Distributed Programming Techniques

Alessandro Pellegrini

Dipartimento di Ingegneria Informatica,
Automatica e Gestionale

A.Y. 2014/2015

Lectures Outline

- Introduction to MPI
- Event-Driven Programming and Simulation
- Parallel Discrete Event Simulation
- Time Warp Synchronization Protocol
- The ROme OpTimistic Simulator (ROOT-Sim)

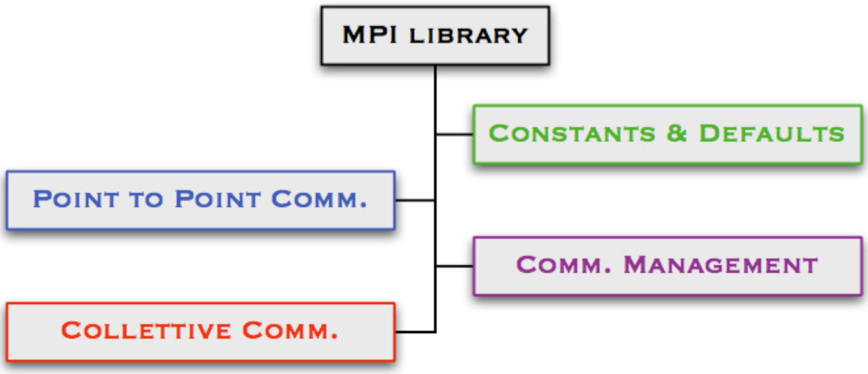
MPI: Message Passing Interface [1]

- The world of MIMD computers is, for the most part, divided into:
  - *Distributed-memory* systems
  - *Shared-memory* systems
- To program distributed-memory systems we use **message passing**
  - A program running on a core-memory pair is called a *process*
  - Two processes can communicate through:
    - *send*
    - *receive*
- **MPI** is a *library* of functions that can be called from C, C++ and Fortran programs
  - It can generate and handle the group of processes
  - Allows to exchange data between each other

What do we expect from MPI

- Communication management functions
  - Definition/identification of group of processes involved in the communication
  - Definition/handling of each process' identity, within a group
- Explicit functions for exchanging messages
  - Send/receive data from a process
  - Send/receive data from a group of processes

MPI Library Structure

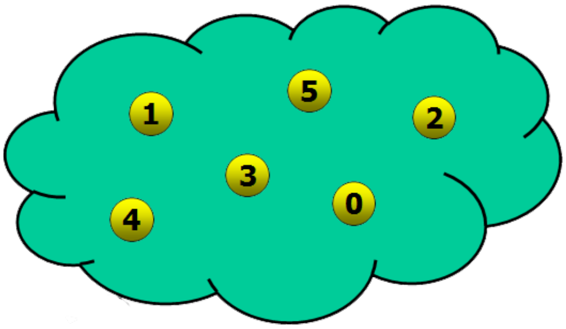


MPI Calls Format

- `err = MPI_Xxxxx(params, ...)`
  - `MPI_` is a prefix used to identify every MPI call
  - The first letter, after the prefix, is always capital
  - Almost every function return an integer error code
  - Constants are all capitalized

Communicators

- A *communicator* describes a collection of processes and a set of attributes
- Each process can has a unique ID, within a communicator
- Processes can send messages to each other only if they are in the same communicator
- More than one communicator can exist



Communication Environment

- `int MPI_Init(int *argc, char **argv);`
  - Is the first call in *every* MPI program
  - Can be called only once
  - Initializes the communication environment
  - Defines the `MPI_COMM_WORLD` communicator, consisting of all the processes started by the user upon program startup

- `int MPI_Finalize(void);`
  - Finalizes the communication environment
  - Releases all MPI resources
  - No additiona MPI call can appear after it

Getting Information from the Communicator

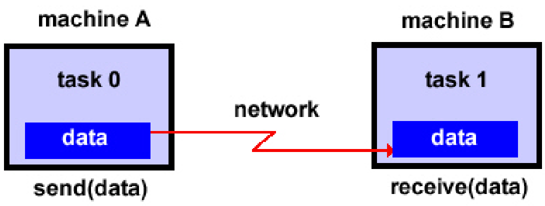
- *Communicator Size*
  - A communicator's *size* is the (integer) cardinality of the set of processes which it contains
  - A process can get the size of the communicator it belongs to:
    - `int MPI_Comm_size(MPI_Comm comm, int *size);`
- *Process Rank*
  - A process can get its unique ID (*rank*):
    - `int MPI_Comm_rank(MPI_Comm comm, int *rank);`
  - Ranks are in the range $[0, size - 1]$

Example: MPI First Program

```c
#include <stdio.h>
#include <mpi.h>

void main (int argc, char *argv[]) {
    int myrank, size;

    /* Initialize MPI */
    MPI_Init(&argc, &argv);
    /* Get my rank and the total number of processes */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Process %d of %d\n", myrank, size);

    /* Terminate MPI */
    MPI_Finalize();
}
```

Interprocess Communication

- Processes can communicate *explicitly*
- Messages can be exchanged between processes belonging to the same communicator
- *point-to-point* is the easiest form of communication



A Message

- A *message* is the communication means for transferring data between processes
- Every message is divided into:
  - **Envelope**
    - *source*: ID of the sender
    - *destination*: ID of the receiver
    - *communicator*: communicator which both processes belong to
    - *tag*: ID of a message, useful to differentiate messages exchanged between the same processes
  - **Body**
    - *buffer*: message content
    - *datatype*: type of data contained within the message
    - *count*: number of occurrences of *datatype* to be sent

How to Send a Message

- Sender process calls an MPI primitive used to uniquely identify the message's *envelope* and *body*
  - Sender's identity is implicit: it's the caller's
  - Other elements are specified through the API call
- `int MPI_Send(void *buff, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);`

How to Receive a Message

- Destination process must call another MPI primitive to explicitly tell the library to deliver a message
- Arguments passed to the API allow to uniquely identify the envelope of the message to be delivered
- If no envelope matches the specified arguments, the operation cannot complete until a matching one is found among the *pending messages*
- Destination process must prepare a memory area large enough to store the message's *body*
- `MPI_Recv(void *buff, int count, MPI_Datatype, int src, int tag, MPI_Comm comm, MPI_Status *status);`

MPI Datatypes

MPI Datatype	C Datatype
MPI_INT	signed int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	signed char

- To send complex data types (e.g. a struct), MPI_CHAR is used as *datatype*, and sizeof(struct) as *count*

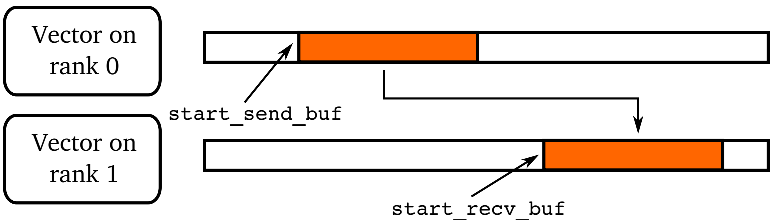
Example: Sending and Receiving an integer

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

void main (int argc, char *argv[]) {
    MPI_Status status;
    int myrank, size;
    int data_int; // What we want to communicate

    /* Initialize Everything */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if(rank == 0) {
        data_int = 10;
        MPI_Send(&data_int, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);
```

Example: Sending and Receiving an integer (2)

```
    } else {
        MPI_Recv(&data_int, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, &
            status);
        printf("Process 1 receives %d from process 0.\n", data_int);
    }

    /* Quit */
    MPI_Finalize();

    return 0;
}
```


Example: Sending and Receiving an array portion




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

#define VSIZE 50
#define BORDER 12

void main (int argc, char *argv[]) {
    MPI_Status status;
    int i, rank, nprocs;
```

Example: Sending and Receiving an array portion (2)

```c
    int start_send_buf = BORDER;
    int start_recv_buf = VSIZE - BORDER;
    int length = 10;
    int vector[VSIZE];

    /* Initialize Everything */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    /* All processes must initialize vector */
    for(i = 0; i < VSIZE; i++) vector[i] = rank;

    if(rank == 0) {
        MPI_Send(&vector[start_send_buf], length, MPI_INT, 1, 123,
            MPI_COMM_WORLD);
```


Example: Sending and Receiving an array portion (3)

```c
    } else {
        MPI_Recv(&vector[start_recv_buf], length, MPI_INT, 0, 123,
            MPI_COMM_WORLD, &status);
    }

    /* Quit */
    MPI_Finalize();

    return 0;
}
```

OpenMPI: How to Compile

- MPI is a library, therefore we have to link our program with it
- We have to instruct the compiler on:
  - the *include files path* (`-I`)
  - the *library path* (`-L`)
  - the *library name* (`-l`)
- There are **not** standard names...
- ...to ease the task, upon installation wrappers are created which call the compiler accordingly:

```
mpicc source.c -o executable
```


OpenMPI: How to Launch

- An MPI executable must initialize the library beforehand
- This entails setting up the whole communication environment and starting a certain number of parallel/distributed processes
- To simplify, an *MPI launcher* exists, which performs these tasks transparently.
- It will ask for:
  - *Number* of processes
  - '*Name*' of processing nodes to use for computation
  - Arguments to the parallel process

OpenMPI: How to Launch (2)

- The launcher is `mpiexec` (on legacy implementations it's `mpirun`)
- The *number of processes* is specified directly as an argument:

```
mpiexec -n 3
```

- The *nodes identity* can be specified in two ways:
  - Directly in the command line:

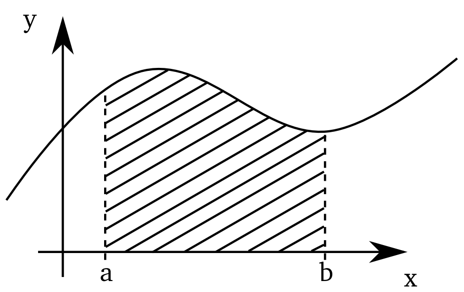
```
mpiexec -H node1,node2,node3 -n 3
```

  - Creating the *hostfile* text file, where each line contains the *host* to be used followed by the keyword `slots=XX`, where *XX* is the number of processes to be spawned on that node

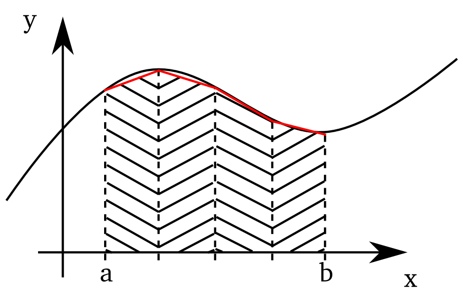
```
mpiexec -hostfile my_hostfile -np 6
```


Exercise: The Trapezoidal Rule

- The Trapezoidal Rule allows to approximate the area between:
  - a function $y = f(x)$
  - two vertical lines $x_0 = a, x_1 = b$
  - The x-axis

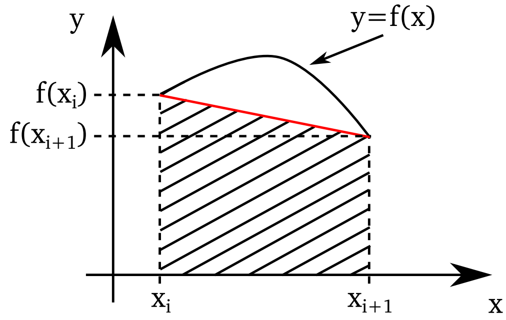


- The interval is divided into n equal subintervals
- The area is approximated as the one of a trapezoid



Exercise: The Trapezoidal Rule (2)

- If the endpoints of the subinterval are x_i and x_{i+1}, its length is $h = x_{i+1} - x_i$
- The height of the two vertical segments are $f(x_i)$ and $f(x_{i+1})$.



- The fourth side is the secant line joining the points where the vertical segments cross the graph
- The area of the trapezoid is therefore:

$$A_i = \frac{h}{2}[f(x) + f(x_{i+1})]$$

Exercise: The Trapezoidal Rule (3)

- Since the n subitervals all have the same length, we know that:

$$h = \frac{b-a}{n}$$

- Thus if we call the leftmost endpoint x_0 and the rightmost endpoint x_n, we have:

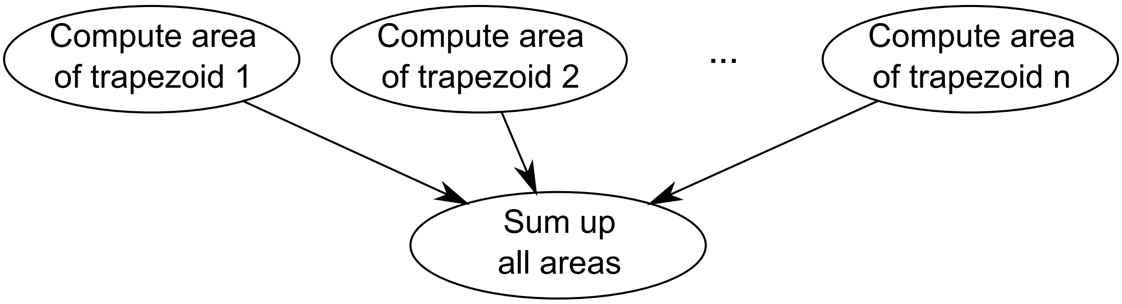
$$x_0 = a, x_1 = a + h, x_2 = a + 2h, \ldots, x_{n-1} = a + (n-1)h, x_n = b$$

- The sum of the areas of the trapezoids (our approximation) is:

$$A = h\left[\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \ldots + f(x_{n-1}) + \frac{f(x_n)}{2}\right]$$

Parallelizing the Trapezoidal Rule

- We can design a paralell program using four basic steps:
  1. Partition the problem solution into tasks
  2. Identify the communication channels between the task
  3. Aggregate the tasks into composite tasks
  4. Map the composite tasks to cores



Trapezoidal Rule: the Code

```c
#include <stdio.h>
#include <mpi.h>

int main(void) {
   int my_rank, comm_sz, n = 1024, local_n;
   double a = 0.0, b = 3.0, h, local_a, local_b;
   double local_int, total_int;
   int source;

   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   h = (b-a)/n; // It's the same for all processes
   local_n = n/comm_sz; // It's the same for all processes

```

Trapezoidal Rule: the Code (2)

```c
   local_a = a + my_rank * local_n * h;
   local_b = local_a + local_n * h;
   local_int = Trap(local_a, local_b, local_n, h);

   // Exchange the estimates
   if(my_rank != 0) {
      MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
   } else {
      total_int = local_int;
      for(source = 1; source < comm_sz; source++) {
         MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         total_int += local_int;
      }
   }

```


Trapezoidal Rule: the Code (3)

```c
   if(my_rank == 0) {
      printf("With %d trapezoids, the integral estimate from %f to %
            f is %f\n", n, a, b, total_int);
   }

   MPI_Finalize();
   return 0;
}

double Trap(double left_endpt, double right_endp, int trap_count,
      double base_len) {
   double estimate, x;
   int i;

   estimate = (f(left_endpt) + f(right_endpt)) / 2.0;
   for(i = 1; i <= trap_count - 1; i++) {
```

Trapezoidal Rule: the Code (4)

```
      x = left_endpt + i * base_len;
      estimate += f(x);
   }

   estimate = estimate * base_len;

   return estimate;
}
```


Dealing with Input

- What if in the previous program we want to specify *a*, *b*, *c*?
- Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin.
- In order to use, say, scanf, we need to branch on process rank:

```
void get_input(int rank, int sz, double *a, double *b, int *n) {
   int dest;

   if(rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d\n", a, b, n);
      for(dest = 1; dest < sz; dest++) {
         MPI_Send(a, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(b, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
```

Dealing with Input (2)

```
            MPI_Send(n, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        MPI_Recv(b, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        MPI_Recv(n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }
}
```


Dealing with Output

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

int main(void) {
    int my_rank, comm_sz;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    printf("Proc %d of %d: Hello World!\n", my_rank, comm_sz);

    MPI_Finalize();
}
```

Dealing with Output (2)

- The order of the output lines is unpredictable, e.g.:

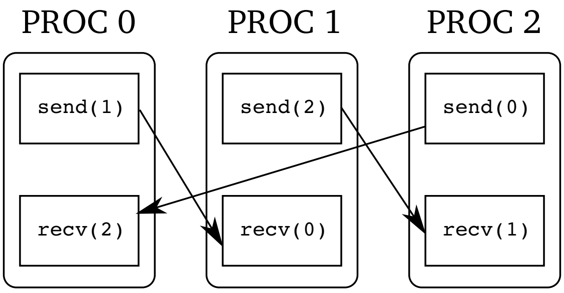
```
Proc 0 of 6 > Hello World!
Proc 1 of 6 > Hello World!
Proc 2 of 6 > Hello World!
Proc 5 of 6 > Hello World!
Proc 4 of 6 > Hello World!
Proc 3 of 6 > Hello World!
```

- MPI processes are *competing* for accessing to the shared `stdout` output device
- It's impossible to predict the order in which the processes' output will be queued up: **nondeterminism**

Some potential pitfalls

- If a process tries to receive a message and there's no matching send, the process will **hang**
- We need to be sure that every receive has a matching send
- If the tags don't match, or there is an error in the destination/source id
  - either a process will hang
  - or the receive will match *another* send!
- If there is no matching receive to an MPI_Send, the sender will hang!
- Use `MPI_Isend` and `MPI_Irecv` as non-blocking counterparts
- `MPI_Wait` and `MPI_Test` allow to check if a receive has completed

Periodic Circular Shift



- Each process generates an array, containing its rank in each item
- Each process sends the array to the neighbour process
- Each process receives the array from the neighbour and stores it into another array

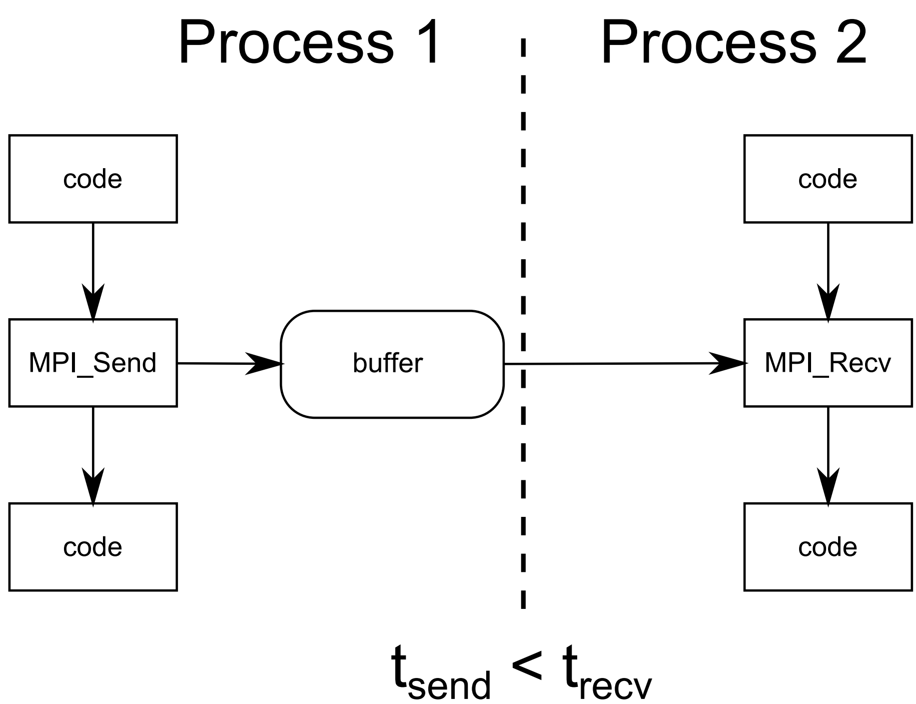
Periodic Circular Shift (2)

```c
int my_rank, comm_sz, to, from, i, A[SIZE], B[SIZE];

MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

for(i = 0; i < SIZE; i++) A[i] = my_rank;

to = (my_rank + 1) % comm_sz;
from = (my_rank + comm_sz - 1) % comm_sz;
MPI_Send(A, SIZE, MPI_INT, to, TAG, MPI_COMM_WORLD);
MPI_Recv(B, SIZE, MPI_INT, from, TAG, MPI_COMM_WORLD,
     MPI_STATUS_IGNORE);

MPI_Finalize();
```

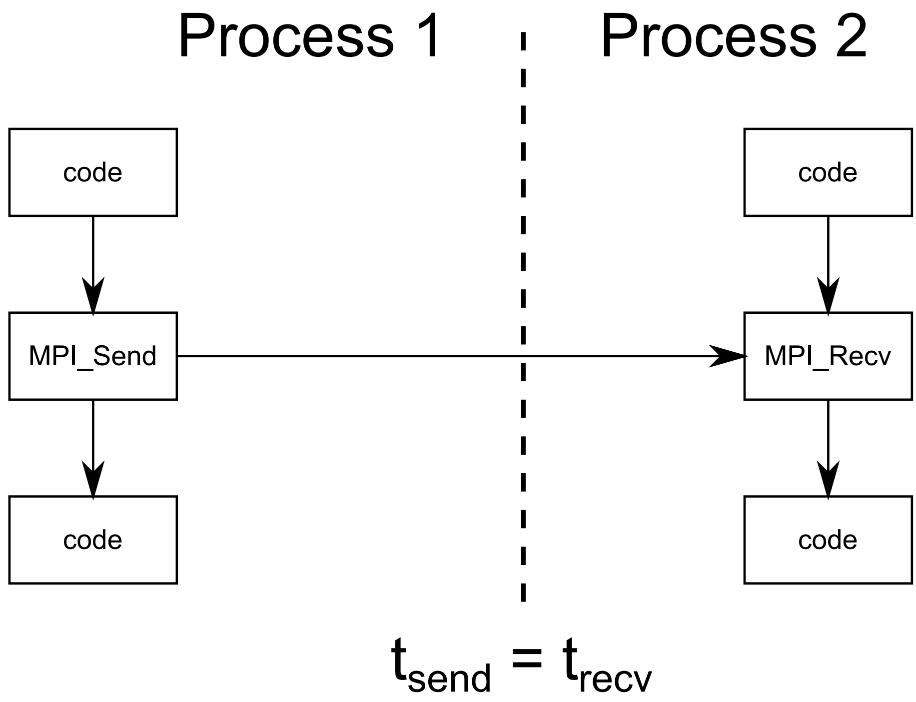
Communication Mechanism

- MPI_Send doesn't return until the delivery of the message is complete, according to two policies:
  - *Buffered*: the message is copied into a system buffer
  - *Synchronous*: the message is directly copied into the receiver's buffer
- The actual behaviour depends on message's size:
  - System buffer is fixed-size, thus can be too small to contain a message
  - An alternative is to use MPI_Bsend which allows to specify a user buffer for message passing
- If buffered, MPI_Send returns after the message has been copied in the local buffer

Communication Mechanism (2)



Communication Mechanism (3)



Why Deadlock?

- The algorithm can be summarized as:

```
if(rank == 0) {
   send A to process 1
   receive B from process 1
} else if (rank == 1) {
   send B to process 1
   receive A from process 1
}
```

- For large `SIZE`, there a 2 send operations waiting for a receive.
- Receives can complete only after the sends complete!

Deadlock: naive solution

- We can rearrange the operations:

```
if(rank == 0) {
   send A to process 1
   receive B from process 1
} else if (rank == 1) {
   receive A from process 1
   send B to process 1
}
```

- What if there are more than 2 processes?

MPI_Sendrecv

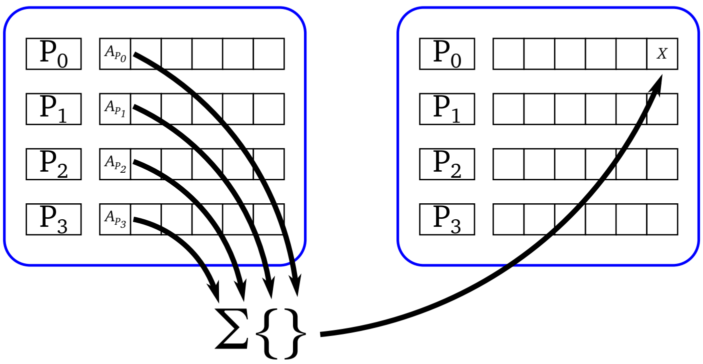
- We need a facility which internally orders send and receive operations to avoid deadlock
- MPI_Sendrecv can be used when a process must send and receive data *at the same time*

```
int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype
s_dtype, int dest, int stag, void *dbuf, int dcount,
MPI_Datatype d_type, int src, int dtag, MPI_Comm comm,
MPI_Status *status);
```

Collective Communications

- Several applications need to communicate among all (or a group) of processes
  - For example, the trapezoidal rule implementation
- MPI provides some communication primitives implementing collective communications
  - They ease the programmer from the burden of sending information multiple times
  - They are more efficient!
- There are three classes:
  - **all-to-one**
  - **one-to-all**
  - **all-to-all**

Reduce



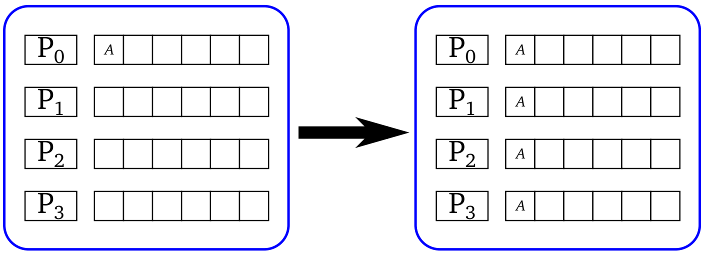
- Collects data from all involved processes
- Applies an operator to reduce the values to a single one
- Stores the result in the *root* process

Reduce (2)

```
int MPI_Reduce(void *sbuf, void *rbuf, int count,
MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm);
```

- Principal operators are
  - Sum (`MPI_SUM`) and Product (`MPI_PROD`)
  - Maximum (`MPI_MAX`) and Minimum (`MPI_MIN`)
  - Logical And (`MPI_LAND`) and Bitwise And (`MPI_BAND`)
  - Logical Or (`MPI_LOR`) and Bitwise Or (`MPI_BOR`)
  - Logical Xor (`MPI_LXOR`) and Bitwise Xor (`MPI_BXOR`)
- Reduce default operators are associative and commutative
- User-defined operators can be created via `MPI_Op_create`

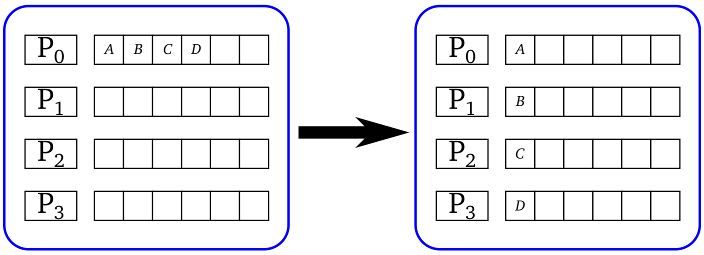
Broadcast



- Copies data from *send* buffer to every process' *receive* buffer
- Belongs to the *one-to-all* class

```
int MPI_Bcast(void *buf, int count, MPI_Datatype dtype,
int root, MPI_Comm comm);
```

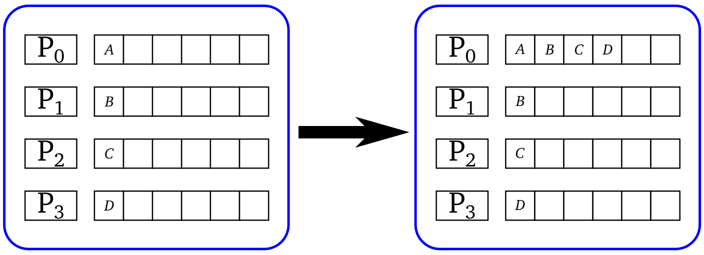
Scatter



- *Root* process
  - divides the data into *N* equal parts
  - send one part to each process in *rank* order
- Belongs to the *one-to-all* class

```
int MPI_Scatter(void *sbuf, int scount, MPI_Datatype
s_dtype, void *rbuf, int rcount, MPI_Datatype r_dtype,
int root, MPI_Comm comm);
```

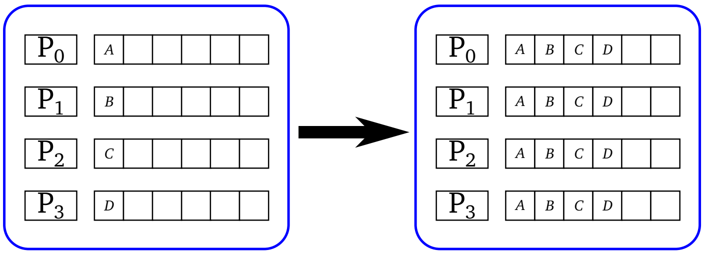

Gather



- Every process (including *root*) sends its data to *root*
- *Root* receives the data and orders them according to the *rank*
- Belongs to the *all-to-one* class

```
int MPI_Gather(void *sbuf, int scount, MPI_Datatype
s_dtype, void *rbuf, int rcount, MPI_Datatype r_dtype,
int root, MPI_Comm comm);
```

All Gather



- It's equivalent to a *gather* operation
- Every process receives the data
- More efficient than a *gather* + *broadcast* operation
- Belongs to the *all-to-all* class

```
int MPI_Allgather(void *sbuf, int scount, MPI_Datatype
s_dtype, void *rbuf, int rcount, MPI_Datatype r_dtype,
MPI_Comm comm);
```


Other Collective Communication Primitives

- `MPI_Barrier`: processes susped their execution until every process has reached the barrier (synchronization primitive)
- `MPI_All_reduce`: the reduction result is sent to every process (it's equivalent to a *reduce* + *broadcast*, but more efficient)
- `MPI_Scatterv` and `MPI_Gatherv`: the logic is the same as *scatter*'s and *gather*'s, but chunks of different size can be exchanged

Event-Driven Programming

- Event-Driven Programming is a programming paradigm in which the flow of the program is determined by *events*
  - Sensors outputs
  - User actions
  - Messages from other programs or threads
- This paradigm is based on a **main loop** divided into two phases:
  - Event selection/detection
  - Event handling
- Events resemble what *interrupts* do in hardware systems

Event Handlers

- An event handler is an *asynchronous callback*
- Each event represents a piece of application-level information, delivered from the underlying framework
  - In a GUI events can be mouse movements, key pression, action selection, . . .
- Events are processed by an event dispatcher which manages associations between events and event handlers and *notifies* the correct handler
- Events can be queued for later processing if the involved handler is busy at the moment

Discrete Event Simulation (DES) [2, 6]

- Simulation is the imitation of the operation of a real-world process or system over time
- A *discrete event* occurs at an instant in time and marks a change of state in the system
- DES represents the operation of a system as a chronological sequence of events
- This technique allows to analyze complex systems, even before they are actually built (*what-if analysis*)
- If the simulation is run on top of a parallel/distributed system, it's named Parallel Discrete Event Simulation (PDES) [3]

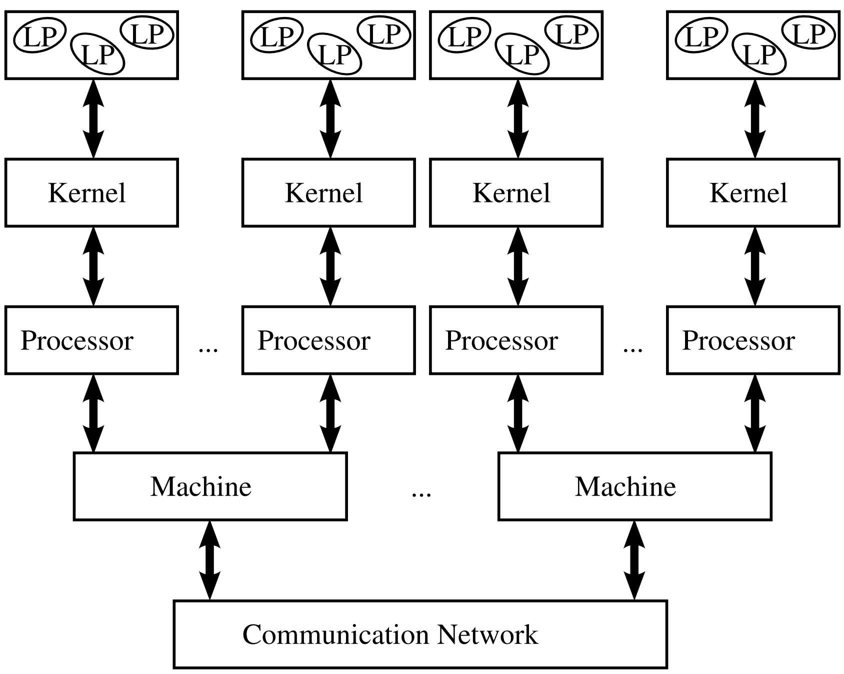
DES Building Blocks

- **Clock**
  - Independently of the measuring unit, the simulation must keep track of the current simulation time
  - Being *discrete*, time hops to the next event's time
- **Events List**
  - At least the *pending event set* must be maintained by the simulation architecture
  - Events can arrive at a higher rate than they can be processed
- **Random-Number Generators**
  - Simulation often rely on distributions, in order to model real world's aspects
- **Statistics**
- **Ending Condition**
  - Real systems can often run forever, so the designer of the model must decide when the simulation will halt

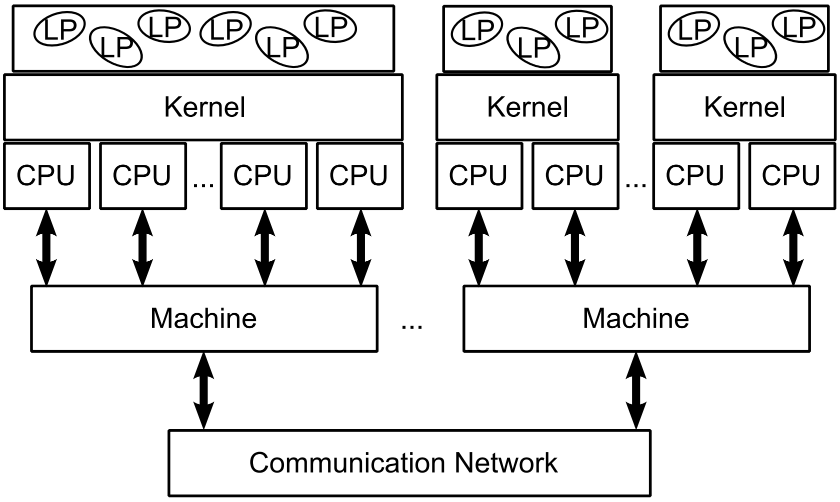
DES Skeleton

```
procedure INIT
    End ← false
    initialize State, Clock
    schedule INIT
end procedure

procedure SIMULATION-LOOP
    while End == false do
        Clock ← next event's time
        process next event
        Update Statistics
    end while
end procedure
```


PDES Logical Architecture



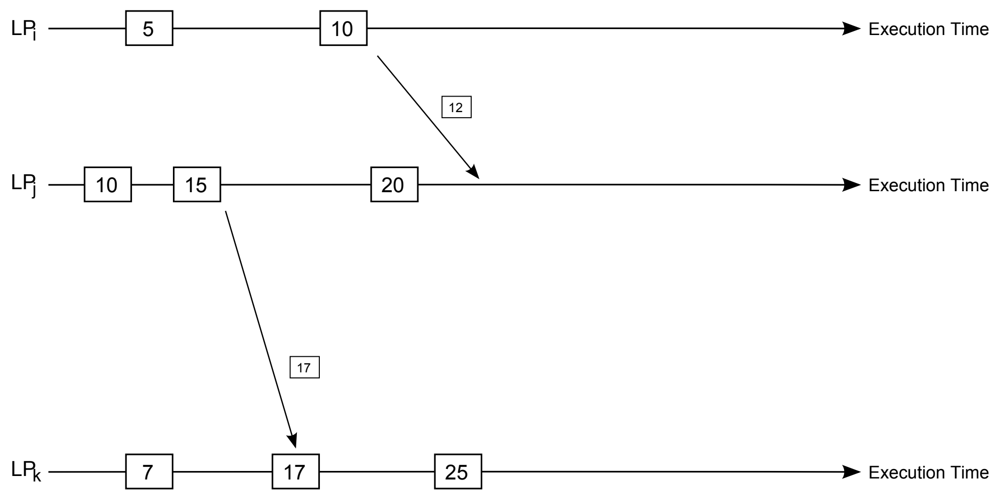
PDES Modern Architecture



The Synchronization Problem

- Consider a simulation program composed of several *logical processes* exchangin timestamped messages
- Consider the *sequential* execution: this ensures that events are processed in timestamp order
- Consider the *parallel* execution: the greatest opportunity arises from processing events from different LPs concurrently on different processors
- Is *correctness* always ensured?

The Synchronization Problem



This is called Causality Violation

Conservative Synchronization: *Lookahead*

- Consider the LP with the *smallest* clock value at some instant T in the simulation's execution
- This LP could generate events relevant to every other LP in the simulation wiith a timestamp T
- No LP can process any event with timestamp larger than T

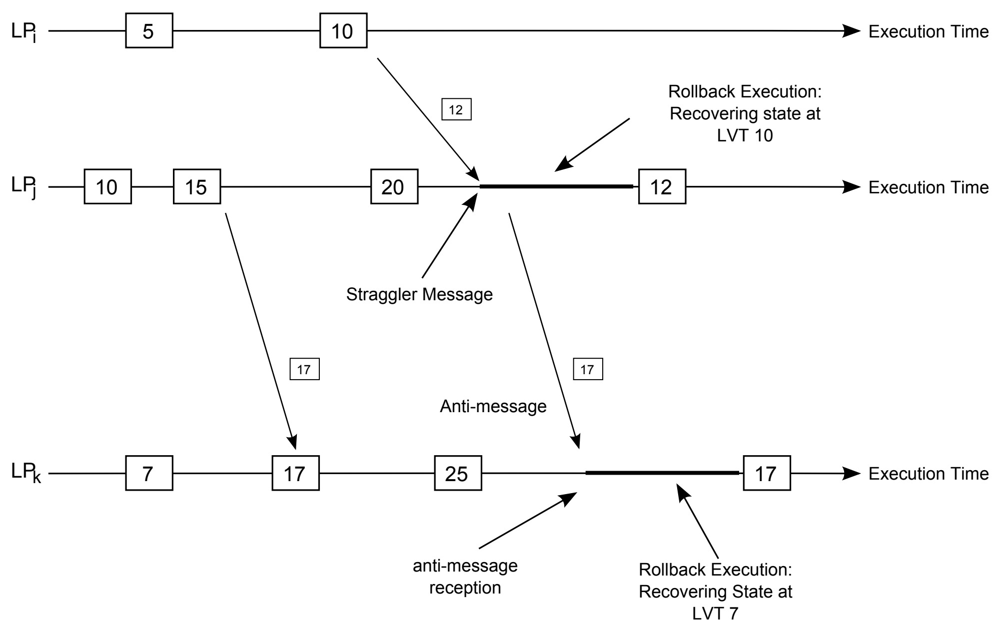
- If each LP has a *lookahead* of L, then any new message sent by an LP must have a timestamp of at least $T + L$
- Any event in the interval $[T, T + L]$ can be safely processed
- L is intimately related to details of the simulation model

Optimistic Synchronization: Time Warp [4]

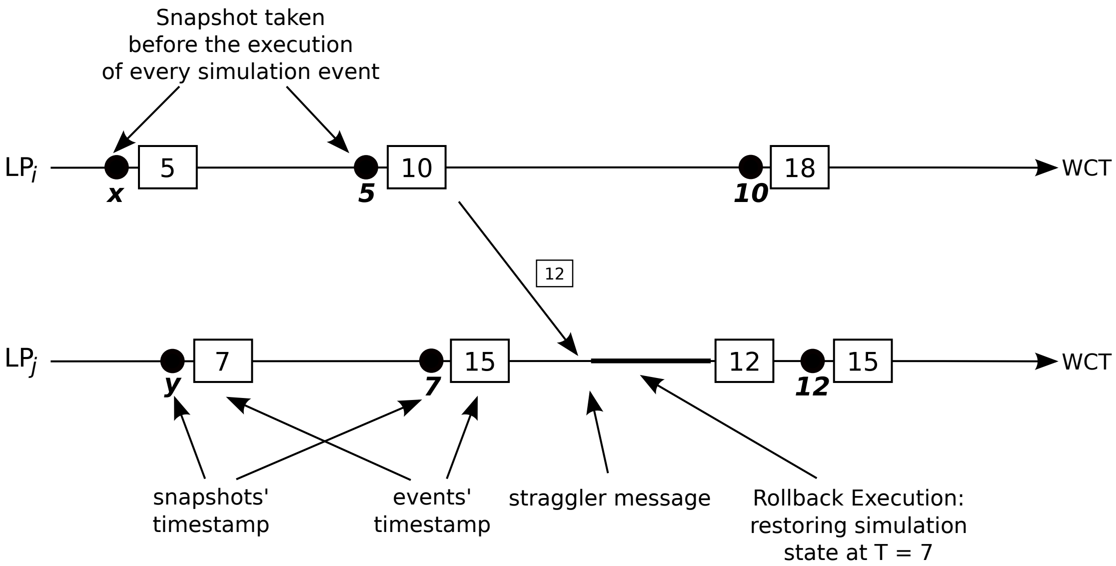
- There are no state variables that are shared between LPs
- Communications are assumed to be reliable
- LPs need not to send messages in timestamp order

- **Local Control Mechanism**
  - Events not yet processed are stored in an *input queue*
  - Events already processed are not discarded
- **Global Control Mechanism**
  - Event processing can be **undone**
  - A-posteriori detection of causality violation

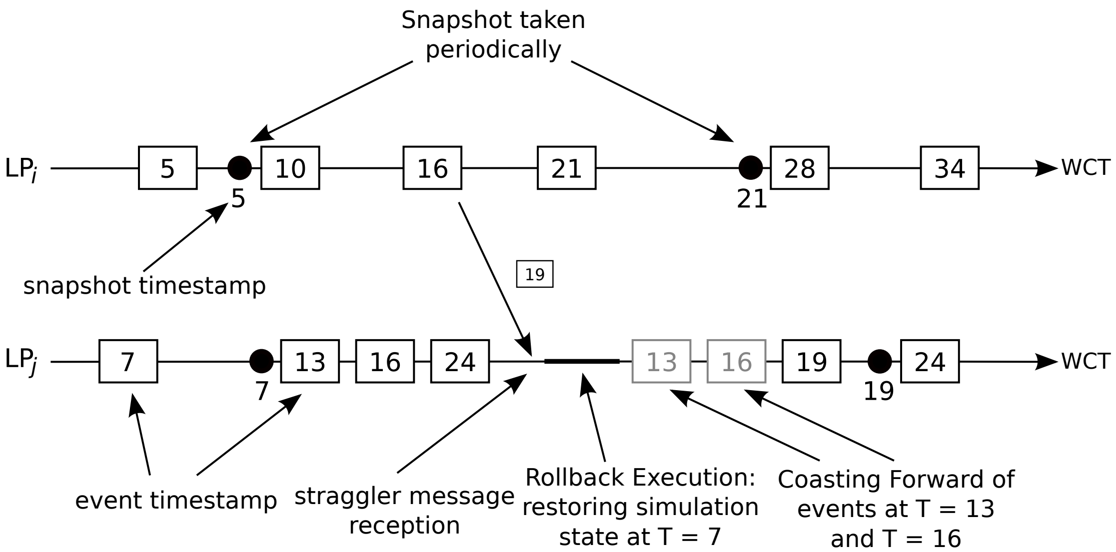
Rollback Operations



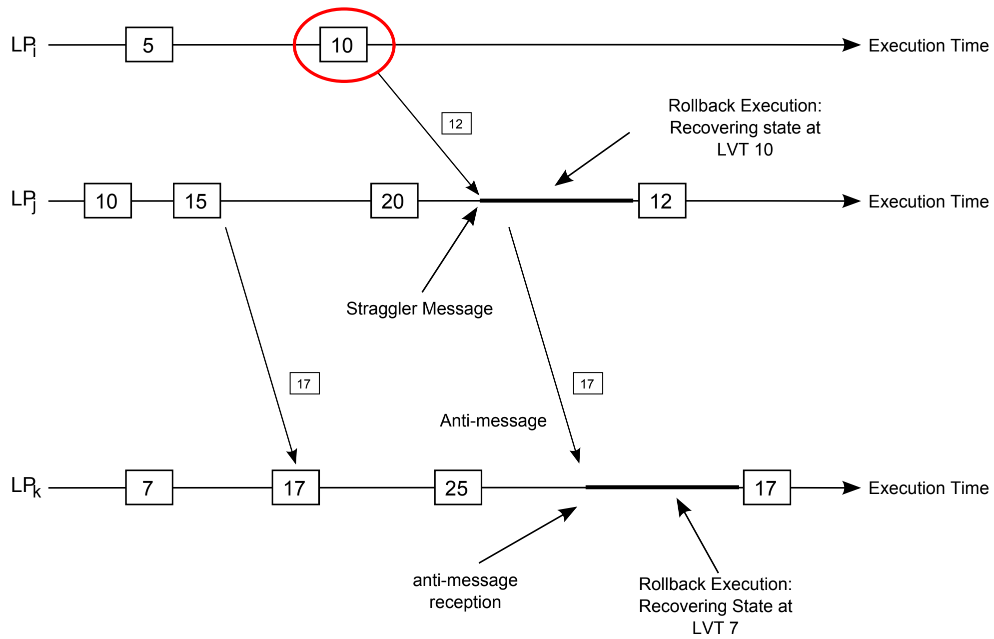
Copy State Saving



Sparse State Saving



Global Virtual Time



Reverse Computation

- It can reduce state saving overheads
- Each event is associated (manually or automatically) with a *reverse event*
- A majority of the operations that modify the state variables are *constructive* in nature
  - the undo operation for such operations requires no history
- *Destructive* operations (assignments, bit-wise computation, . . .) can only be restored via traditional state saving

Reversible Operations

Type	Description	Application Code			Bit Requirements		
		Original	Translated	Reverse	Self	Child	Total
T0	simple choice	if() s1	if() {s1; b=1;}	if(b==1){inv(s1);}	1	x1,	1+
		else s2	else {s2; b=0;}	else{inv(s2);}		x2	max(x1,x2)
T1	compound choice	if () s1;	if() {s1; b=1;}	if(b==1) {inv(s1);}	lg(n)	x1,	lg(n) +
	(n-way)	elseif() s2;	elseif() {s2; b=2;}	elseif(b==2) {inv(s2);}		x2,	max(x1....xn)
		elseif() s3;	elseif() {s3; b=3;}	elseif(b==3) {inv(s3);}		....,	
		else() sn;	else {sn; b=n;}	else {inv(sn);}		xn	
T2	fixed iterations (n)	for(n)s;	for(n) s;	for(n) inv(s);	0	x	n*x
T3	variable iterations	while() s;	b=0;	for(b) inv(s);	lg(n)	x	lg(n) +n*x
	(maximum n)		while() {s; b++;}				
T4	function call	foo();	foo();	inv(foo)();	0	x	x
T5	constructive	v@ = w;	v@ = w;	v = @w;	0	0	0
	assignment						
T6	k-byte destructive	v = w;	{b =v; v = w;}	v = b;	8k	0	8k
	assignment						
T7	sequence	s1;	s1;	inv(sn);	0	x1+	x1+...+xn
		s2;	s2;	inv(s2);		....+	
		sn;	sn;	inv(s1);		xn	
T8	Nesting of T0-T7	Recursively apply the above			Recursively apply the above		

if/then/else

```
if(qlen > 0) {
    qlen--;
    sent++;
}
```

```
if(qlen "was" > 0) {
    sent--;
    qlen++;
}
```

- the reverse event must check an “old” state variables’ value, which is not available when processing it!

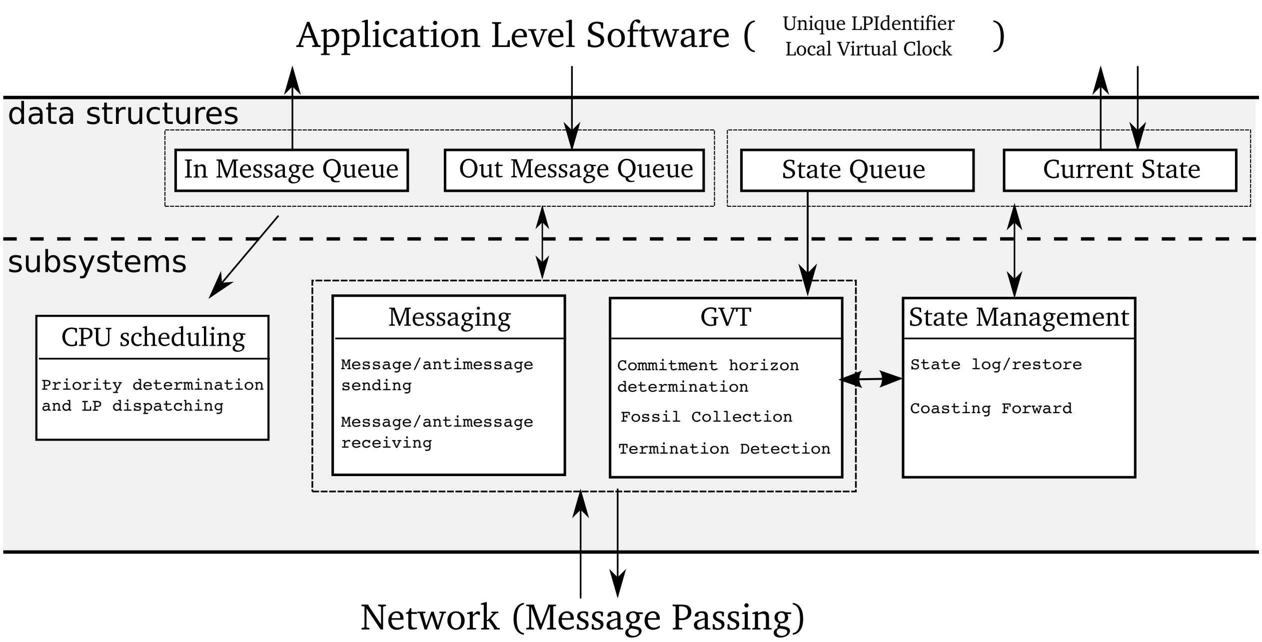
if/then/else

- Regular events are modified by inserting “bit variables”
- They are transparently-added state variables telling whether a particular branch was taken or not during the forward execution

```
if(qlen > 0) {
    b = 1;
    qlen--;
    sent++;
}
```

```
if(b == 1) {
    sent--;
    qlen++;
}
```


Time Warp Fundamentals

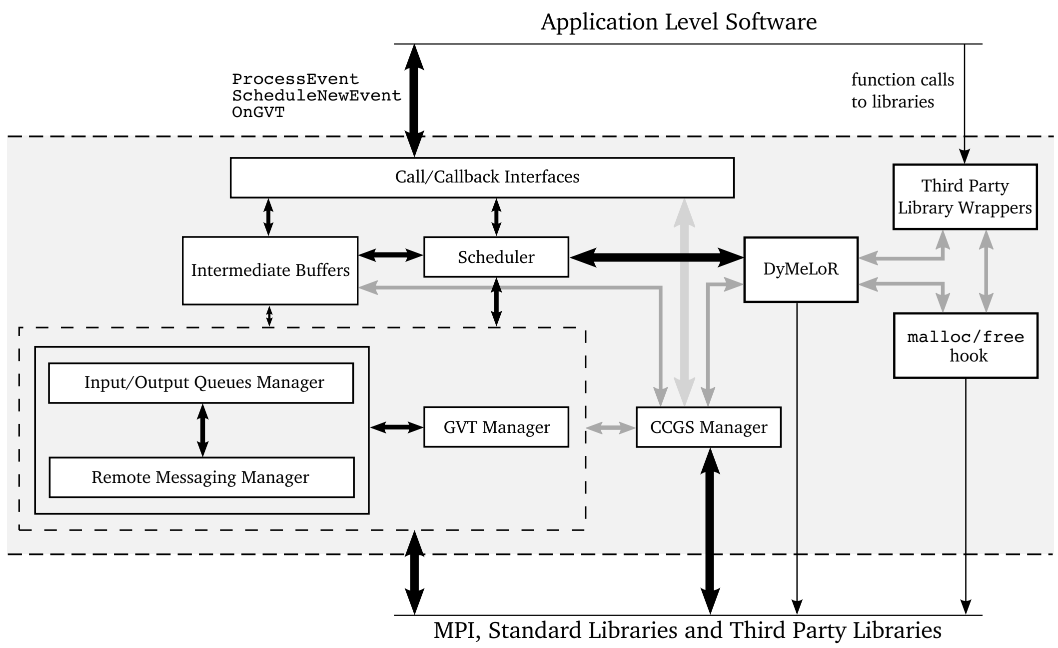


The ROme OpTimistic Simulator (ROOT-Sim) [5]

- Simulation Platform built according to the Time Warp Synchronization Protocol
- Supports ANSI-C programming
- Simulation state is scattered around dynamically allocated memory
- Supports Full, Incremental and Autonomic Logging

- http://www.dis.uniroma1.it/∼hpdcs/ROOT-Sim/

ROOT-Sim's Architecture



An Example Simulation Model: Data Definition

```
#include <ROOT-Sim.h>

#define INIT 0

#define PACKET 1
#define PACKETS 1000000
#define DELAY 1.5

typedef struct event_content_t {
   time_type send_time;
} event_content_t;

typedef struct lp_state_t {
   int pckt_count;
} lp_state_t;
```


An Example Simulation Model: Events Processing

```
void ProcessEvent(int me, time_type now, int event_t,
    event_content_t *event content, unsigned int size, void *ptr) {
   event_content_t new_evt;
   lp_state_t *pointer = (lp_state_t *)ptr;
   time_type ts;
   int r;

   switch(event_type) {
      case INIT:
         pointer = (lp_state_t *)malloc(sizeof(lp_state_t));
         pointer->pckt_count = 0;

         ts = (time_type)(20 * Random());
         ScheduleNewEvent(me, ts, PACKET, NULL, 0);

      break;
```

An Example Simulation Model: Events Processing (2)

```
    case PACKET:
          pointer->pckt_count++;
          new_evt.sent_at = now;
          r = n_prc_tot * Random();
          ts = now + Expent(DELAY);
          ScheduleNewEvent(r, ts, PACKET, &new_evt, sizeof(new_evt));

    }
}
bool OnGVT(lp_state_t *snapshot, int gid) {
    if(snapshot->pckt_count < PACKETS)
       return false;
    return true;
}
```


Bibliography

[1] Message Passing Interface: http://www.mpi-forum.org/.

[2] J. Banks, J. Carson, B. L. Nelson, and D. Nicol.
Discrete-Event System Simulation (4th Edition).
Prentice Hall, 4 edition, Dec. 2004.

[3] R. M. Fujimoto.
Parallel discrete event simulation.
In *WSC '89: Proceedings of the 21st conference on Winter simulation*, pages 19–28. ACM Press, 1989.

Bibliography (2)

[4] D. R. Jefferson.
Virtual Time.
ACM Transactions on Programming Languages and System, 7(3):404–425, July 1985.

[5] A. Pellegrini, R. Vitali, and F. Quaglia.
The ROme OpTimistic Simulator: Core internals and programming model.
Proceedings of the 4th ICST Conference of Simulation Tools and Techniques (SIMUTools), 0, 2011.

[6] S. Robinson.
Simulation: The Practice of Model Development and Use.
John Wiley & Sons, 2004.