Message-Passing Computing

Software Tools for Clusters

Late 1980's	Parallel Virtual Machine (PVM) - developed Became very popular.
Mid 1990's -	Message-Passing Interface (MPI) - standard defined.

Based upon Message Passing Parallel Programming model.

Both provide a set of user-level libraries for message passing. Use with sequential programming languages (C, C++, ...).

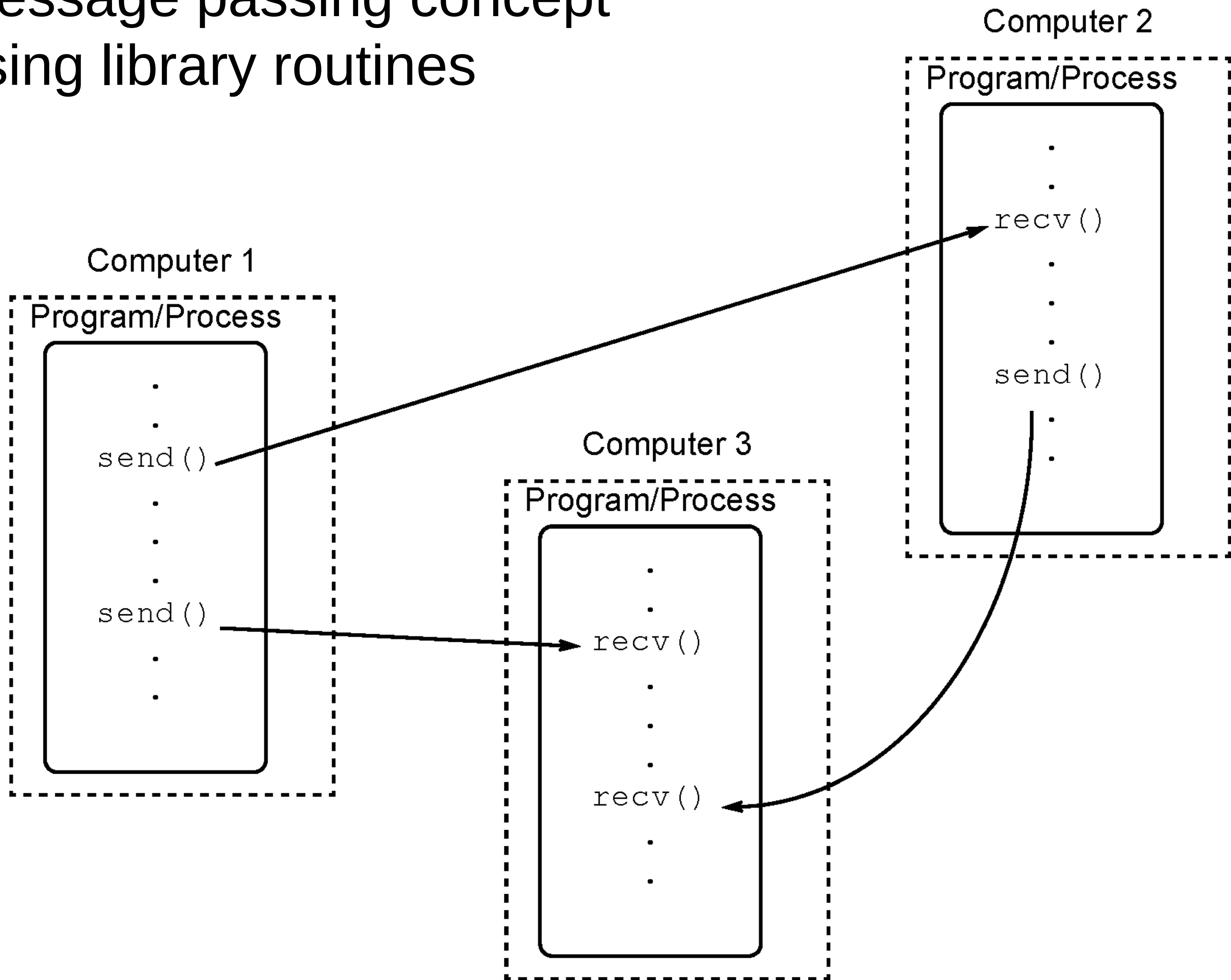
MPI
(Message Passing Interface)

- Message passing library standard developed by group of academics and industrial partners to foster more widespread use and portability.

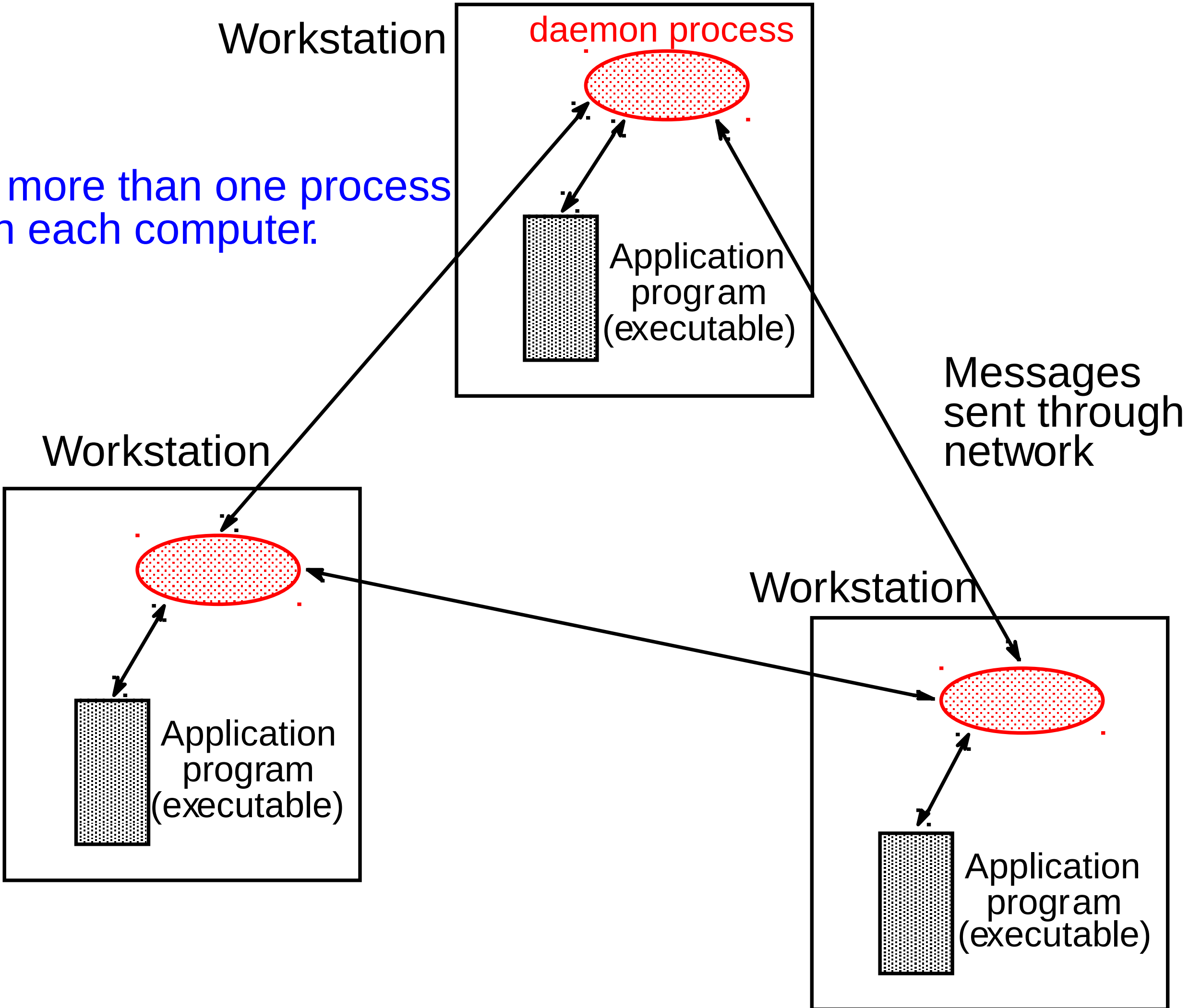
- Defines routines, not implementation.

- Several free implementations exist.

Message passing concept using library routines



Message routing between computers typically done by **daemon processes** installed on computers that form the "virtual machine".



Message-Passing Programming using **User-level** Message-Passing Libraries

Two primary mechanisms needed:

1. A method of creating processes for execution on different computers

2. A method of sending and receiving messages

In the Beginning there were sockets

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

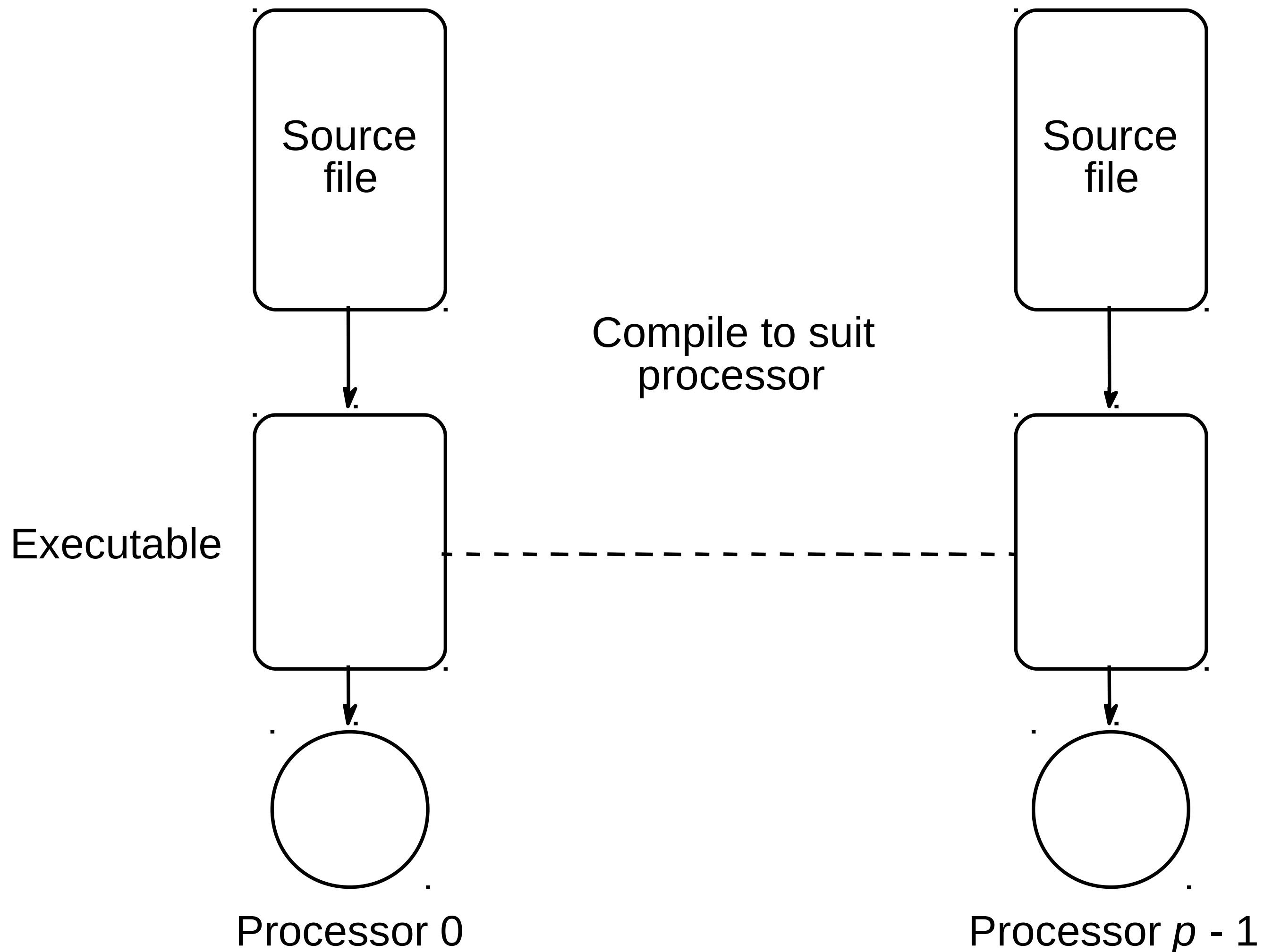
```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

A demo by Guy. See “code” folder
Window1: python server.py , Window2: python client.py
Source: http://docs.python.org/library/socket.html

Creating processes on different computers

Multiple program, multiple data (MPMD) model

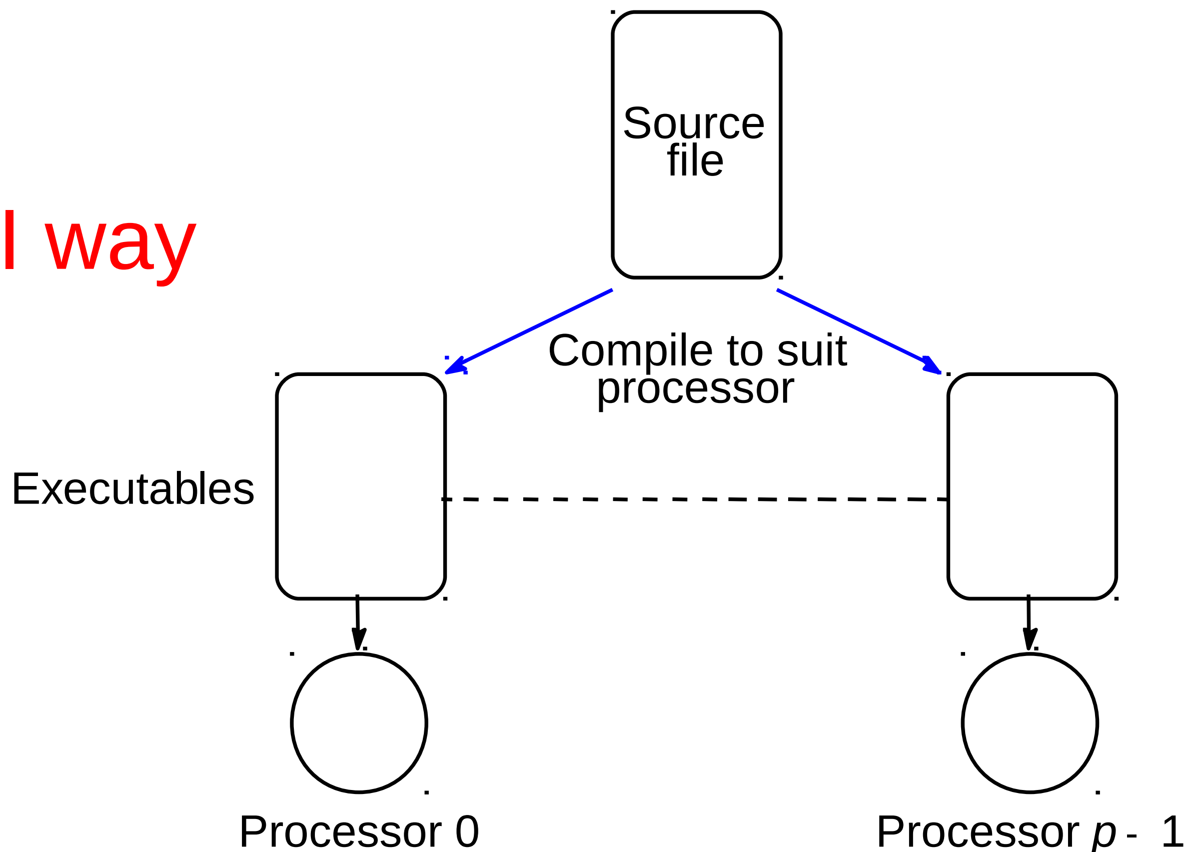
- Different programs executed by each processor



Single Program Multiple Data (SPMD) model

- Same program executed by each processor
- Control statements select different parts for each processor to execute.

Basic MPI way



In MPI, processes within a defined communicating group given a number called a **rank** starting from zero onwards.

Program uses control constructs, typically IF statements, to direct processes to perform specific actions.

Example

```
if (rank == 0) ...  /* do this */;
if (rank == 1) ...  /* do this */;
.
.
.
```

Master-Slave (Worker) approach

Usually computation constructed as a master-slave model

One process (the master), performs one set of actions and all the other processes (the slaves) perform identical actions although on different data, i.e.

```
if (rank == 0) ...  /* master do this */;
else ...            /* all slaves do this */;
```

Static process creation

- All executables started together.
- Done when one starts the compiled programs.
- Normal MPI way.

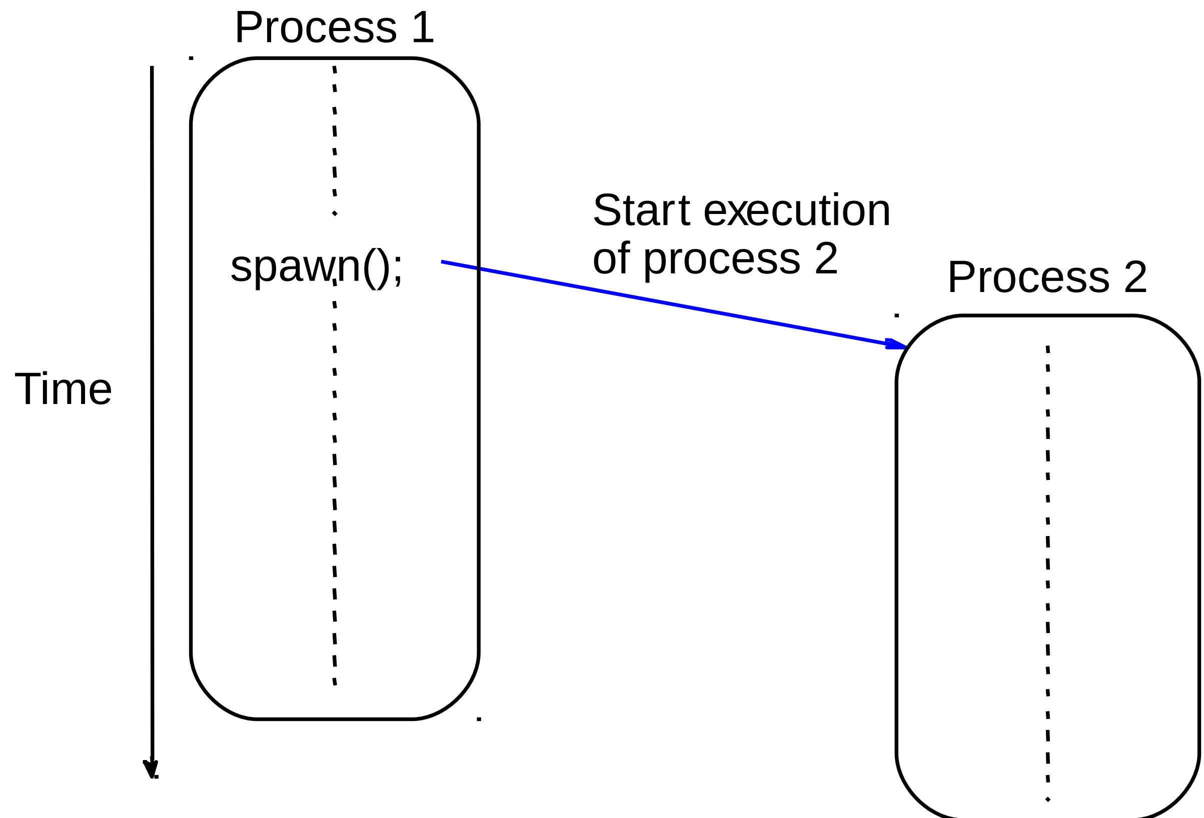
Multiple Program Multiple Data (MPMD) Model with Dynamic Process Creation

- One processor executes master process.
- Other processes started from within master process

Available in MPI-2

Might find applicability if do not initially how many processes needed.

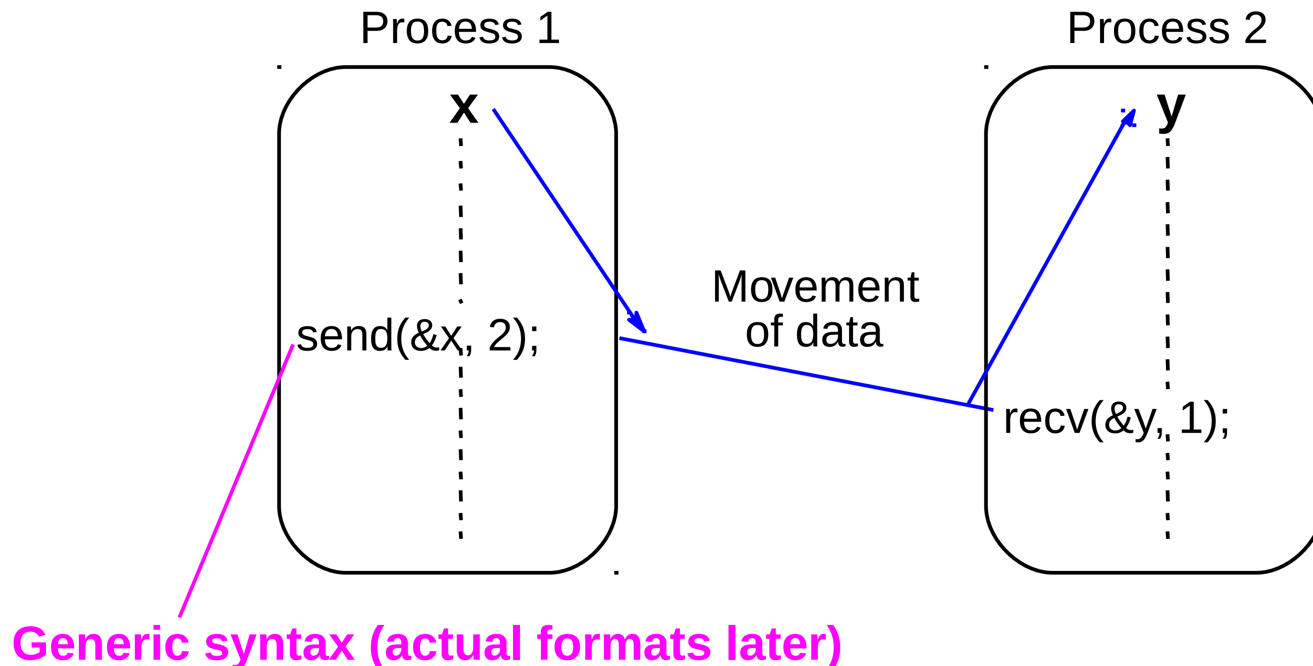
Does have a process creation overhead.



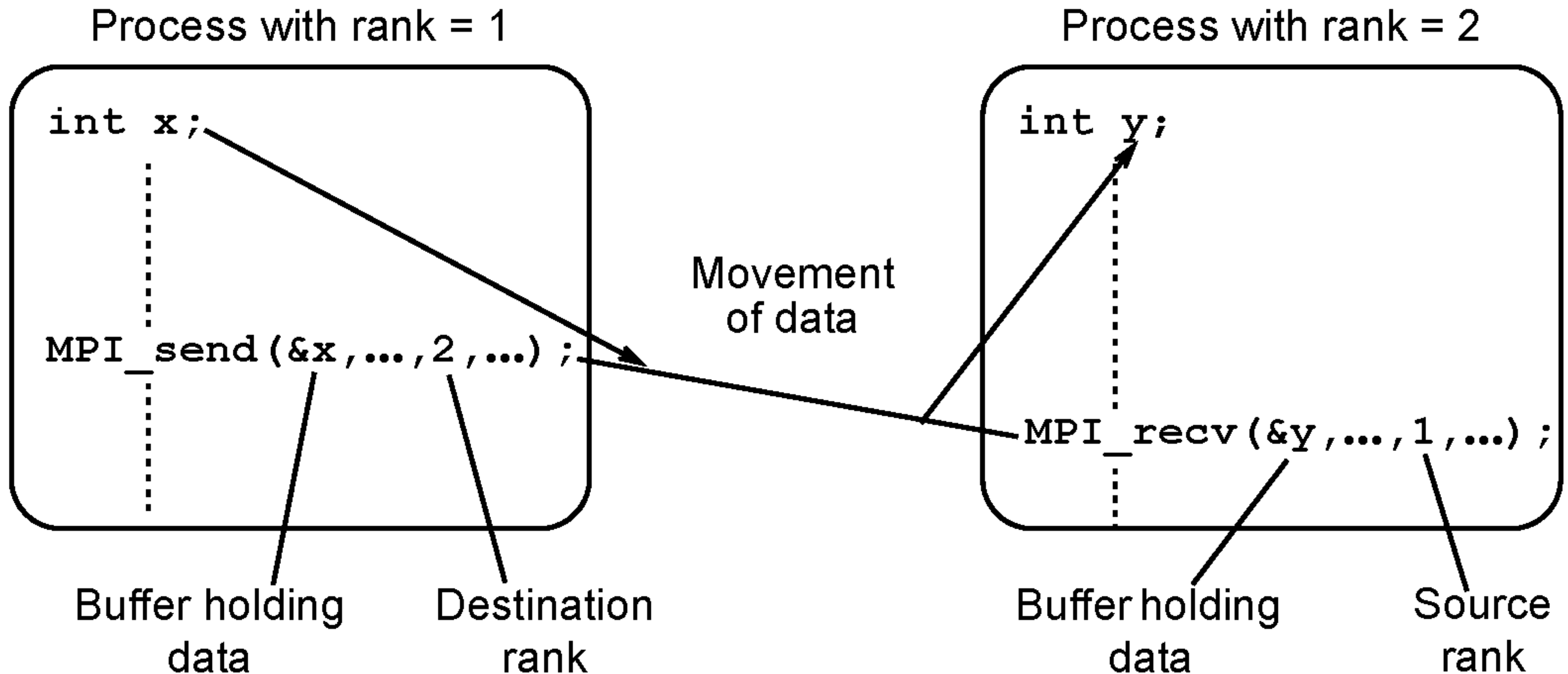
Methods of sending and receiving messages

Basic “**point-to-point**” Send and Receive Routines

Passing a message between processes using send() and recv() library calls:



MPI point-to-point message passing using MPI_send() and MPI_recv() library calls



Semantics of MPI_Send() and MPI_Recv()

Called blocking, which means in MPI that routine waits until all its local actions have taken place before returning.

After returning, any local variables used can be altered without affecting message transfer.

MPI_Send() - Message may not reached its destination but process can continue in the knowledge that message safely on its way.

MPI_Recv() – Returns when message received and data collected. Will cause process to **stall** until message received.

Other versions of MPI_Send() and MPI_Recv() have different semantics.

Message Tag

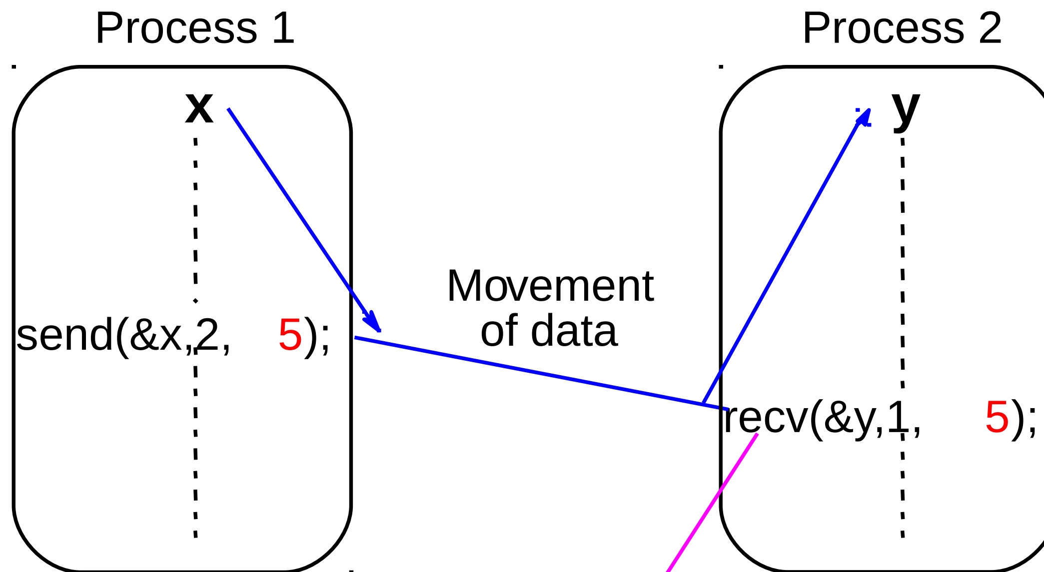
- Used to differentiate between different types of messages being sent.

- Message tag is carried within message.

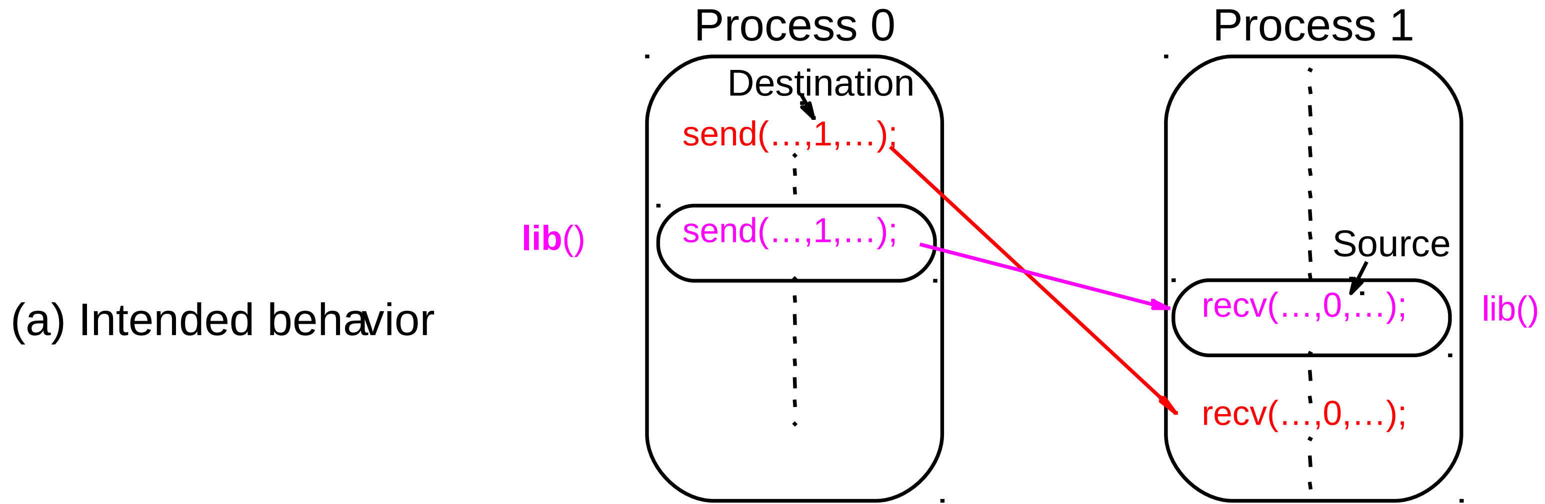
- If special type matching is not required, a **wild card** message tag used. Then recv() will match with any send().

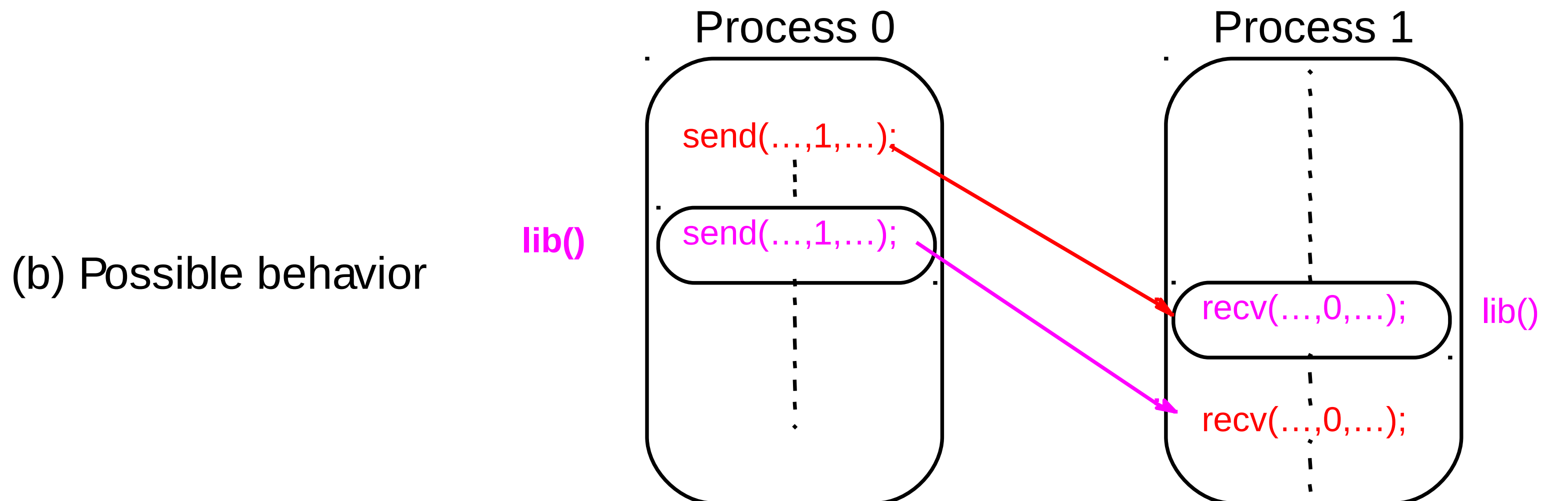
Message Tag Example

To send a message, x, with message tag 5 from a source process, 1, to a destination process, 2, and assign to y:



Unsafe message passing - Example





MPI Solution
"**Communicators**"

- Defines a **communication domain** - a set of processes that are allowed to communicate between themselves.

- Communication domains of libraries can be separated from that of a user program.

- <u>Used in all point-to-point and collective MPI message-passing communications.</u>

Default Communicator

MPI_COMM_WORLD

- Exists as first communicator for
  all processes existing in the application.

- A set of MPI routines exists for forming communicators.

- Processes have a “**rank**” in a communicator.

Using **SPMD** Computational Model

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

where master() and slave() are to be executed by master process and slave process, respectively.

Parameters of blocking send

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

- buf — Address of send buffer
- count — Number of items to send
- datatype — Datatype of each item
- dest — Rank of destination process
- tag — Message tag
- comm — Communicator

Parameters of blocking receive

MPI_Recv(buf, count, datatype, src, tag, comm, status)

- buf — Address of receive buffer
- count — Maximum number of items to receive
- datatype — Datatype of each item
- src — Rank of source process
- tag — Message tag
- comm — Communicator
- status — Status after operation

Example

To send an integer x from process 0 to process 1,

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find
   rank */
if (myrank == 0) {
   int x;
   MPI_Send(&x, 1, MPI_INT, 1, msgtag,
   MPI_COMM_WORLD);
} else if (myrank == 1) {
   int x;
   MPI_Recv(&x, 1, MPI_INT,
   0,msgtag,MPI_COMM_WORLD,status);
}
```

Sample MPI Hello World program

```
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv ) {
    char message[20];
    int i,rank, size, type=99;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank == 0) {
    strcpy(message, "Hello, world");
    for (i=1; i<size; i++)
    MPI_Send(message,13,MPI_CHAR,i,type,MPI_COMM_WORLD);
    } else
    MPI_Recv(message,20,MPI_CHAR,0,type,MPI_COMM_WORLD,&status);
    printf( "Message from process =%d : %.13s\n", rank,message);
    MPI_Finalize();
}
```

Program sends message “Hello World” from master process (rank = 0) to each of the other processes (rank != 0). Then, all processes execute a printf statement.

In MPI, standard output automatically redirected from remote computers to the user’s console so final result will be

```
Message from process =1 : Hello, world
Message from process =0 : Hello, world
Message from process =2 : Hello, world
Message from process =3 : Hello, world
...
```

except that the order of messages might be different but is unlikely to be in ascending order of process ID; it will depend upon how the processes are scheduled.

Setting Up the Message Passing Environment

Usually computers specified in a file, called a **hostfile** or **machines** file.

File contains names of computers and possibly number of processes that should run on each computer.

Implementation-specific algorithm selects computers from list to run user programs.

Users may create their own machines file for their program.

Example

hobbit1.ee.bgu.ac.il

hobbit2.ee.bgu.ac.il

hobbit3.ee.bgu.ac.il

hobbit4.ee.bgu.ac.il

hobbit5.ee.bgu.ac.il

If a machines file not specified, a default machines file used or it may be that program will only run on a **single computer**.

Compiling/Executing MPI Programs

- Minor differences in the command lines required depending upon MPI implementation.

- For the assignments, we will use MPICH or MPICH-2.

- Generally, a machines file need to be present that lists all the computers to be used. MPI then uses those computers listed. Otherwise it will simply run on one computer

MPICH and MPICH-2

- Both Windows and Linux versions
- Very easy to install on a Windows system.

MPICH Commands

Two basic commands:

- mpicc, a script to compile MPI programs

- mpirun, the original command to execute an MPI program, or
- mpiexec - MPI-2 standard command mpiexec replaces mpirun although mpirun still exists.

Compiling/executing (SPMD) MPI program

For MPICH. At a command line:

To start MPI: Nothing special.

To compile MPI programs:

for C `mpicc -o prog prog.c`

for C++ `mpiCC -o prog prog.cpp`

To execute MPI program:

A positive integer

```
mpiexec -n no_procs prog
```

or

```
mpirun -np no_procs prog
```

Executing MPICH program on multiple computers

Create a file called say "machines" containing the list of machines, say:

hobbit1.ee.bgu.ac.il

hobbit2.ee.bgu.ac.il

hobbit3.ee.bgu.ac.il

hobbit4.ee.bgu.ac.il

hobbit5.ee.bgu.ac.il

mpiexec -machinefile machines -n 4 prog

Or:

mpirun -machinefile machines -np 4 prog

would run prog with four processes.

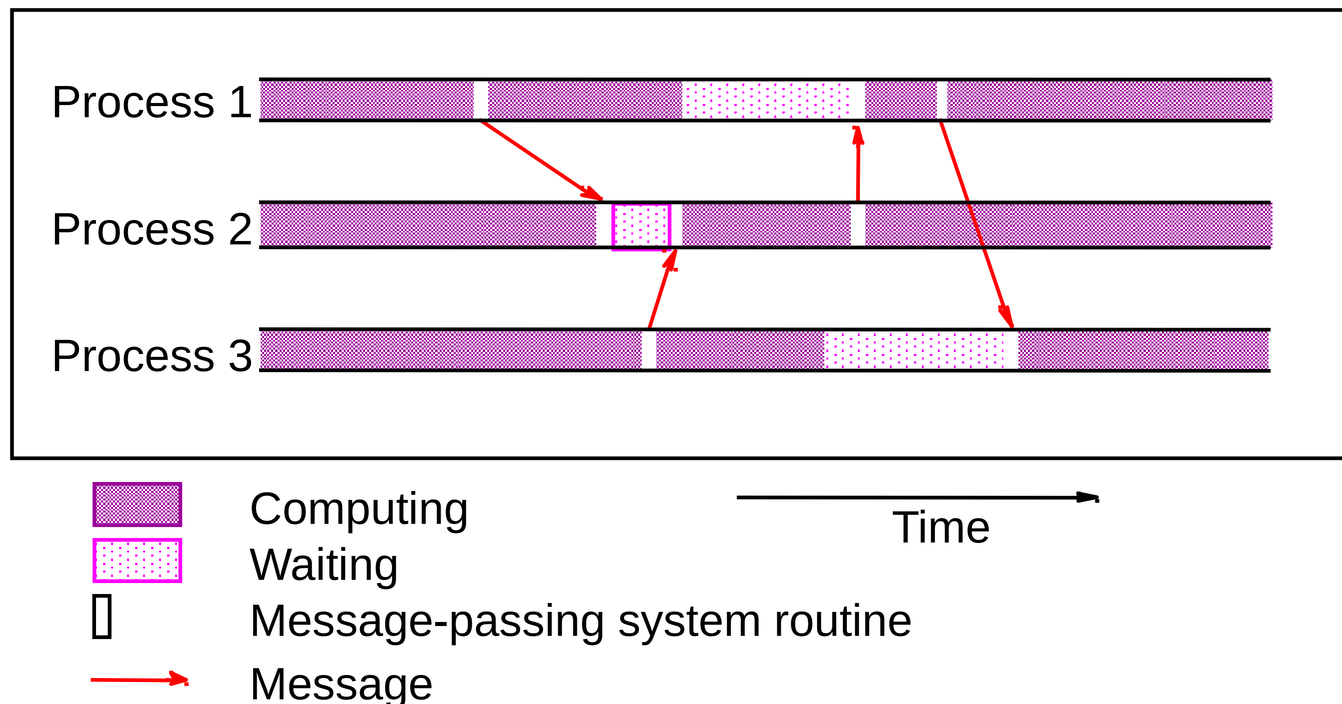
Each processes would execute on one of machines in list. MPI would cycle through list of machines giving processes to machines.

Can also specify number of processes on a particular machine by adding that number after machine name.)

Debugging/Evaluating Parallel Programs Empirically

Visualization Tools

Programs can be watched as they are executed in a space-time diagram (or process-time diagram):



Implementations of visualization tools are available for MPI.

An example is the Upshot program visualization system.

Guy:

Upshot → Jumpshot (A Java based tool)

Profiling with Jumpshop

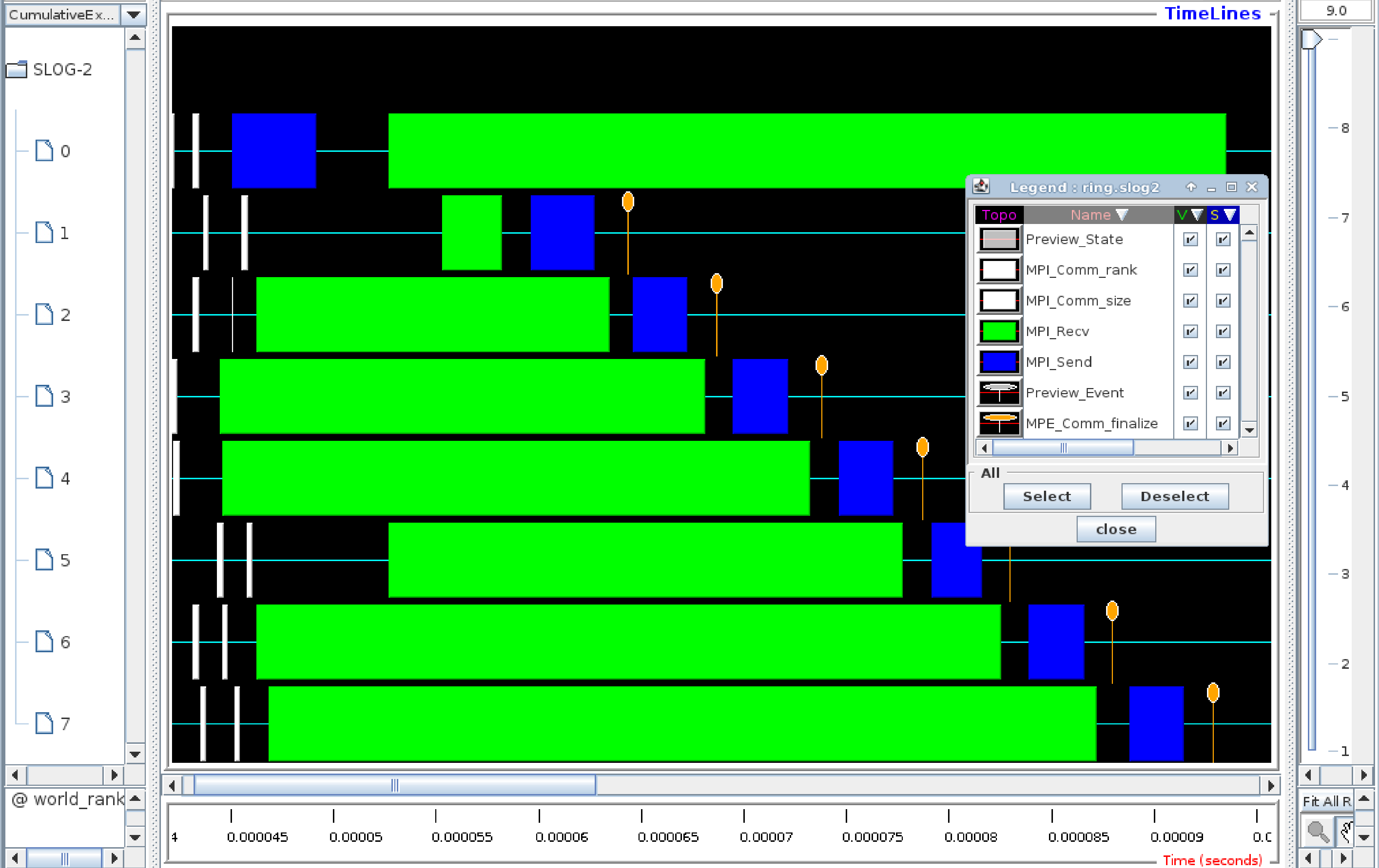
Environment: Linux (my laptop)

Directory:
~/Documents/Teaching/BGU/PP/PP2015A/
lectures/02/code

MPI version: MPICH2 + MPE

Instructions

```
mpecc -g -mpilog -o ring ring.c
mpirun -np 8 ./ring
/usr/local/mpe/bin/clog2TOslog2  \
./ring.clog2
jumpshot ./ring.slog2 &
```



Lowest / Max. Depth
0 / 0
Zoom Level
1
Global Min Time
0.0000405312
View Init Time
0.0000421338
Zoom Focus Time
0.000068934
View Final Time
0.0000957342
Global Max Time
0.0001840591
Time Per Pixel
0.0000000665
Row
Row Count
9.0
CumulativeEx...
TimeLines
SLOG-2
0
1
2
3
4
5
6
7
Legend : ring.slog2
Topo
Name
V
S
Preview_State
MPI_Comm_rank
MPI_Comm_size
MPI_Recv
MPI_Send
Preview_Event
MPE_Comm_finalize
All
Select
Deselect
close
@ world_rank
Fit All R
0.000045
0.00005
0.000055
0.00006
0.000065
0.00007
0.000075
0.00008
0.000085
0.00009
Time (seconds)

Evaluating Programs Empirically
Measuring Execution Time

To measure execution time between point L1 and point L2 in code, might have construction such as:

```
                         .
L1: time(&t1);       /* start timer */
     .
     .
L2: time(&t2);      /* stop timer */
          .
elapsed_Time = difftime(t2, t1); /*time=t2-t1*/

printf("Elapsed time=%5.2f secs",elapsed_Time);
```

MPI provides the routine **MPI_Wtime()** for returning time (in seconds):

```
double start_time, end_time, exe_time;

start_time = MPI_Wtime();

        .
        .

end_time = MPI_Wtime();
exe_time = end_time - start_time;
```

* עד כאן מצגת זו