

# CS 475/575

Parallel and Distributed  
Simulation  
Spring 2005  
M. Overstreet  
Old Dominion University

---

---

---

---

---

---

---

---

## Parallel & Distributed Simulation

- ⌘ Overview only
  - Original goal is
    - Speed-up
    - Do bigger models (more memory)
- ⌘ Long history in academia (20+ years)
- ⌘ Seems straightforward; turns out not to be
- ⌘ Some success:
  - If "speed-up" is not primary objective, then other reasons:
    - Move simulation to players, not players to simulation
  - If model is too large to fit on one machine
  - Replications
    - Needed for statistical purpose: each run can produce one data point; need many data points to compute confidence interval

---

---

---

---

---

---

---

---

## What's the original problem?

- ⌘ Simulations take too long
  - We've done several network simulations in which the simulator can take several hours to simulation several seconds
- ⌘ Simulation are too big, can't fit state space into one machine.
- ⌘ People's workstations have all of these unused cycles. Should be able to harness all of this compute power.

---

---

---

---

---

---

---

---

## Simple definitions

- ⌘ Parallel simulation: use of tightly coupled processors (shared memory and backplane)
- ⌘ Distributed simulation: use of loosely coupled processes (usually involves moving data across network)

---

---

---

---

---

---

---

---

## Task 1: Decompose computation

- ⌘ Short history: academic community really likes Simula. Model represented as a group of interacting processes running in parallel.
- ⌘ So it's simple: each process runs on its own processor. Obvious!
  - ☑ Maybe some load balancing problems if many fewer physical processors than processes
- ⌘ All too frequent early result: order of magnitude slowdown.

---

---

---

---

---

---

---

---

## What went wrong?

- ⌘ Lots of things.
- ⌘ What if two processors update/ref same variable at same time? The different processes do interact, after all.
  - ☑ Critical region problem
    - ☑ Standard operating systems issue
    - ☑ Well-known solutions
      - Semaphores, etc.
  - ☑ Have to lock memory.
    - ☑ Overhead: set lock/ref memory/release lock
    - ☑ Processes get blocked waiting on locks.
- ⌘ Shared services: who manages event list?
  - ☑ It becomes bottleneck
  - ☑ Must there be only one? can each processor have its own?

---

---

---

---

---

---

---

---

## More Background - 1

### Task-parallel (partition code)

(Ignore that this is a simulation): Like any large computation, identify the set of tasks that have to be computed and assign them to separate processors (& do load balancing), e.g.,

- Event server
- Random number server
- Event servers

- Remember, processes are really collections of events that run sequentially

### Identify data sources & data sinks

- Precompute data items when possible and deliver them to consumer (so that consumer never waits)

---

---

---

---

---

---

---

---

## More Background - 2

### Space-parallel (replicated code):

System decomposed into subsystems, each assigned to one logical processor (LP). LPs communicate by passing messages.

Each LP is essentially a sequential, event-driven simulation.

LP1 sending a msg to LP2 may cause LP2 to add a new event to its event list.

Depends on model!

---

---

---

---

---

---

---

---

## More Background - 3

### Time-parallel:

- Time axis decomposed into intervals  $[t_1, t_2]$ ,  $[t_2, t_3]$ ,  $[t_3, t_4]$ , ...

- Separate processors simulate each subinterval

- Trick: how to determine the starting state for each subinterval

(Replicated runs for statistic (confidence interval estimation) is a trivial variation of above.)

---

---

---

---

---

---

---

---

## Much Work on Synchronization

Go back to use of LPs, each with its own events list.

May not be the best idea, but single events list has been a bottleneck

Some have studied the hold model with a parallel implementation

To reduce coupling, LP1 and LP2 have separate clocks. Say LP1 sends msg to LP2 (msg must have sim. time stamp). What can happen?

**Conservative** approach: no LP advances its clock until it knows it will never receive an "old" message.

**Optimistic** approach: each LP runs as fast as possible, and must "rollback" if it receives an "old" message.

---

---

---

---

---

---

---

---

## Conservative approach - 1

⌘ Credited to Chandy and Misra, '79

⌘ Basic problem: deadlock. Why?

⌘ Lots of "solutions"

☒ Null messages: I'm stuck, can't advance my clock, but I can send a message to other LPs with the time of my next event.

☒ Regular message: LPi tells LPj something LPj needs to know

☒ Null messages: LPi tells everyone something no one may be interested in

☒ What's the advantage of this?

☒ It's really a promise that I won't send a message with time stamp less than this event time (unless I have to react to someone else's message).

---

---

---

---

---

---

---

---

## Conservative app - 2

⌘ Under some mild constraints it can be shown that this avoids deadlock.

⌘ So what's the problem?

☒ Lots and lots and lots of null messages for many simulations

⌘ So some schemes (to reduce null message traffic) include a deadlock detection scheme

⌘ Some schemes use distance (of some type) between LPs so that if LP1 does something, it will take some minimal amount of sim. before it can effect me.

☒ E.g., divide the world geographically

---

---

---

---

---

---

---

---

## Optimistic approach - 1

Invented by Jefferson '82. Time Warp

Basic idea: if processor is going to be blocked anyway (the conservative approach) and processing time is almost free, why not keep going? Worst case is that processor "throws away" some computations when it receives an "old message." No harm, because the alternative is that it be idle and idle processors cost as much as busy ones.

When LP gets old msg, it runs its clock backwards to the time before it received the "bad" message (hence, the "time warp") and starts over from that time. This is called a "rollback."

Assumes dedicated processors

---

---

---

---

---

---

---

---

## What's it take to implement?

If a process is going to do this, what does it take?

It needs to "checkpoint" its state occasionally.

Used to be common back in olden times ('50's): if the mean time between failures of an OS is 5 hrs and you have a 20 hr computation, how do you ever assure you can complete the computation?)

So this concept is reasonably well understood. Sorta, anyway.

Is this enough?

Ideally want to checkpoint to memory rather than disk (why?)

Need some garbage collection scheme: after all LPs have advanced their clocks beyond t1, will anyone need to rollback to t1?

---

---

---

---

---

---

---

---

## How does LP handle a new message?

⌘ Each message must have a simulation time and then some action.

⌘ If msg time > my clock

☑ Just post to my events list and process it when my clock advances to the msg time.

⌘ But if not, ...

---

---

---

---

---

---

---

---

## TW: anti-messages

What about the messages with a time stamp back in the past?

1. Restore state to closest time before the msg time (use the last checkpoint taken before msg timestamp).
2. Then start a new computational trajectory.

What about the messages sent during the rollback period?

On the second time through, may not send the time stamp 100 msg or it may send it again.

Hence, must be some way to "take back" messages already sent. Hence anti-messages.

---

---

---

---

---

---

---

---

## Anti-messages

What does the receiver do with an anti-message?

Maybe it has not yet processed the org. message, which means that the org. message had a time stamp greater than that LP's clock. So it just cancels the org. message.

But if it's processed the msg, then what?

Rollback. Restore state. More anti-messages may need to be sent, one for each msg since that checkpoint.

And if I may have to cancel old messages, and in the general case, I don't know when I might need to, I have to keep a copy of each so I know what to cancel -- and so the receiver can uniquely identify each message.

This is getting a lot messier with a lot more overhead than I first expected.

Maybe I should only send messages to the LPs whose actions might be influenced by the msg rather than broadcasting it to all LPs.

---

---

---

---

---

---

---

---

## Does TW work?

If few rollbacks, then yes.

How can you tell beforehand if a model will have lots of rollbacks?

Depends on model. Maybe it just won't work for some models. Some computational tasks are just serial; can't be avoided.

It's frustrating to code because performance really depends on frequency of rollbacks.

Poor performance may be due to my really stupid implementation.

If I were really clever, I could find a way that generates very few rollbacks.

But maybe not. Could be the system interacts in such a way that excessive rollbacks are inevitable.

How do you know which of the two are true?

---

---

---

---

---

---

---

---

## TW Benefits

- ⌘ Stimulated lots of research. Just seems like such a good idea that it ought to work.
- ⌘ Got lots of people tenure.
- ⌘ Got lots of grad students through grad school.
- ⌘ Caused creation of new operating systems
  - ☑ UNIX just too inefficient and gets in the way, so need something that handles msg passing efficiently, supports checkpointing, garbage collection, ...
  - ☑ Hence TWOS. You can get a free copy. (Consider: if it's free, there's a reason)

---

---

---

---

---

---

---

---

## Time Warp Environment

- ⌘ Environment manages sending msgs, anti-msgs, checkpoints, rollbacks, garbage collection (throw away checkpoints when no longer needed; throw away sent msgs when no longer needed), etc.
  - ☑ Based on clock time of slowest LP (why?)
  - ☑ Environment tries to keep as much in memory as feasible since I/O ops to disk take orders of magnitude more time to complete than memory refs.
- ⌘ Modeler focus on getting the model right! This is hard!
- ⌘ At least ideally!
  - ☑ If you need really fast code, you need to understand how the way you've put your model together will effect msgs, anti-msgs, rollbacks, garbage collection, etc.
  - ☑ Remember, you picked Time Warp since "normal" sequential execution was (assumed to be) too slow!

---

---

---

---

---

---

---

---

## Time Warp Disadvantages

- ⌘ Often frustrating to implement models in a Time Warp
  - ☑ Always the nagging feeling that if I were really clever, I could figure out an implementation of my model that causes few rollbacks
- ⌘ Creating efficient implementations time consuming
  - ☑ Unless the model is going to be run a lot, it may not be worth the extra development costs
  - ☑ A slight change in model may really mess up a balanced time warp implementation

---

---

---

---

---

---

---

---

## Case Study: OneSAF

- ⌘ Simulation objectives:
  - ☑ Train decision makers
- ⌘ More effective war-gaming
  - ☑ Make more realistic
  - ☑ Reduce costs
    - ☑ Reduce staff requirements
  - ☑ Reduce risks
- ⌘ Extension of previous simulation efforts
  - ☑ Distributed Interactive Simulation (DIS)
- ⌘ Have done work with a stripped down previous version:  
about 600,000 loc.

---

---

---

---

---

---

---

---

## Concept

- ⌘ Run simulation on hundreds of networked machines (primarily laptops running Linux)
- ⌘ Simulation consists of thousands of “semi-autonomous” entities:
  - ☑ Tanks, ships, aircraft, missiles, “dismounted infantry”  
all possible interacting
- ⌘ Each machine responsible for simulating the behavior of several entities.
  - ☑ How many depends on cpu load of each entity

---

---

---

---

---

---

---

---

## Additional requirements

- ⌘ Simulation typically runs ~12 hours a day for a week.
- ⌘ Participants (and PCs) located, for example, San Diego to England
- ⌘ High band-width requirement so network “built” for simulation, then torn down
- ⌘ Loss of a few machines (or a network connection) should not stop simulation

---

---

---

---

---

---

---

---

## More details

- ⌘ Many events scripted (rather than random)
  - ☒ To meet training objectives, e.g.
    - ☒ Bad weather makes roads impassible, air support difficult
    - ☒ Many enemy actions and time of their occurrences preplanned
      - Hence semi-autonomous
- ⌘ Typically decision maker has limited resources & assigned objectives
- ⌘ Each entity (or group of entities) is tasked by decision maker
  - ☒ E.g. pilots/squadrons given "air tasking order"
- ⌘ Goal is to have trainee make the same kind of decisions as would be made "for real", then automate (through simulation) the carrying-out of those orders.

---

---

---

---

---

---

---

---

## Basic idea - 1

- ⌘ Each entity is assigned a task to complete
  - ☒ Take that hill; defend that bridge, etc.
  - ☒ Or, at a lower level: Drive from here to there avoiding rivers, taking advantage of whatever cover is available, scanning for enemy tanks. Do this in a "doctrinally correct fashion"
    - ☒ Respond to enemy based on rules of engagement

---

---

---

---

---

---

---

---

## Basic Idea - 2

- ⌘ Each entity has a "physical model" consisting of components
  - ☒ Sensors with specified capabilities, e.g.
    - ☒ Radio (so if within range, can be contacted)
    - ☒ Radar
    - ☒ Visual
  - ☒ Weaponry
  - ☒ Other physical characteristics (depends on entity type)
    - ☒ Aircraft
      - Speed
      - Full consumption table (based on alt. & speed)
      - Role rate, turn rate

---

---

---

---

---

---

---

---

## Implementation approach

- ⌘ Basically a fixed-time step simulation
  - ☒ But with some “fast-forward” capabilities
    - ☒ Have heard war described as “long periods of intolerable boredom interspersed with brief instances of total terror”
  - ☒ Replay capability
    - ☒ For “after action review”

---

---

---

---

---

---

---

---

## Entity communication

- ⌘ Entities communicate by sending a data packet at a fixed rate:
  - ☒ Their type (tank, e.g.)
  - ☒ Their location (ground truth)
  - ☒ Their velocity
- ⌘ Thus another entity, by running its simulation of its sensors, determines if it detects any other entities
  - ☒ Line of sight issues
  - ☒ Resolution, range of radar
  - ☒ Detect radio message
  - ☒ It's responsible for inserting “cloud of war” confusion
- ⌘ Concern about “level playing field” issues

---

---

---

---

---

---

---

---

## Problem 1: too many data packets

- ⌘ “More entities make training more realistic hence more effective”
  - ☒ Implies never enough CPU power!
- ⌘ Network easily swamped if everybody tells everybody else their position, velocity, etc.
  - ☒ Some data indicated CPUs could spend 60% of their time deciding that received packets had no effect on the entities it was simulating

---

---

---

---

---

---

---

---

## Some Solutions

- ⌘ Each machine publishes what it can sense (frequency, range)
- ⌘ Each machine publishes what it can emit (frequency, range)
- ⌘ So each subnetwork has a "gate-keeper"
  - ☒ Some packets never leave the subnetwork if no entity on another network can detect them
  - ☒ Likewise, some packets are never admitted to the subnetwork
- ⌘ Reliance on "dead reckoning"
- ⌘ Entities may be assigned to processors and networks "geographically"

---

---

---

---

---

---

---

---

## Problem 2: reliability

- ⌘ Over the course of a week, it's likely that some machines and some subnetworks will fail
- ⌘ Thus every machine can simulate every entity
  - ☒ Each is running exactly the same code, but is assigned only some entities to simulate
  - ☒ Each machine keeps track of the entities its heard from
    - ☒ If it doesn't hear from an entity for a while, and if it thinks its the most lightly loaded machine, it starts simulating that entity
      - Thus every machine must have some idea about the state of all other entities – and the load of every other machine

---

---

---

---

---

---

---

---

## Corrections

- ⌘ In a large simulation with thousands of entities, responsibilities get assigned to different geographic locations
  - ☒ Particular types of entities will be simulated by a particular group of machines
    - ☒ Each of those machines can simulate any of the assigned entities while other machine groups may not have the code necessary code
    - ☒ These machines can "pick-up" entities from failed machines in their group

---

---

---

---

---

---

---

---

## Enough detail?

- ⌘ Early work done for U.S. Army
  - ☑ Like to drive tanks around
  - ☑ Getting autonomous tanks to behave realistically (this is not the same as "correctly") requires detailed information about the terrain.
    - ☑ Each tank does its own route planning based on terrain data
  - ☑ Hence terrain data bases must include info
    - ☑ Line of sight (effects visual contact, radio range, shooting)
    - ☑ Tree cover
    - ☑ Bridges, roads ( | --- problem)
    - ☑ Soil type, weather effects? slope
    - ☑ Terrain can change substantially during battle
  - ☑ In some simulations, most CPU time consumed by tanks checking terrain
    - ☑ I need to drive ahead 6 inches. Let's see, is there a rock/tree/duck/land-mine in my way. Yes! so I need to find a route around the thing

---

---

---

---

---

---

---

---

## Final Exam Details

- ⌘ No more assignments!
- ⌘ Final exam comprehensive!
- ⌘ Will have take-home and in-class components
  - ☑ Take-home will assume access to Arena
- ⌘ If you do better on the final than the midterm, midterm grade will be replaced by final grade

---

---

---

---

---

---

---

---