Programming and Optimizing Distributed Algorithms

(work in progress)

Y. Annie Liu

Computer Science Department
State University of New York at Stony Brook

The slides have some informal parts and lack some explanations and alternatives given in the presentation. Please feel free to ask if you have questions: liu@cs.sunysb.edu

Age of distributed programming

search engines

social networks

cloud computing

mobile computing

...

Programming algorithms

languages: ... ALGOL ... C++ ... Java ... Python ...

- statements: assignments, conditionals, loops
- expressions: arithmetic, boolean, other data (sets)
- subroutines: functions, procedures (recursion)
- logic rules (relation and recursion), though not as much used
- objects: keep data and do operations (organization)

that's mostly all sequential and centralized.

Concurrent programming

threads: multiple threads accessing shared data

threads as concurrent objects

- is the concurrent programming model in Java
- is adopted by other languages, such as Python and C#

Java made concurrent programming easy to do routinely.

Distributed programming

sockets: C, Java, ... most widely used languages

MPI: Fortran, C++, ... for high performance computing

RPC: C, ... just about any language, Java RMI

processes: Erlang, Python multi-processing module, ...

...

Example: distributed mutual exclusion

Lamport's algorithm: developed to show logical timestamps

n processes access a shared resource, need mutex, go in CS

a process that wants to enter CS

- send requests to all
- wait for replies from all
- enter CS
- send releases to all

each process maintains a queue of requests

- ordered by logical timestamps,
- enter CS only if it is the first on the queue.
- when receiving a request, enqueue;
- when receiving a release, dequeue.

safety, liveness, fairness, efficiency

How to program it

two extremes and many in between

1: IO automata: precise, formal; low level, unclear flow

2: English: high level, clear flow; imprecise, informal

best ones I think:

- Michel Raynal's pseudo code: still informal and imprecise
- Leslie Lamport's PlusCal: 134 lines (90 lines w/o comments and empty lines)

In PlusCal (Stephan Merz, May 2010)

```
EXTENDS Naturals, Sequences
CONSTANT N, maxClock

(* N nodes execute the algorithm. Each node is represented by two processes:
   the main "site" that requests access to the critical section, and a
   "communicator" that asynchronously receives messages directed to the "site"
   and updates the data structure. Unfortunately, PlusCal does not have
   nested processes, so we have to model sites and communicators as top-level
   processes. Sites are numbered from 1 to N, communicators from N+1 to 2N.

   The constant maxClock is used to bound the state space explored by the
   model checker, see predicate ClockConstraint below.
*)

Sites == 1 .. N
Comms == N+1 .. 2*N
site(c) == c - N   (* site a communicator is acting for *)
max(x,y) == IF x < y THEN y ELSE x

(* --algorithm LamportMutex
       variables
         (* two-dimensional array of message queues: enforce FIFO order
            between any pair of processes *)
         network = [from \in Sites |-> [to \in Sites |-> << >> ]];
         (* logical clock per site, initialized to 1 *)
         clock = [s \in Sites |-> 1];
         (* queue of pending requests per process, ordered by logical clock;
     entries are records of the form [site |-> s, clk |-> c] where
     the clock value c is non-zero *)
  reqQ = [s \in Sites |-> << >>];
  (* set of processes who sent acknowledgements for own request *)
  acks = [s \in Sites |-> {}];

define
  (* check if request rq1 has higher priority than rq2 according to
     time stamp: both requests are records as they occur in reqQ *)
  beats(rq1, rq2) ==
    \/ rq1.clk < rq2.clk
    \/ rq1.clk = rq2.clk /\ rq1.site < rq2.site

  (* Compute the network obtained from net by sending message "msg" from
     site "from" to site "to".
     NB: Use a definition rather than a macro because this allows us to
         have multiple changes to the network in a single atomic step
         (rather kludgy, though).
  *)
  send(net, from, to, msg) ==
    [net EXCEPT ![from][to] = Append(@, msg)]

  (* Compute the network obtained from net by broadcasting message "msg"
     from site "from" to all sites. *)
  broadcast(net, from, msg) ==
    [net EXCEPT ![from] = [to \in Sites |-> Append(net[from][to], msg)]]
end define;

(* insert a request from site from in reqQ of site s *)
macro insertRequest(s, from, clk)
begin
  with entry = [site |-> from, clk |-> clk],
       len = Len(reqQ[s]),
       pos = CHOOSE i \in 1 .. len + 1 :
                 /\ \A j \in 1 .. i-1 : beats(reqQ[s][j], entry)
                 /\ \/ i = len + 1
                    \/ beats(entry, reqQ[s][i])
  do
    reqQ[s] := SubSeq(reqQ[s], 1, pos-1) \circ << entry >>
               \circ SubSeq(reqQ[s], pos, len);
  end with;
end macro;

(* remove a request from site from in reqQ of site s --
   assume that there is at most one such request in the queue *)
macro removeRequest(s, from)
begin
  with len = Len(reqQ[s]),
       pos = CHOOSE i \in 1 .. len : reqQ[s][i].site = from
  do
```

```
    if (reqQ[s][pos].site = from)
    then
       (* request actually exists *)
       reqQ[s] := SubSeq(reqQ[s], 1, pos-1) \circ SubSeq(reqQ[s], pos+1, len);
    end if;
  end with;
end macro;

process Site \in Sites
begin
  start:
    while TRUE
    do
  ncrit:
      skip;
  try:
      network := broadcast(network, self,
                           [kind |-> "request", clk |-> clock[self]]);
      acks[self] := {};
  enter:
      await /\ Len(reqQ[self]) > 0
            /\ Head(reqQ[self]).site = self
            /\ acks[self] = Sites;
  crit:
      skip;
  exit:
      network := broadcast(network, self, [kind |-> "free"]);
    end while;
end process;

process Comm \in Comms
begin
  comm:
    while TRUE
    do
      (* pick some sender "from" and the oldest message sent from that node *)
      with me = site(self),
           from \in {s \in Sites : Len(network[s][me]) > 0},
           msg = Head(network[from][me]),
           _net = [network EXCEPT ![from][me] = Tail(@)]
      do
        if msg.kind = "request" then
           insertRequest(me, from, msg.clk);
           clock[me] := max(clock[me], msg.clk) + 1;
           network := send(_net, me, from, [kind |-> "ack"]);
        elsif (msg.kind = "ack") then
           acks[me] := @ \union {from};
           network := _net;
        elsif (msg.kind = "free") then
           removeRequest(me, from);
```

```
                  network := _net;
               end if;
             end with;
           end while;
       end process;
     end algorithm
*)
```

The PlusCal Algorithm Language

http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#pluscal

Theoretical Aspects of Computing-ICTAC 2009, Martin Leucker and Carroll Morgan editors. LNCS 5684, page 36-60.

PlusCal (formerly called +CAL) is an algorithm language. It is meant to replace pseudo-code for writing high-level descriptions of algorithms. An algorithm written in PlusCal is translated into a TLA+ specification that can be checked with the TLC model checker. This paper describes the language and the rationale for its design.

An earlier version was rejected from POPL 2007. Based on the reviews I received and comments from Simon Peyton-Jones, I revised the paper and submitted it to TOPLAS, but it was again rejected. It may be possible to write a paper about PlusCal that would be considered publishable by the programming-language community. However, such a paper is not the one I want to write. For example, two of the three TOPLAS reviewers wanted the paper to contain a formal semantics--something that I would expect people interested in using PlusCal to find quite boring. (A formal TLA+ specification of the semantics is available on the Web.) I therefore decided to publish it as an invited paper in the ICTAC conference proceedings.

In dreamed language (Feb 2009 lecture)

```
assume fifo channel

Pi:
  ts_i: value of logical clock starting at 0 and inc by 1, initialized to 0.
  queue_i: queue for pending requests ordered by the logical timestamp, init[]

  request:
    add (request,ts_i,i) to queue_i
    send (request,ts_i,i) to all other processes

    await:
      (request,ts_i,i) where (ts_i,i) < others in queue_i, and
      having received (reply,ts_j,j) with (ts_j,j)> (ts_i,i) from all Pj

  CS

  release:
    remove request from queue_i
    send (release,ts_i,i) to all other processes

  receive (request,ts_j,j):
    add (request,ts_j,j) to queue_i
    send (reply,ts_i,i) to Pj

  receive (release,ts_j,j):
    remove (request,ts_j,j) from queue_i
```

In our new language—algorithm part

```
def setup(s):
  int self.c = 0  //logical clock, init 0
  set self.q = {} //set of pending requests, to be ordered by logic clock, init {}
  set self.s = s  //set of all processes excluding self, passed in

def cs():
  request: //to enter cs, enqueue and send request to all
    add (request, c, self) to q
    send (request, c) to s
    c_req = c //thanks to Georges Gonthier for finding an error that needs this fix

  reply:   //then wait for replies from all
    await each (request,c2,p2) in q | (c_req,self) <= (c2,p2),
          each p2 in s | some received (reply,c2,p2) | (c_req,self) < (c2,p2)

  ...      //critical section

  release: //after exiting cs, dequeue and send releases to all
    remove (request, _, self) from q
    send (release, c_req) to s

receive (request, c2, p2): //when receiving requests, enqueue and reply
  add (request, c2, p2) to q
  c = max(c,c2) + 1
  send (reply, c) to p2

receive (release, c2, p2): //when receiving releases, dequeue
  remove (request, c2, p2) from q
```

In our new language—complete program

```
class P extends Process:

  ... //content of the previous slide

  def run():

    ... //any non-cs code can call cs() to request, enter, and release cs
    cs()
    ...

def main():

  channel: fifo //configuration

  ps = newprocesses(5,P) //create 5 processes of P class

  for p in ps: p.setup(ps-{p}) //pass to each process other processes

  for p in ps: p.start() //each process then starts the run method
```

Outline

- motivation, problem, prior languages

- language: no new concepts but simple and powerful
- compilation: to executable programs
- optimization: expensive queries and message passing

- implementation and summary

name: DistPL or DistAlgo?

Language overview

1. distributed processes and sending of messages
   processes are like threads but with private memory,
   and communicate by message passing.

2. control flows and handling of received messages
   use labels where control flow can yield to
   handling of messages and resume afterwards.

3. configuration
   define channel type,
   handling of messages,
   and setup for starting processes.

Lang: distributed procs and sending msgs

process definition

```
class P extends Process:
```

defines class `P` of process objects, with private fields.

process creation

```
new P(s,...)                                   newprocesses(n,P)
```

creates a new proc of class `P` on site `s`, returns the proc.

sending messages

```
send m to p                                    send ms to ps
```

sends message `m` to process `p`.

possibly define a class for messages;
a field or first component is the kind of the message.

Lang: control flows and receiving msgs

label for statement

```
l:  stmt
```

defines program point `l` where the control flow can yield to handling of certain messages and resume afterwards

handling messages received

```
receive ms at ls:  stmt
```

allows handling of `ms` at `ls`; default is at all labels

synchronization

```
await bexp
```

awaits value of `bexp` to be true; can incl receivings of msgs

Compilation: processes and sending msgs

process definition

check that a process can access fields of only self, not other processes.

process creation

create a process that has its private memory.
physical if location/machine is specified; simulated o.w.

each process is implemented using two threads:
main thread executes the main flow of control of the process;
side thread receives and enqueues msgs sent to this process.

`send m to p`

enqueue `m` via side thread of `p`.

Compilation: control flows and recv'g msgs

label `l`

```
l:  stmt → l() //call msg handler
           stmt
def l(): //msg handler method
    for all kinds of msgs in all recv stmts whose at's incl l:
        compiled code for receive m:  stmt, described next
```

receiving message `m`

look for a msg or msgs of the kind of `m`,
dequeue if found, bind free variables if any, and
execute the stmt that follows.

synchronization

```
l:  await bexp → l(); while not bexp:  l()
await bexp → allmsglabel:  await bexp
```
`allmsglabel`: special label, can receive all kinds of msgs

Compilation: configuration

channels: fifo: one queue definitely.
  other choices: ...

handling messages: one or all, or in between.
  take care when sending and receiving messages.

processes: things about faulty or non-faulty processes. other
  setups: ...

Optimization

identify expensive computations

expensive queries and message passing:

```
await each (request,c2,p2) in q | (c_req,self) <= (c2,p2),
each p2 in s |some received (reply,c2,p2) |(c_req,self)<(c2,p2)
```

incrementalize expensive queries with respect to updates

local updates and message passing:

update to `q`, and passing request, reply, release

remove unnecessary messages

Optimization: incrementalization

```
await each (request,c2,p2) in q | (c_req,self) <= (c2,p2),
each p2 in s |some received (reply,c2,p2) |(c_req,self)<(c2,p2)
```

queries, in comprehension: using all pairs distinct

```
(c_req,self) = min{(c2,p2) for (request,c2,p2) in q}
size{p2 in s |some ...} = size(s)
```

updates:

add and delete to `q`

assign to `s` and receive replies

incrementalization:

maintain min for `q`: priority queue, heap, ...

maintain count for replies: increment until size of `s`

Optimization: removing unnecessary msgs

```
await each (request,c2,p2) in q | (c_req,self) <= (c2,p2),
each p2 in s |some received (reply,c2,p2) |(c_req,self)<(c2,p2)
```

reply from `p2` that requested before `c_req` is useless until release

delay reply, and remove release—Ricart-Agrawala's algorithm:

```
receive (request, c2, p2):
  if (c_req,self) in q and (c_req,self) < (c2,p2):
    add p2 to waiting
  else: send (reply, c) to p2
```

after critical section:

```
  send (reply, c) to waiting
  waiting = {}
```

maintain `waiting`, but
  don't need to check min: don't need queue, enq, deq
  don't need fifo! need proof!

Optimization: parallelization

separate threads for receiving msgs

  will be easy to figure out independencies after dependencies are clear.

note: this is for high-level threads,

  not the low-level threads for compilation purposes.

Implementation of Lamport's algorithm

by Bo Lin, nice code

with variations

- C using sockets: 358 lines (272 w/o comments & empty lines)
- Java using sockets: 281 lines (216)
- Python using multiprocessing module: 165 lines (122)
- Erlang: 177 lines (99)

our language: 44 lines (30 w/o empty lines)

Implementation of compilation

syntax and parsing:

  use python parser.

  modify python ASDL to take labels and receives.

compilation:

  generate python code. close to working

  may generate C, Java, and Erlang too.

  generate PlusCal spec for verification. todo

Summary and conclusion

programming distribute algorithms

clear and high-level; precise and formal

a language and compilation: simple and powerful

distributed processes and sending messages

yield points and receiving messages

configuration

optimization: powerful

incrementalizing expensive queries

removing unnecessary messages

parallelizing independent computations

other ongoing work:

incrementalizing queries over objects and sets, statically

answering queries over rules efficiently, for Datalog & ext's

an invariant-driven transformation language and system

book: Systematic Program Design: From Clarity to Efficiency