MPI

ECSE 420 - Tutorial 5

Dimitrios Stamoulis

TR 4110

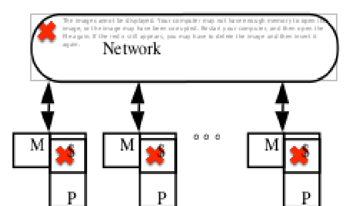
October 22, 2014

Lab 3 – Introduction

From ECSE 420 lecture slides!!

Message Passing Architectures

- Complete computer as building block, including I/O
  - Communication via explicit I/O operations
- Programming model
  - direct access only to private address space (local memory),
  - communication via explicit messages (send/receive)
- High-level block diagram
  - Communication integration?
    - Mem, I/O, LAN, Cluster
  - Easier to build and scale than SAS
- Programming model more removed from basic hardware operations
  - Library or OS intervention



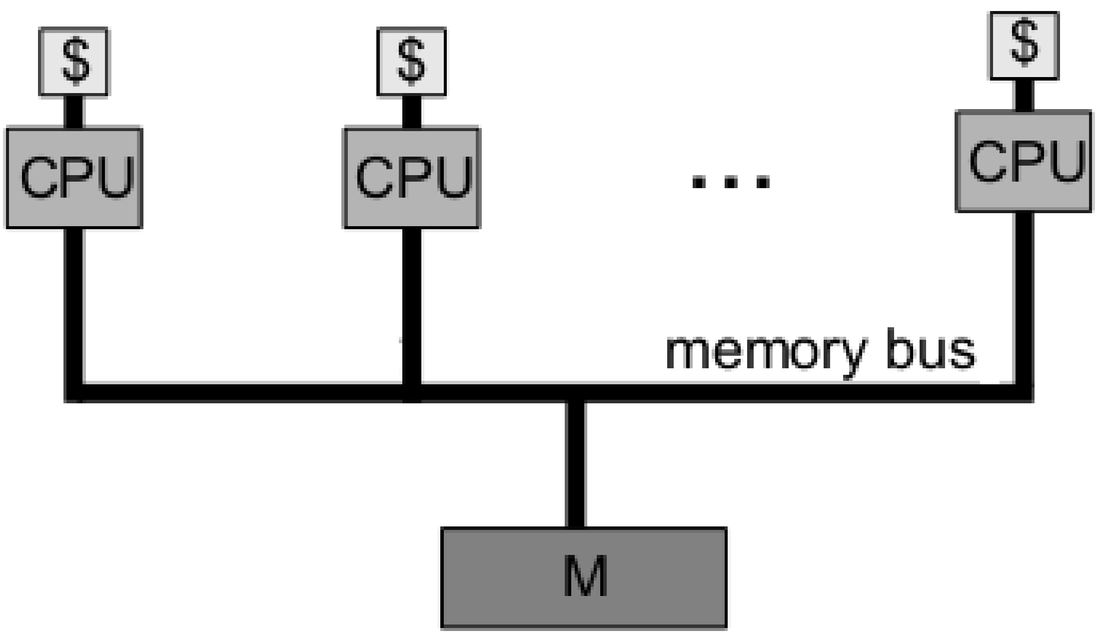
Software

Boundary

Hardware

During Lab 2...

We discussed about
- Shared memory architectures
- Shared address space programming model (OpenMP)

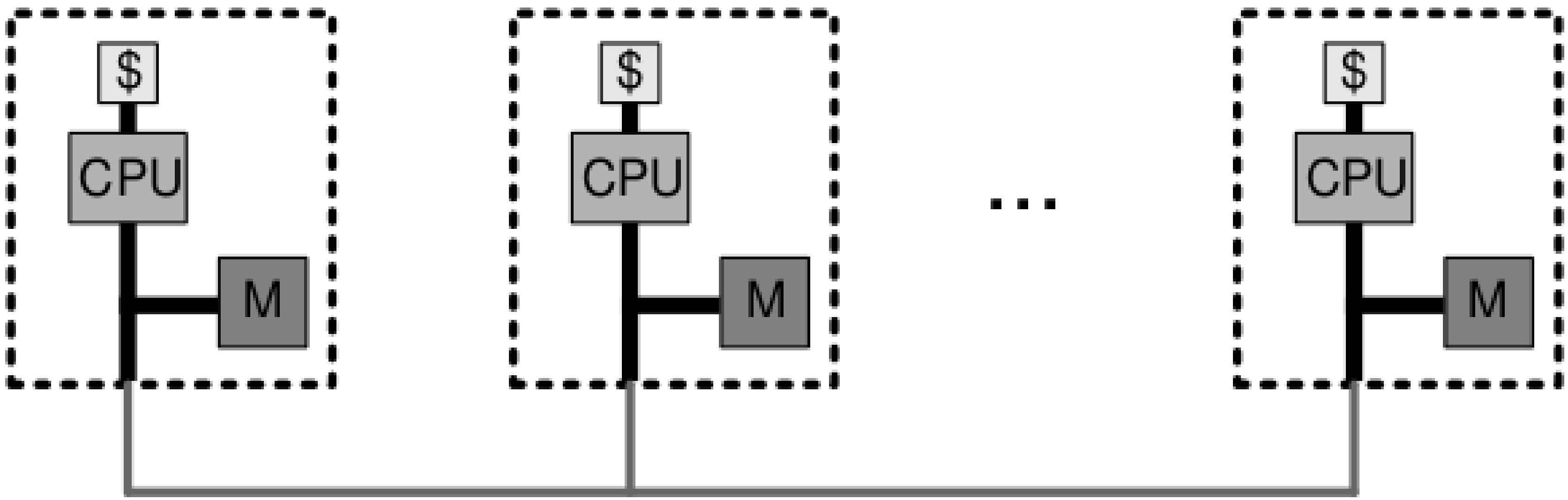


++ popular architectural approach

++ simple programming model

++ efficient parallel programming

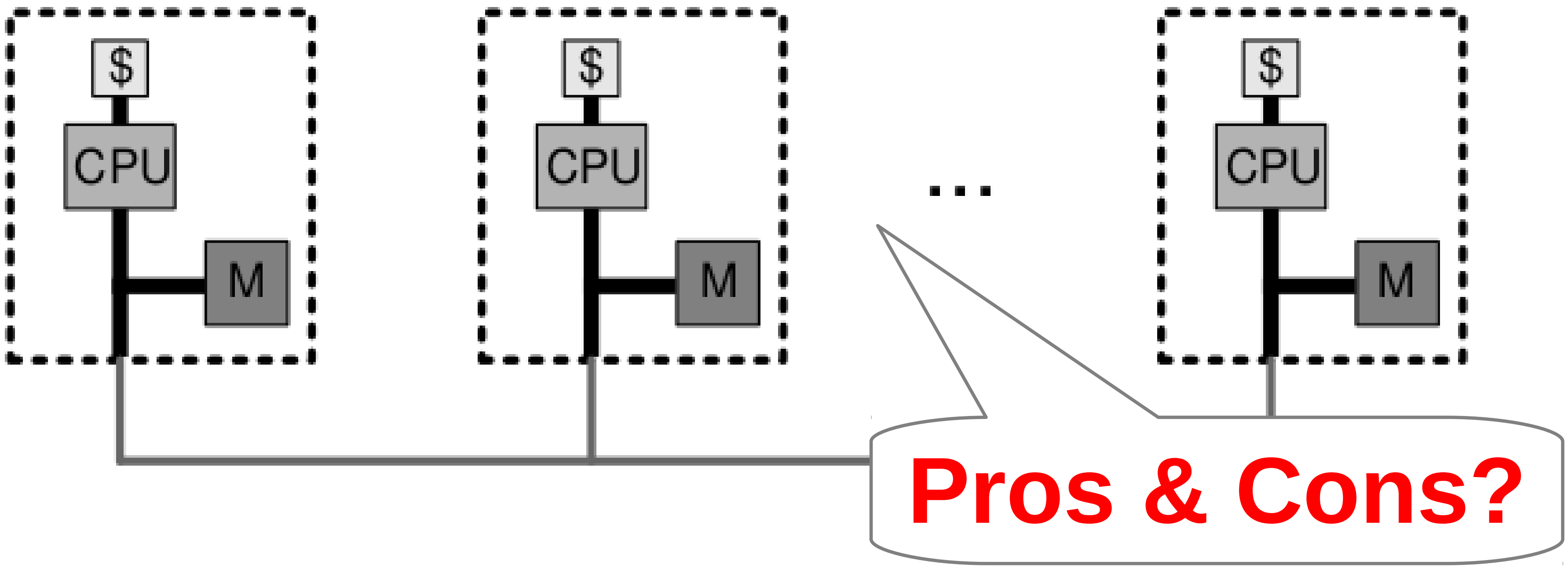
Message Passing Architectures



Programming model : we have to change it properly

→More removed from basic hardware operations

Message Passing Architectures

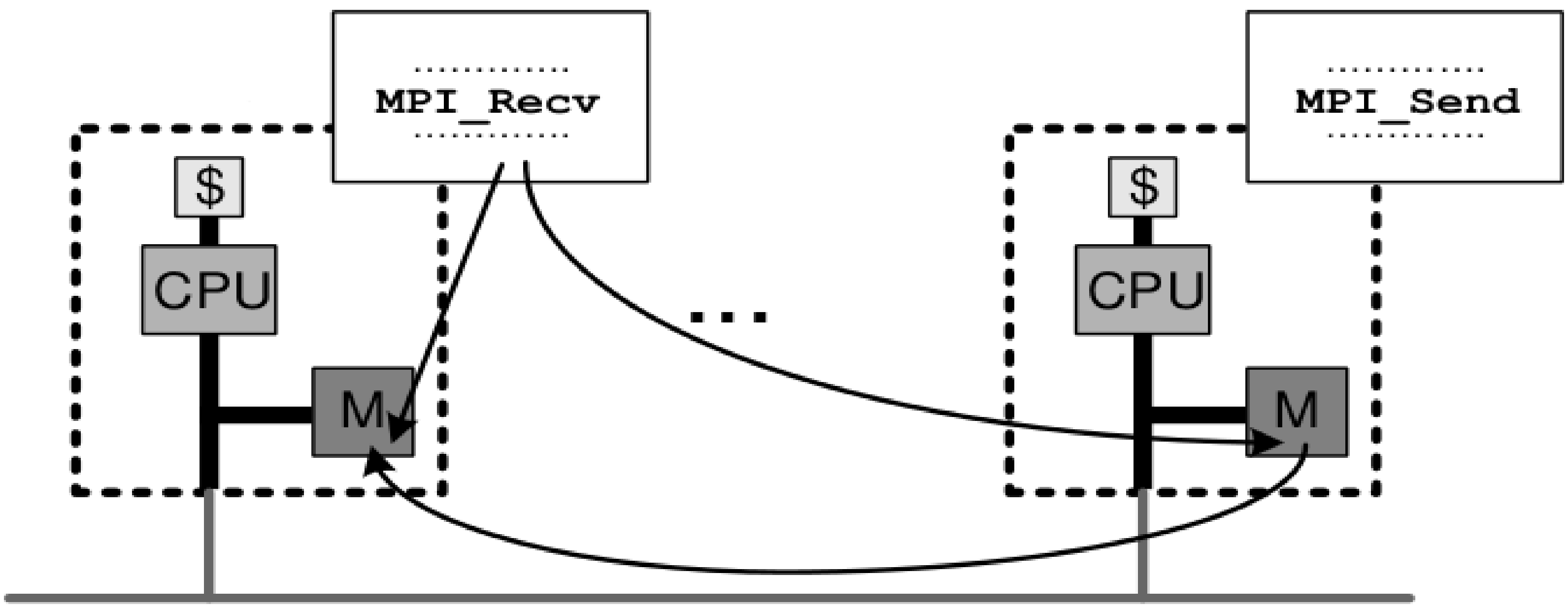


Programming model : we have to change it properly

→More removed from basic hardware operations

Send & receive operations (1/2)

We have:
→ send(void *sendbuf, int nelems, int dest)
→ receive(void *recvbuf, int nelems, int source)



Send & receive operations (2/2)

We have:

→ send(void *sendbuf, int nelems, int dest)

→ receive(void *recvbuf, int nelems, int source)

P0 sends data to P1

```
P0                    P1
a = 100;              receive(&a, 1, 0)
send(&a, 1, 1);       printf("%d\n", a);
a = b;
```

Good semantics → P1 to receives 100

Bad semantics → P1 receives b

Send & receive operations (2/2)

We have:

→ send(void *sendbuf, int nelems, int dest)

→ receive(void *recvbuf, int nelems, int source)

P0 sends data to P1

```
P0                     P1
a = 100;               receive(&a, 1, 0)
send(&a, 1, 1);        printf("%d\n", a);
a = b;
```

Good semantics → P1 to receives 100

Bad semantics → P1 receives b

HOW?

Programmer's challenges - Deadlock

We assume blocking, non-buffered send/receive:

```
P0
send(&b, 1, 1);
receive(&a, 1, 1);
```

```
P1
send(&b, 1, 0);
receive(&a, 1, 0);
```

Both sends wait for both receives :

DEADLOCK

WHY?

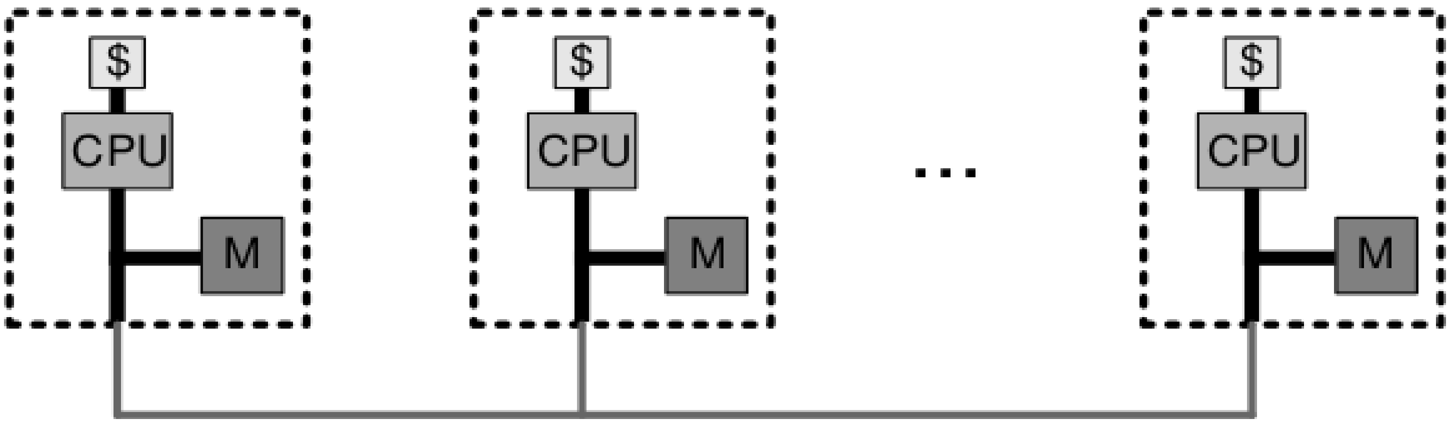
Programmer's challenges

- Structure of message passing programs :
  SPMD (single program, multiple data) model
  - All processes execute the same code
  - The process' execution flow differs based on its rank

- Programmer's challenges :
  - Maximum parallelization
  - Efficient usage of HW resources (e.g memory)
  - Minimum data to be transfered
  - Minimum number of messages
  - Minimum synchronization effort

Message Passing Programming Model

Programming model : we have to change it properly
→More removed from basic hardware operations



- Message Passing Libraries
  - Vendors all had their own message passing libraries
- MPI :
  - Defines syntax, semantics of core set of library routines

Core set of routines for MPI

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

Starting and Terminating MPI

```
int MPI_Init(int *argc, char ***argv)
```

- MPI_Init is called prior to other MPI routines-it initializes the MPI environment

```
int MPI_Finalize()
```

- MPI_Finalize is called at the end-it does clean-up

Return code for both is MPI_success

Communicators

Communicators :

- Are variables of type MPI_Comm.
- Define a set of processes that can communicate with each other

MPI_COMM_WORLD

- default communicator, all processes in program

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- MPI_Comm_size - number of processes in communicator

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Rank id's each process

Typical structure of MPI code

```
#include <mpi.h>

main(int argc, char *argv[])
{
   ...
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   ...
   MPI_Finalize();
}
```

Our first example

```
#include <mpi.h>

main(int argc, char *argv[])
{
  int npes, myrank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  printf("I'm process %d out of %d\n",myrank, npes);
  MPI_Finalize();
}
```

- Compilation: $ mpicc.mpich ex1.c -o ex1
- Execution: $ mpiexec.mpich -np 2 ./ex1

MPI_Send()

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

→ MPI_Send sends data in `buf`

→ `dest` = rank of destination process

→ Length of message = number of entries of type `MPI_Datatype`

→ `tag` = type of message (e.g. a number)

→ `count` & `datatype` specify length of buffer

Example :

```
int message[20],dest=1,tag=55;
MPI_Send(message, 20, MPI_INT, dest, tag,
         MPI_COMM_WORLD);
```

MPI_Recv()

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

→`buf` is where received message is stored

→Message to be received from `source` process

→`status` variable used to get info on `MPI_Recv` status

Example :

```
int message[50],source=0,tag=55;
MPI_Status status;
MPI_Recv(message, 50, MPI_INT, source, tag,
         MPI_COMM_WORLD,&status);
```

Datatypes

MPI Datatype	C Datatype
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	

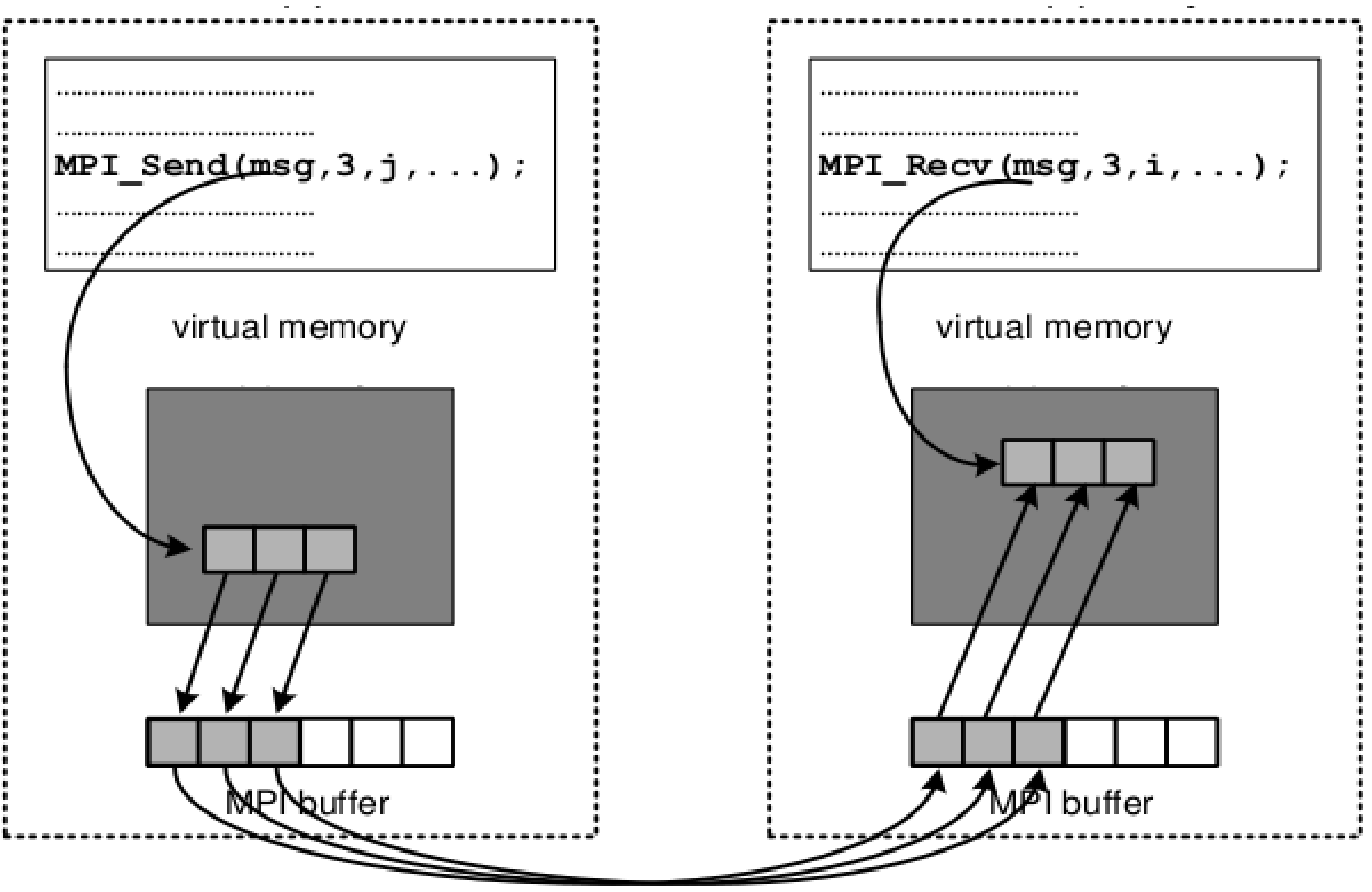
Let's see a simple example

```
//Compute f(0) and f(1) in parallel and find their sum
#include <mpi.h>

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

 int v0,v1,sum,rank;
 MPI_Status status;
 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 if (rank == 1) {
  v1 = f(1);
  MPI_Send(&v1,1,MPI_INT,0,50,MPI_COMM_WORLD);
 }
 else if (rank == 0) {
  v0=f(0);
  MPI_Recv(&v1,1,MPI_INT, 1,50,MPI_COMM_WORLD,&status);
  sum=v0+v1;
  printf("f(0)+f(1) = %d + %d = %d\n",v0,v1,sum);
 }
  MPI_Finalize();
}
```

Communication (1/2)



Communication (2/2)

We can have different types of communication :

→Synchronous (MPI_Ssend) vs Buffered (MPI_Bsend)

What I consider to be `MPI_Send` 's successful completion?

→Blocking vs Non-blocking (MPI_Isend)

Shall I wait for `MPI_Recv` to be completed?

→Point-to-Point vs Collective

Can I send a message to everyone else?

Sending/Receiving Messages

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

→A blocking operation : Returns only after message is in buffer..!!

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

→Synchronous : Returns after `MPI_Recv` issued and message is sent

→Buffering : Returns after `MPI_Send` copied message into buffer

→ Does **NOT** wait for `MPI_Recv` to be issued

Deadlocks

```
int a[10], b[10], rank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (rank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
```

Deadlocks

```
int a[10], b[10], rank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (r… == 1) {
    MPI_Rec…  10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Rec…  …0, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
```

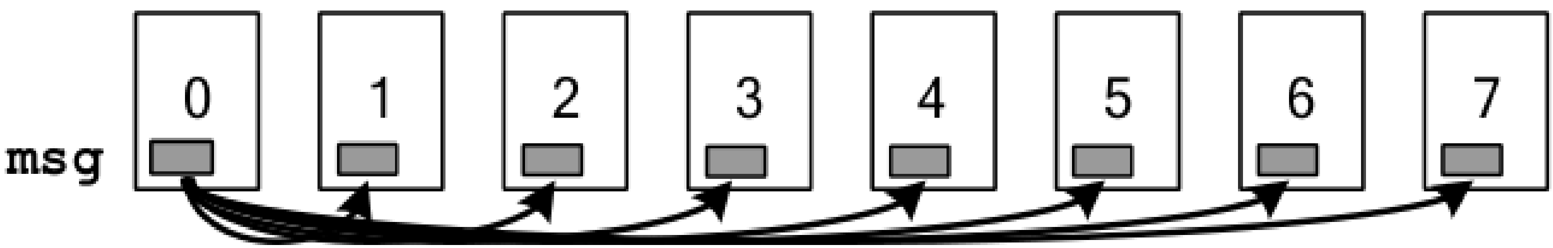
With a synchronous Send, we have a deadlock e.g let's try MPI_Ssend()

MPI_Bcast - Collective Comm (1/2)

We could do something like :

```
...
if(rank==0){
 for(dest=1;dest<size;dest++)
 MPI_Send(msg,count,dest,tag,MPI_FLOAT,MPI_COMM_WORLD);
}
```

...

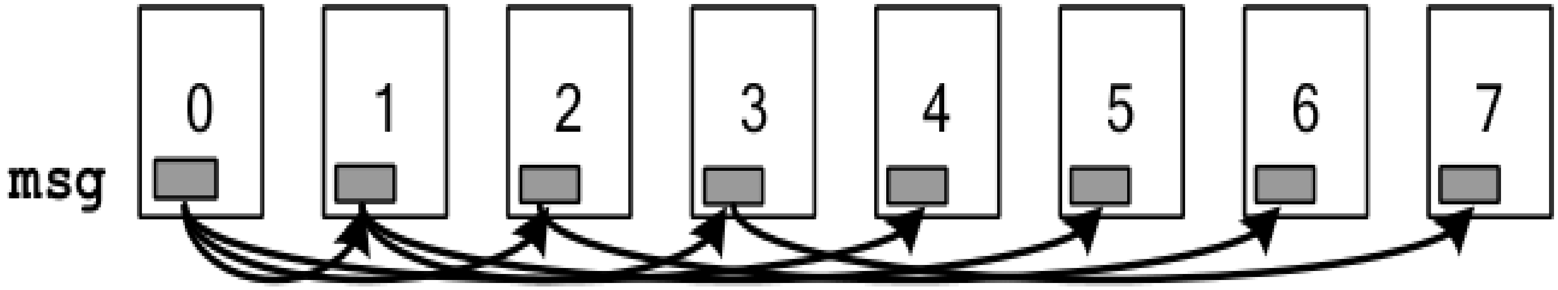


Is that efficient? → We need **size-1** communication steps!!

MPI_Bcast - Collective Comm (2/2)

```
int MPI_Bcast(void* message,int count,MPI_Datatype
                datatype, int root, MPI_Comm comm)
```

→`message` is sent from process `rank` to all other processes of the communicator comm



What about now? Efficient? → We need **$\log_2$(size)** communication steps!!

MPI_Barrier & MPI_Reduce

```
int MPI_Barrier(MPI_Comm comm)
```

→ Call returns after all processes have called the function

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int
               target, MPI_Comm comm)
```

→ Combines the `sendbuf` operand of each process using the operation specified by `MPI_Op op`.

→ Returns combined values in `recvbuf` of process with rank `target`

MPI_Barrier & MPI_Bcast - Example

```
int rank;
MPI_Init(&argc, &argv);
MPI_Status status;
int key = 0;

...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  key = 1;
}
MPI_Bcast(&key, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

if (rank != 0) {
  printf("I am %d and I got the key=%d\n", rank, key);
}

MPI_Finalize();
```

Reduction Types

`MPI_MAX`	Maximum	C integers and floating point
`MPI_MIN`	Minimum	C integers and floating point
`MPI_SUM`	Sum	C integers and floating point
`MPI_PROD`	Product	C integers and floating point
`MPI_LAND`	Logical AND	C integers
`MPI_BAND`	Bit-wise AND	C integers and byte
`MPI_LOR`	Logical OR	C integers
`MPI_BOR`	Bit-wise OR	C integers and byte
`MPI_LXOR`	Logical XOR	C integers
`MPI_BXOR`	Bit-wise XOR	C integers and byte

...

MPI_Reduce example

```
//Compute f(0) and f(1) in parallel and find their sum
#include <mpi.h>

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

 int mypart,sum,rank;
 MPI_Status status;
 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 mypart = f(rank);
 MPI_Reduce(&mypart,&sum,1,MPI_INT,MPI_SUM,0,
             MPI_COMM_WORLD);

 MPI_Finalize();
}
```