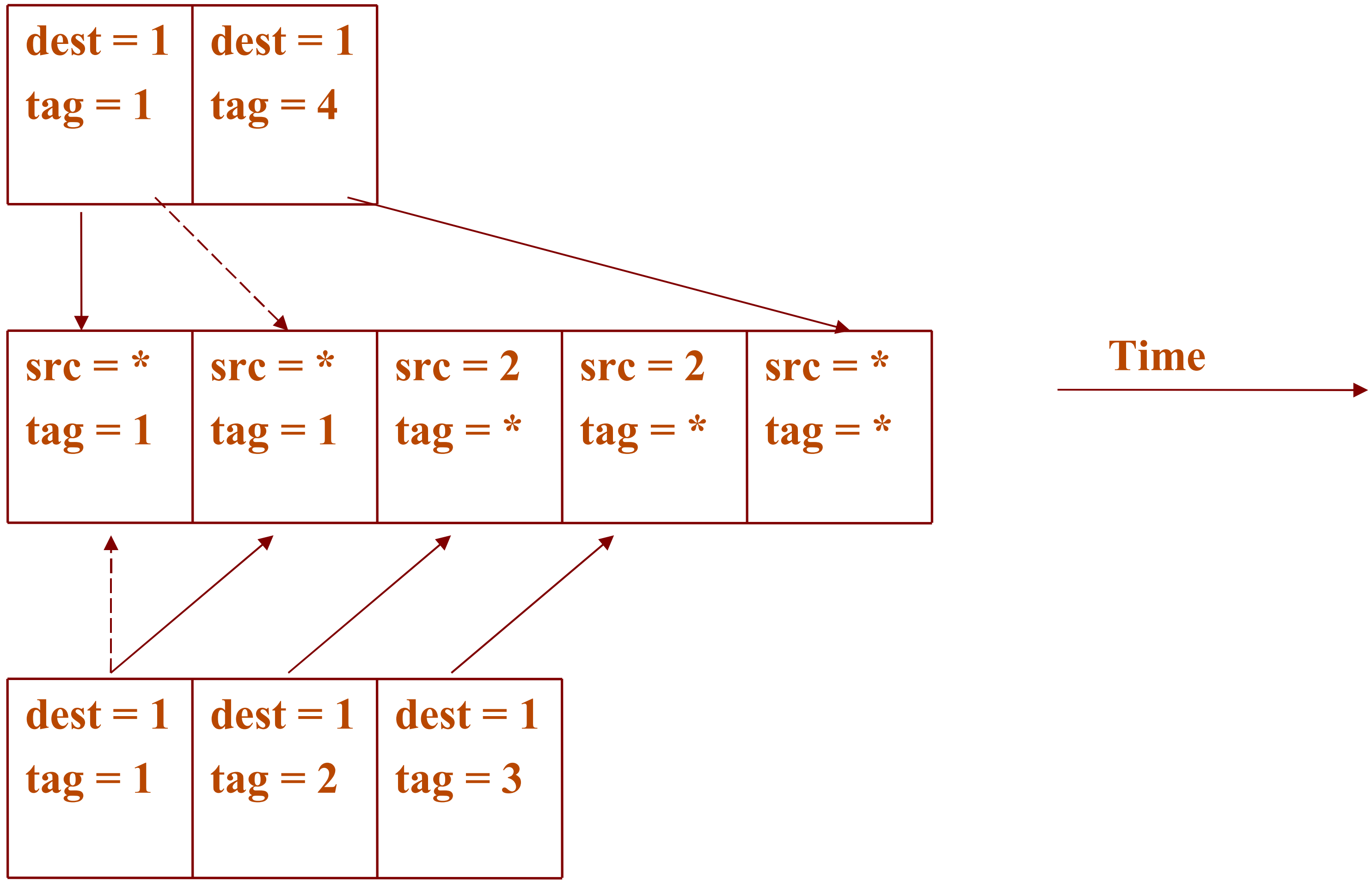
Node 1

Send( )

Recv( )

Point to Point (PTP) *Communication*

Order



Process 0

(Send)

dest = 1 tag = 1

dest = 1 tag = 4

Process 1

(recv)

src = * tag = 1

src = * tag = 1

src = 2 tag = *

src = 2 tag = *

src = * tag = *

Time

Process 2

(Send)

dest = 1 tag = 1

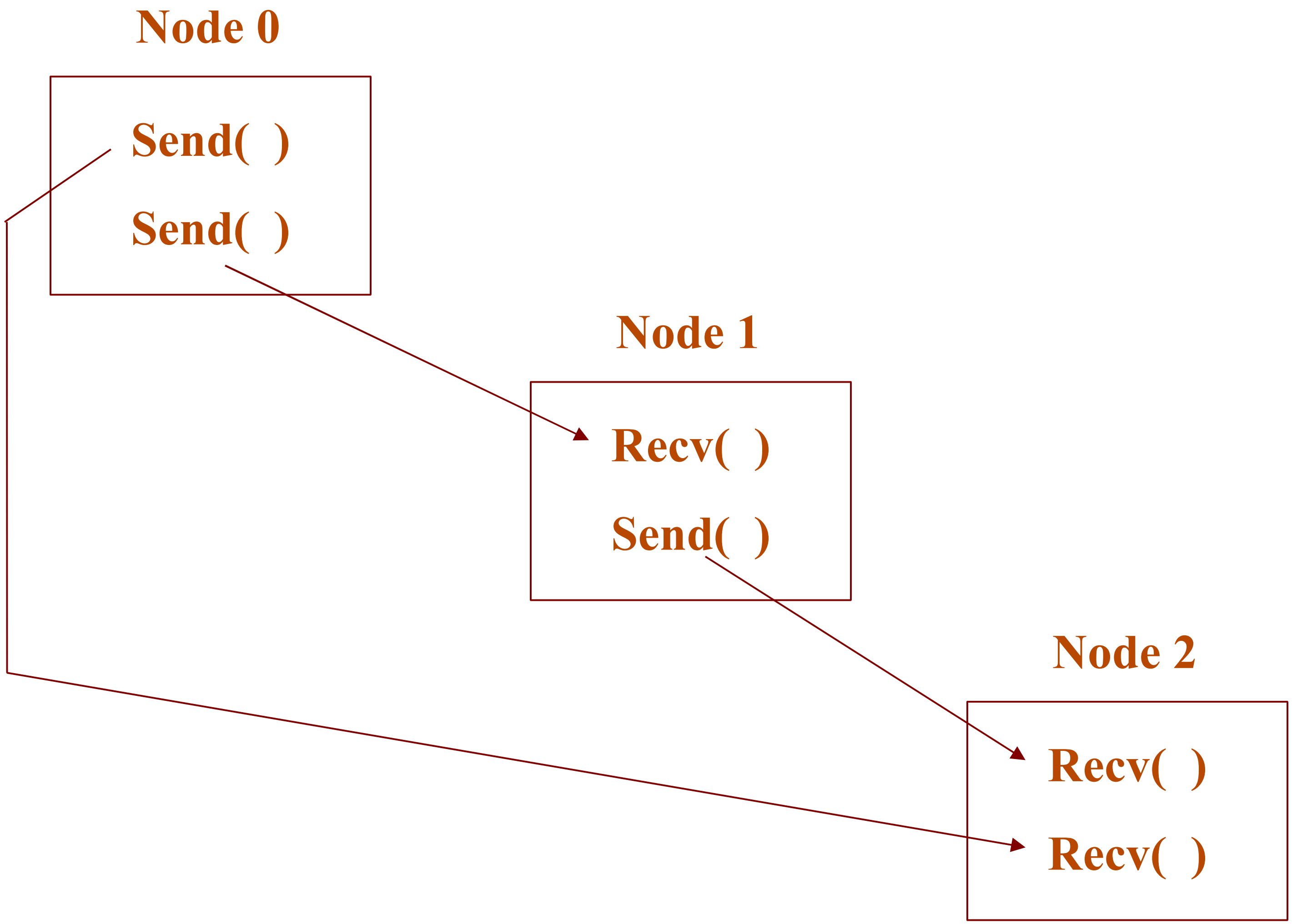
dest = 1 tag = 2

dest = 1 tag = 3

src = * :: MPI_ANY_SOURCE tag = * :: MPI_ANY_TAG

Point to Point (PTP) *Communication*

Order



Node 0

Send( )

Send( )

Node 1

Recv( )

Send( )

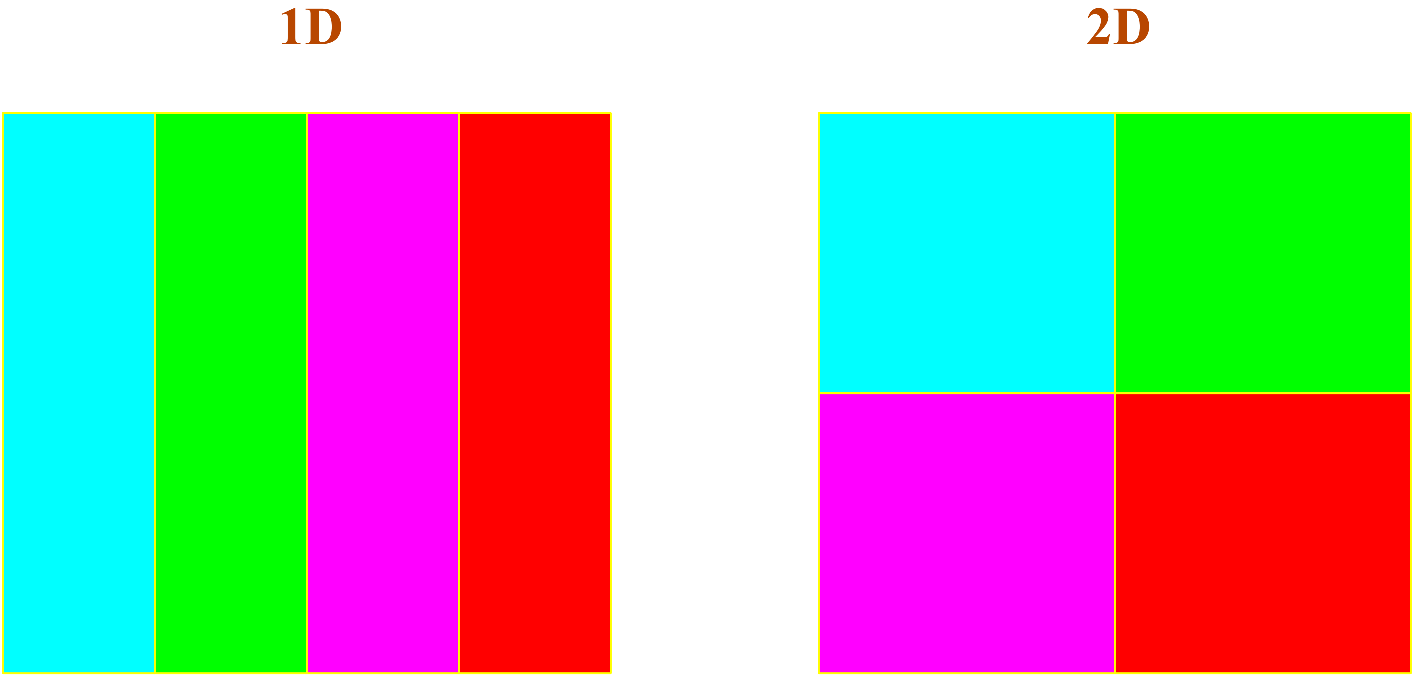
Node 2

Recv( )

Recv( )

Program Blocking Send & Recd 2

Domain Decomposition



Program Bsend_recd_2.cpp

```
for(j =0; j<= np+1; j++){
      for(i=0; i<= np+1; i++)
          a[i][j] = myrank*100+10*j+i;
  }
  left = myrank - 1;
  right= myrank + 1;
  if(myrank == 1 ) left = npes -1 ;
  if(myrank == (npes-1)) right = 1;
if (myrank != 0 ) {
      if(myrank%2==0){
       MPI_Send(&a[1][0],np+2,MPI_INT,left,1,MPI_COMM_WORLD);
       MPI_Send(&a[np][0],np+2,MPI_INT,right,1,MPI_COMM_WORLD);
    MPI_Recv(&a[0][0],np+2,MPI_INT,left,1,MPI_COMM_WORLD,&stat);
    MPI_Recv(&a[np+1][0],np+2,MPI_INT,right,1,MPI_COMM_WORLD,&stat);
      }else{
    MPI_Recv(&a[np+1][0],np+2,MPI_INT,right,1,MPI_COMM_WORLD,&stat);
    MPI_Recv(&a[0][0],np+2,MPI_INT,left,1,MPI_COMM_WORLD,&stat);
       MPI_Send(&a[np][0],np+2,MPI_INT,right,1,MPI_COMM_WORLD);
       MPI_Send(&a[1][0],np+2,MPI_INT,left,1,MPI_COMM_WORLD);
      }
}
```

Program Bsend_recd_2.for

```
    IF(myrank.NE.0) THEN
     IF(MOD(myrank,2).EQ.0)THEN
        call MPI_SEND(a(0,1),np+2,MPI_INTEGER,left,1,
*            MPI_COMM_WORLD,ierr)
        call MPI_SEND(a(0,np),np+2,MPI_INTEGER,right,1,
*            MPI_COMM_WORLD,ierr)
         call MPI_RECV(a(0,0),np+2,MPI_INTEGER,left,1,
*                   MPI_COMM_WORLD,stat,ierr)
         call MPI_RECV(a(0,np+1),np+2,MPI_INTEGER,right,1,
*                   MPI_COMM_WORLD,stat,ierr)
     ELSE
         call MPI_RECV(a(0,np+1),np+2,MPI_INTEGER,right,1,
*                   MPI_COMM_WORLD,stat,ierr)
         call MPI_RECV(a(0,0),np+2,MPI_INTEGER,left,1,
*                   MPI_COMM_WORLD,stat,ierr)
        call MPI_SEND(a(0,np),np+2,MPI_INTEGER,right,1,
*            MPI_COMM_WORLD,ierr)
        call MPI_SEND(a(0,1),np+2,MPI_INTEGER,left,1,
*            MPI_COMM_WORLD,ierr)
     ENDIF
    END IF
```

Blocking Send Receive Operation

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvount, recvtype, source, recvtag, comm, status)

IN	**sendbuf**
IN	**sendcount**
IN	**sendtype**
IN	**dest**
IN	**sendtag**
OUT	**recvcount**
IN	**recvtype**
IN	**source**
IN	**recvtag**
IN	**comm**
OUT	**status**

Point to Point (PTP) *Communication*

C version

```
int  MPI_Sendrecv(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)
```

Fortran version

```
MPI_RECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG,  RECVBUF
RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
<type> SENDBUF(*)  RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT,
RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

Blocking Send Receive Replace Operation

MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)

INOUT	**buf**
IN	**count**
IN	**datatype**
IN	**dest**
IN	**sendtag**
IN	**source**
IN	**recvtag**
IN	**comm**
OUT	**status**

Point to Point (PTP) *Communication*

C version

```
int  MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype, int dest,
 int sendtag,  int source, Int recvtag,  MPI_Comm comm,  MPI_Status *status)
```

Fortran version

```
MPI_RECV(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE,  RECVTAG,
           COMM, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, SENDTAG,  SOURCE, RECVTAG, COMM,
STATUS(MPI_STATUS_SIZE), IERROR
```

Non Blocking PTP Communication

Non Blocking Send Operation

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

IN	**buf**	**initial address of send buffer**
IN	**count**	**number of entries to send**
IN	**datatype**	**datatype of each entry**
IN	**dest**	**rank of destination**
IN	**tag**	**message tag**
IN	**comm**	**communicator**
OUT	**request**	

Point to Point (PTP) *Communication*

C version

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

Fortran version

MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)

<type> BUF(*)

INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

Non Blocking Receive Operation

MPI_IRECV(buf, count, datatype, source, tag, comm, request)

OUT	**buf**	
IN	**count**	
IN	**datatype**	
IN	**source**	**rank of source**
IN	**tag**	
IN	**comm**	
OUT	**request**	**request handle**

Point to Point (PTP) *Communication*

C version

```
int  MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int
tag,

 MPI_Comm comm,  MPI_Request *request)
```

Fortran version

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,

 IERROR)

<type> BUF(*)  INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,

 REQUEST, IERROR
```

Completion Operations

MPI_WAIT(request, status)

INOUT	**request**
OUT	**status**

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
MPI_WAIT(REQUEST, STATUS, IERROR)
 INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

Point to Point (PTP) *Communication*

MPI_TEST(request,flag,status)

INOUT	**request**
OUT	**flag**
OUT	**status**

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

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)

LOGICAL FLAG

INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

Thank You