Discrete Event Simulation in PARSEC

Richard A. Meyer
meyerr@cs.ucla.edu

Including slides by Rajive Bagrodia and Mineo Takai

Outline

- Parsec Basics
- Building a Simulation

PARSEC Basics

- LP modeled as entity
  - light-weight thread
  - encapsulated data
  - all activity follows message receipt
- Events modeled as message communication
  - e(t,p,a): send message *m(e)* at time *t* to LP *p*
  - on receiving *m(e)*, *p* executes actions *a*
    - change its state (local variables)
    - schedule events(messages) at *time* >=*t*
    - unschedule previously scheduled events (e.g. *timeout*) ; only *future* (conditional) events may be unscheduled

Summary of Parsec Constructs

- Constructs
  - Entities -- classes and instances
  - Messages
- Simple Types
  - **ename**
  - **clocktype**
- Concepts
  - Buffered communication
  - Selective receive
  - Conditional/Unconditional events

Entities

- Entity definition: similar to C function or C++ class

  <message declarations>

  entity <EntityClass> (<parameter list>) [stacksize (<size>)]

  { <variable declarations>

  <entity body>

  [<finalize-stmt>]

  }

- Finalize Statement -- used at termination
  - may not contain message operations

Entity Example

```
entity Manager (int maxResources) stacksize (20000) {
   int unitsAvailable = maxResources;
   int totalRequests = 0;
   …
   finalize {
     printf (“Manager got %d total requests.\n”,
              totalRequests);
   }
}
```

Stacksize

- Each entity instance is a light-weight thread that runs in its own stack space
- The stack must be large enough to hold
  - the entity's hidden state data, parameters, and local variables
  - the space required by any functions called by the entity
- Default size is 200K, minimum is 20K
- Many OS's limit stacksize
  - the *unlimit* shell command removes this limit on most unix systems (not on linux)

Entity Creation

- Ename: entity identifier

  ename s1, s2;
- Instantiation

  s1 = new *Manager* (5);

  s2 = new *Manager* (10) at 2; /* remote creation */
- self: self-reference

```
entity Server (ename creator) {
   …
}
. . .
s2 = new Server (self);
```

Parameter Passing

- All parameters, including arrays, are passed by value (to enforce encapsulation)
  - Array parameters must have fixed size.

```
entity Manager(char filename[20]) {
  …
}
char filename[20];
ename e;
e = new Manager(filename);
```


Messages

- Entities communicate via typed messages
- Message declaration similar to a C *struct*
  - message *Data* {int value; ename *sender*;};
    message *Ack* {};

```
entity node (int node_no) {
   int num_pkts ;
   message Data  data;        /* declaration */

   num_pkts = data.value;    /* referencing data */
```

Sending Messages

- Asynchronous buffered communication
- Message time-stamp (T)
  - default: current simulation clock (T = simclock())
  - user may specify future values (T = simclock() + t_s)
- Message placed in destination message buffer at T
- Transmission time may be modeled by
  - the sender by specifying an appropriate 'receive time' for the message
  - the receiver
  - using a separate entity (or entities) to model the channel
    - bus-bases, point-point, wireless (with & without interference effects)

Send Statement

send *message_type{params}* to *ename* [after *t*];
send *message_variable* to *ename* [after *t*];

Examples:
message *Request* {ename *requester* ;} *oldreq* ;
message *Release* {} *release*;
message *Done* {} *done*;

send *Request* {self} to *s1* ;
send *release* to *s1* ;
send *done* to *oldreq.requester* after (10);

Receiving Messages

- Messages received in timestamp order
- Messages of same type with same timestamp are received in order from same source, non-deterministically from different sources
- Messages of different types with same timestamp are received in non-deterministic order, even if from same source
- When a message is received, the local clock is updated to max(simclock(), message-timestamp)

Receive Statement

receive (m_1 *msg_var*) [when b_1] {
    statements;
}
or receive (m_2 *msg_var*) [when b_2] {
    statements;
} …
or timeout in/after (*exp*) {
    statements;
}

where m_i is a message type, b_i is a boolean expression (guard), and *exp* is a clocktype expression

Sample Receives

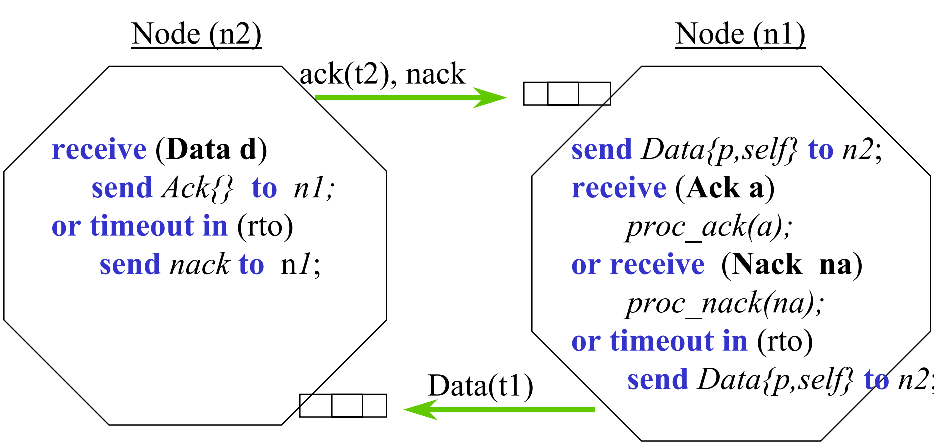
```
message InitialData {}; message Data {int value;}

receive (InitialData init) {
    initialize(init);
}

receive (Data data)
    processData(data);
or receive (Interrupt interrupt) { … }

receive (Data data)
    processData(data);
or timeout in (CannotWaitAnyLonger) { … }
```


PARSEC Code Sample



Guards

receive (m_t *msg_var*) [when b_1] { statement;}

- Conditional receive based on local data and messages
- Guard functions:
  - qempty(Message-type)
  - qlength(Message-type)
  - qhead(Message-type)

receive (*Data data*) when qempty(*Ack*) { … }
or receive (*Ack ack*) { … }

Resource Manager - unit requests

```
message Request {ename requester;};
message Release {};
message Done {};
entity Manager (int maxResources) {
   int units = maxResources;
   while (TRUE) {
      receive (Request request) when (units > 0) {
         units--;
         send Done{} to request.requester;
      }
      or receive (Release release)
         units++;
   }
}
```

Resource Manager - block requests

```
message Request {ename requester; int num;};
message Release {int num;}; message Done {};
entity Manager (int max_printers) {
   int units = max_printers;
   while (TRUE) {
      receive (Request request)
         when (request.num <= units) {
            units -= request.num;
            send Done{} to request.requester;
      }
      or receive (Release release)
            units+= release.num;
   }
}
```


Using Guards

Guards can enforce different queuing disciplines:

message *Request* { ename *requester*; int *count*;};

- First Fit

  receive (*Request r*) when (*r.count* <= *units*)

- FIFO (first in, first out): use function qhead(msg)

  receive (*Request r*) when ((qhead*(r)).count* <= *units*)

- Priority: use function qlength(msg) or qempty(msg)

  receive (*Request r*) when (qempty*(Release*) && *r.count* <= *units*)

Program Structure

- Library entities/separate compilation

  extern entity *entity_type(parameter_list);*

- Driver entity
  - similar to main() function in C programs
  - processes parameters
  - creates and initializes entity instances
  - sets simulation duration

```
entity driver (int argc, char** argv) {

   ename manager;

   manager = new Manager (atoi(argv[1]));

}
```


Delay Entities: Ping-Pong

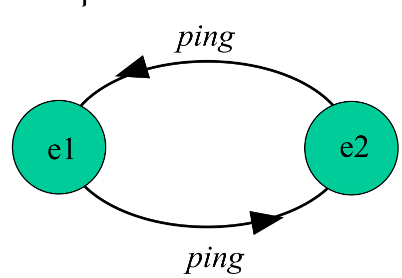
```
message Init { ename id;};
message Ping { int originator;  int trips;};

entity Delay(int myno, int mean_delay) {
   ename  next;
   message Ping ping;

   receive (Init i)  next = i.id;
   send Ping{myno,0}  to next;

   while (1)
      receive (Ping temp)  {
         ping = temp;
         if (ping.originator == myno)
            ping.trips++;
         hold(exp(mean_delay));
         send ping to next;
      }
 }
```

```
entity driver() {
   ename e1, e2;
   setmaxclock(500);
   e1 = new Delay(1,10);
   e2 = new Delay(2,10);
   send Init{e2} to e1;
   send Init{e1} to e2;
}
```



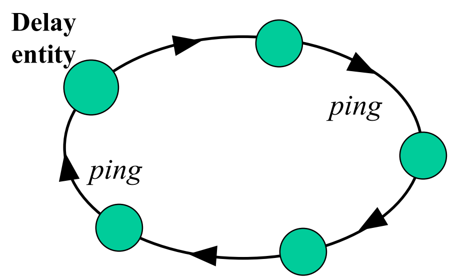
Ring of Delay Entities

```
entity driver() {
   ename prev;
   ename next;
   ename first;

   setmaxclock(500);
   first = new Delay(0,10);
   prev  = first;

   for (i = 0; i < 5; i++) {
     next = new Delay(i,10);
     send Init{next} to prev;
     prev = next;
   }
   send Init{first} to prev;
 }
```

- Extend previous example to ring of 5 delay entities; only driver needs to be changed.



Compiling and Running

> pcc -o sim simulation.pc

> pcc -c utilities.c

> pcc -c delay.pc

> pcc -o sim driver.pc delay.o utilities.o

> sim

Building a Simulation in Parsec

- Time Management
- Event Management
- Topology Setup

Time Management

- Each entity has an internal clock
  - advances to timestamp of message
- clocktype -- integral type - either
  - unsigned long, or
  - long long (specified with -clock longlong)
- Clock Functions
  - simclock(): returns current simulation time
  - setmaxclock(clocktype t): sets simulation duration to t
  - **hold(clocktype t)**: advances an entity instance's clock by t
  - ctoa() and atoc() for converting to/from string

Events

- *Definite* event: An event that must occur once it is scheduled; e.g. job departure from FIFO server.
  - hold statement: suspend process for a fixed interval

```
receive (Job j1) {
    hold (tc);
    send Job {mean} to next ;
}
```

- *Conditional* event: An event that is not definite; e.g.: retransmission of a message.
  - receive statement with timeout option

```
receive (Ack a1) {<process-ack-message>}
or timeout in (rto) {< retransmit>}
```


Events in Parsec

- Each message can be two events in Parsec
  - an arrival event
    - the message is placed in the input buffer as of the time of the message's timestamp
    - at this time, the message may affect the values of guards
  - an acceptance event
    - the message is selected by a receive statement
- Side effect
  - messages may arrive in the past

Example of past message

```
message Data {clocktype processingTime;};
Entity e1:  send Data {10} to e2 after 1;
            send Data {10} to e2 after 2;

Entity e2:  while (true) {
               receive (Data data) { /* received at time 1 */
                 hold (data.processingTime); /* now it's time 11 */
               }
            }
```


Timeout First versus Timeout Last

```
send Message{} to self after 10;

receive (Message m) { … }
or timeout in (10);               /* timeout has precedence */

receive (Message m) { … }       /* message has precedence */
or timeout after (10);
```

Nonblocking Receive

- By default, receive is blocking
  - receive (Data data) { … }

- To check message buffers without blocking

  - receive (Data data) { … }
  - or timeout after (0);

Hold versus Send After

- Two ways of modeling the passage of time
  - hold()
    - models a processing delay
    - advances the clock of the entity
  - send … after
    - models a transmission delay
    - does not advance the entity's clock

```
send Message{} to e after 10; /* my local clock is unchanged */

hold(10);  /* increases my local clock by 10 */
send Message{} to e;
```

Timeout versus Message to self

- Timeout
  - conditional event

    ```
    receive (Interrupt i) {} or timeout in (NotInterrupted) {}
    ```
  - catch and cancel

    ```
    while (true) { receive (Message m) {/*drop message*/
                   or timeout in (UnblockedAgain) break; }
    ```
- Message to self
  - periodic events

    ```
    nextPeriod = (2 Hours);
    send PeriodicEvent{} to self after (nextPeriod - simclock());
    ```
  - multiple timeouts (not supported directly)
    - implement in user code with a sorted list

Typical Entity Structure

```
message InitSimEntity{ename dests[N];};

entity SimEntity (ename creator, int id) {

   /* local variables */
   receive (InitSimEntity init) { /* initialize */ }

   while (true) {
       receive (M1 m) { /* do something */}
       or receive (M2 m) { /* do something else */ }
   }
   finalize {
       /* print accumulated data */
   }
}
```

Typical Driver Entity

```
entity driver (int argc, char** argv) { /* must use char** */
   /* check parameters, read input, etc. */

   /* set simulation duration */
   setmaxclock(50 Seconds);

   /* create entities */
   server = new Server(self, 0);
   client = new Client(self, 0);

   /* initialize entities */
   send InitServer{client} to server;
   send InitClient{server} to client;
}
```