POSE

Parallel Object-oriented Simulation Environment

An Overview

Terry Wilmarth
Parallel Programming Laboratory
University of Illinois at Urbana-Champaign
September 1, 2001

POSE

Outline of Talk

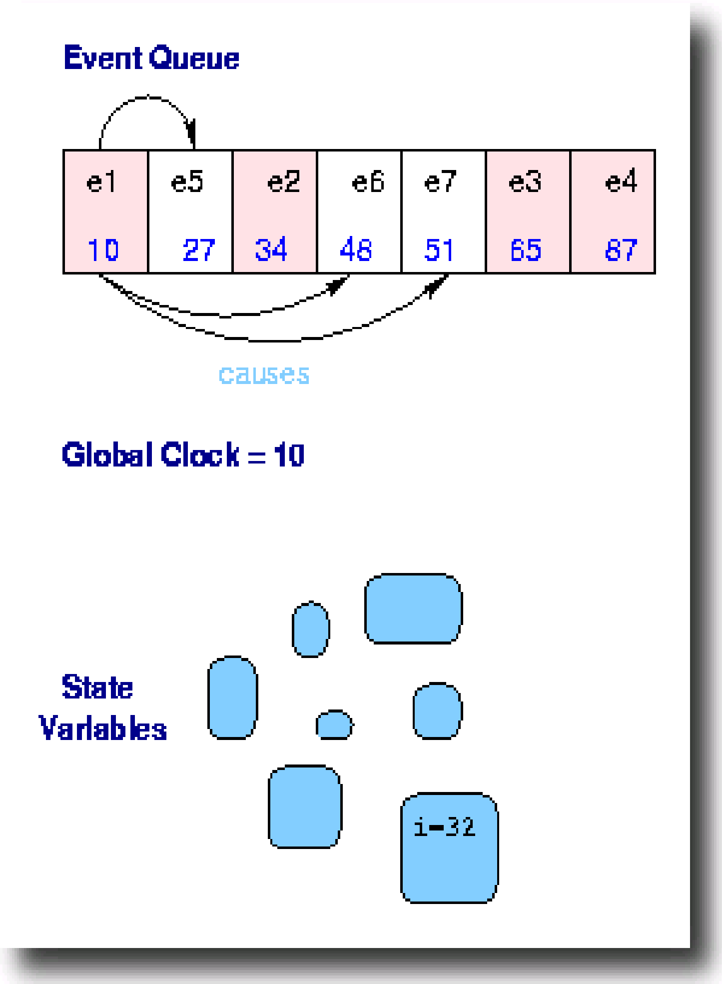
- Background
  - Discrete Event Simulation (DES)
  - Parallel Discrete Event Simulation (PDES)

- Parallel Object-oriented Simulation Environment (POSE)
  - Objectives
  - Object-oriented DES
  - Mixing sychronization protocols & global virtual time
  - Current performance
  - Load Balancing in POSE

Discrete Event Simulation

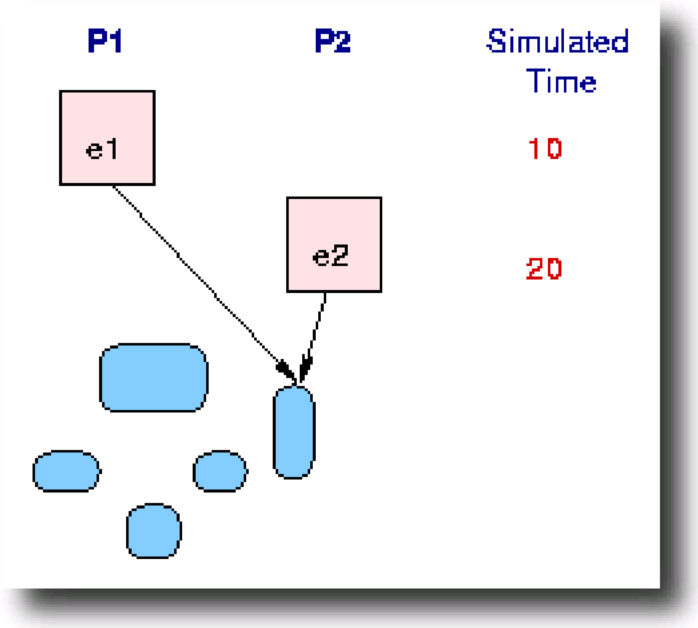
- **Discrete Event Simulation** (DES): simulation of complex systems in which state changes or *events* occur at discrete points in simulated time, typically at irregular time intervals
- Data structures for sequential DES:
  - *State variables* describe the state of the system
  - The *event queue* contains pending scheduled events
  - The *global clock* keeps track of simulation progress
- Each event has a *timestamp* and typically changes or accesses the state variables in some way
- An event can also schedule new events for the simulated future
- Always select event with minimum timestamp from event queue to avoid *causality* errors

Discrete Event Simulation



Parallel Discrete Event Simulation

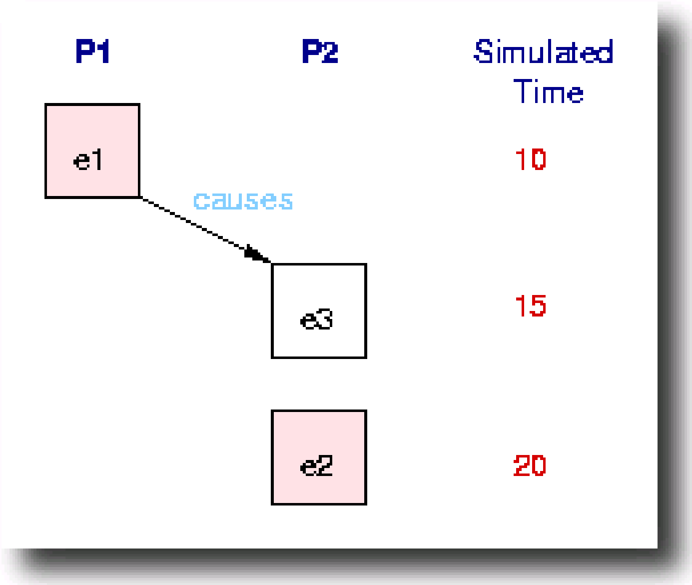
- How can we parallelize DES?
  - Distribute the events across processors, shared memory



- Need *sequencing constraints* to ensure correctness

Parallel Discrete Event Simulation

- Distribute the state variables across processors as well



- Still must handle *event causality errors*

POSE

Parallel Discrete Event Simulation

- *Physical processes* in the system being modeled are mapped onto *logical processes* (LPs), each containing a portion of the state and a local clock. LPs interact via timestamped event messages.
- To ensure no shared-state causality errors, LPs process events in nondecreasing timestamp order, i.e. they adhere to the *local causality constraint*
- Preventing event causality errors is more difficult -- sequencing constraints are complex and highly data-dependent
- Two broad categories of mechanisms for handling sequencing restraints: *conservative* and *optimistic*

PDES Mechanisms

- Conservative Mechanisms
  - Avoid the possibility of the occurrence of causality errors
  - Rely on the ability to determine when it is safe to process an event

- Optimistic Mechanisms
  - *Detect and recovery*: detect causality errors and rollback the computation to recover from them

POSE

PDES Mechanisms

- Optimistic mechanisms speculate that a causality error will not occur, i.e. they perform *speculative computations*
- An event arriving with a timestamp earlier than events that were executed speculatively, or a *straggler* event, causes a *rollback*.
- A rollback involves undo-ing executed events: the local state must be restored (possibly from *checkpointed* state data), and any caused events must be *cancelled*.
- We still monitor *safety*, via the *global virtual time* (GVT): the smallest timestamp among all unprocessed event messages
- Actions performed with timestamp prior to GVT can be *committed*: allows for reclaimation of checkpoint space and committing irrevocable operations (such as I/O)

POSE Objectives

- A usable language: focus on modeling the system, hide the parallelism, hide much of the simulation engine
  - POSE is a C++-like subset of Charm++

- Good performance: scalable to large numbers of processors
  - Base implementation of POSE scales well to 16, to 32 on larger problems, and to 64 on the largest problems
  - Develop load balancers that take into account the special irregularities of PDES system models
  - Explore hierarchical approaches to modeling for PDES

Object-Oriented DES

- The object-oriented programming paradigm offers a natural approach to modeling both data and processes
- LPs and state variables translate directly into *objects*
- Event messages correspond to timestamped *method invocations*
- Data encapsulation will make load balancing straightforward later on
- Charm++ provides much support for PDES (without ever meaning to!)
- LPs and state variables are easily distributed via *chares* with event messages provided as chare *entry points*
- Prioritized messages and the scheduler act to presort timestamped events before delivery

Synchronization Protocols and the GVT

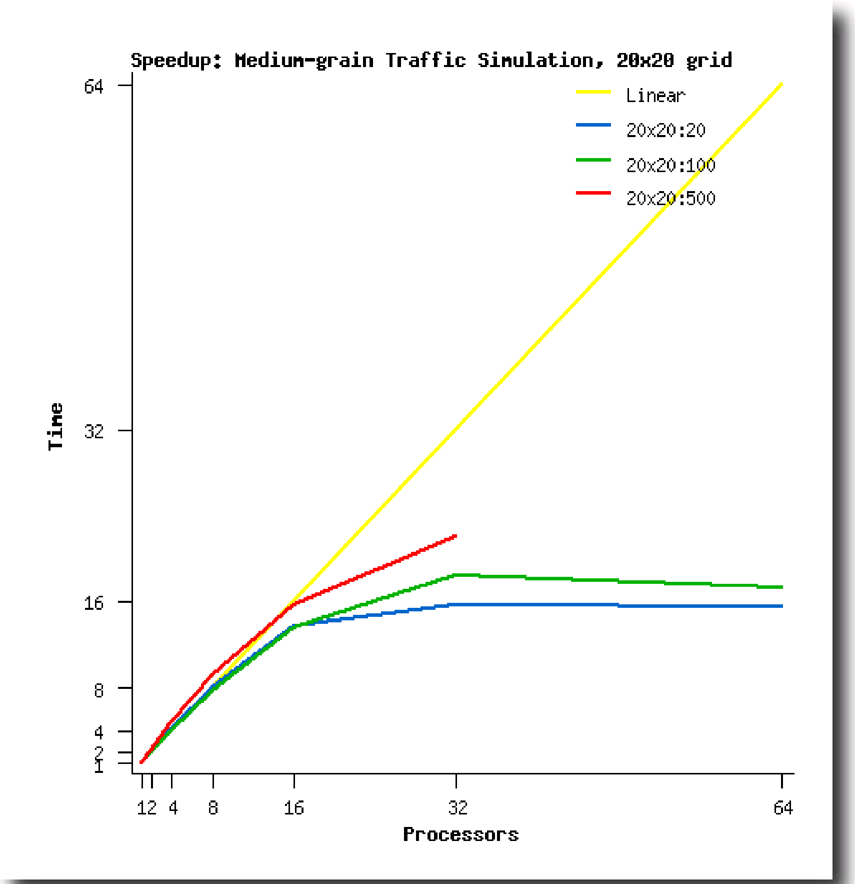
- POSE allows for both conservative and optimistic methods in the same simulation; two simple versions are provided
- GVT algorithm drives the very simple conservative mechanism that uses no lookahead or deadlock detection
- Optimistic mechanism uses checkpointing and has a "flexible leash" to control its speculativeness

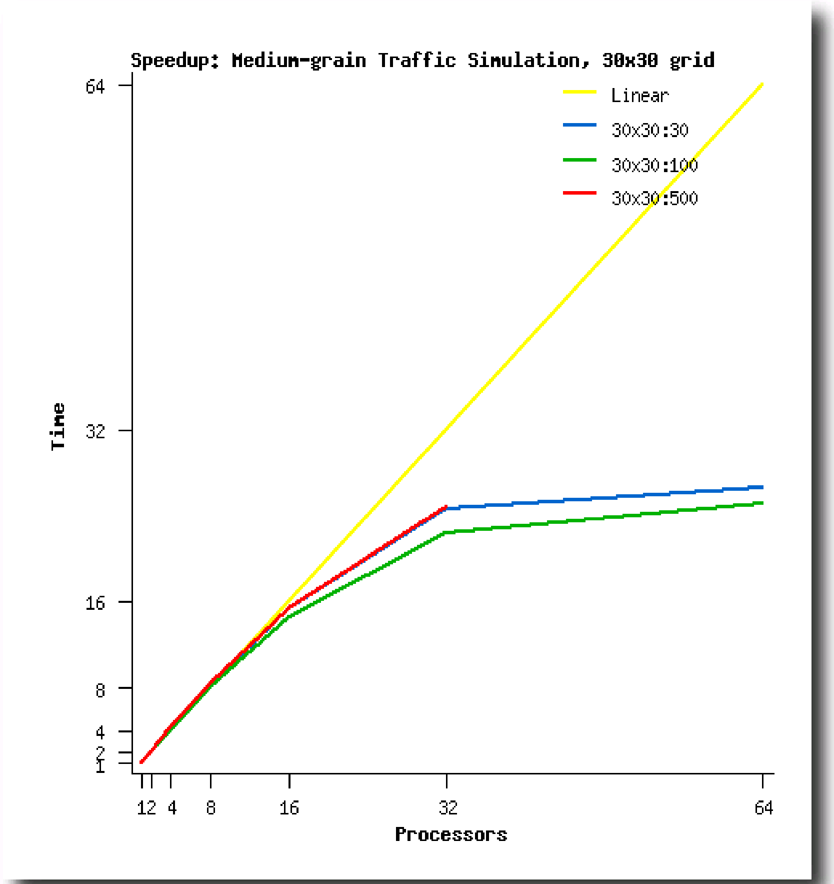
Current Performance

- Speedups beyond 16 are difficult
  - Load balancing could answer this problem

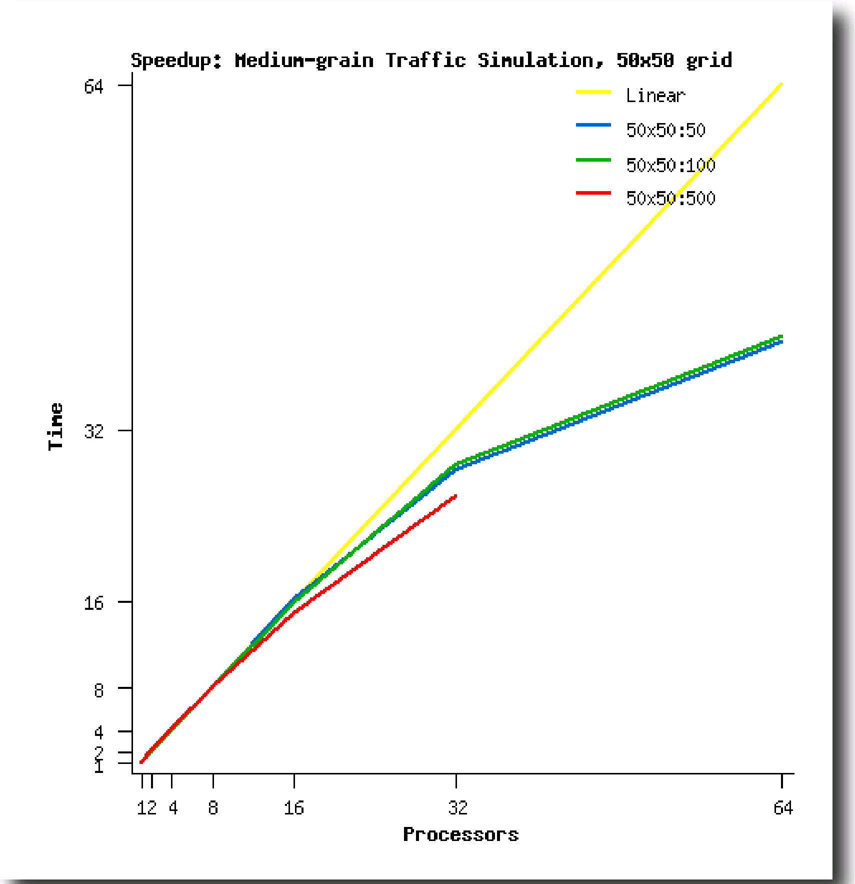
- Fine-grained simulations are the hardest to scale up
  - More time is spent on communication
  - Could load balance based on object interactions to reduce communication overhead

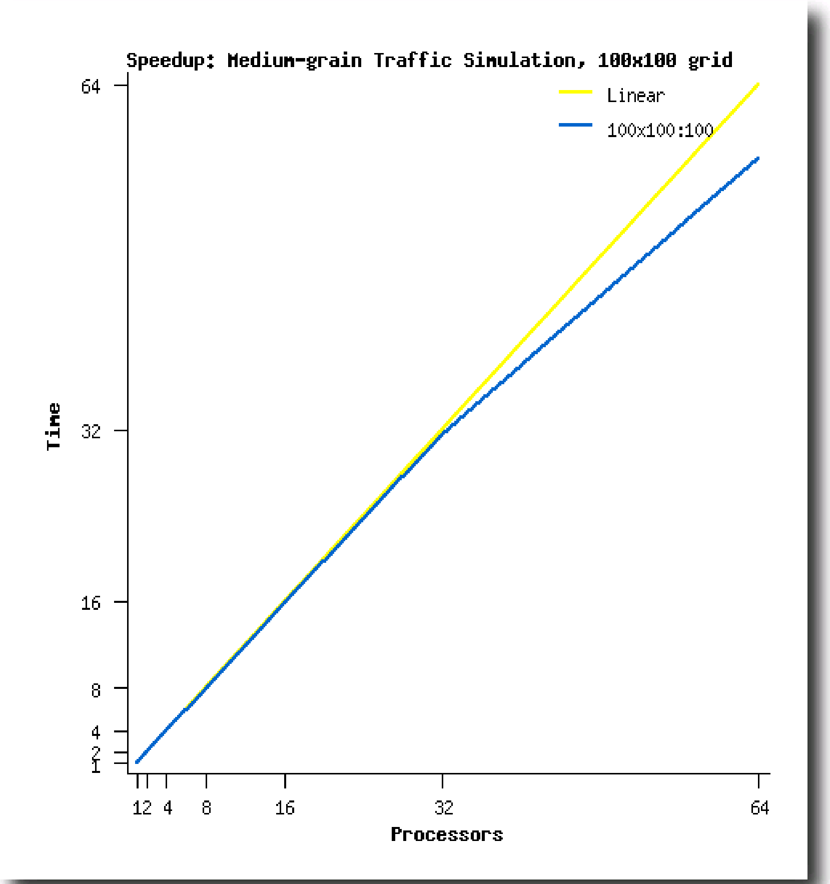
Speedup: Medium-grained Traffic Simulation



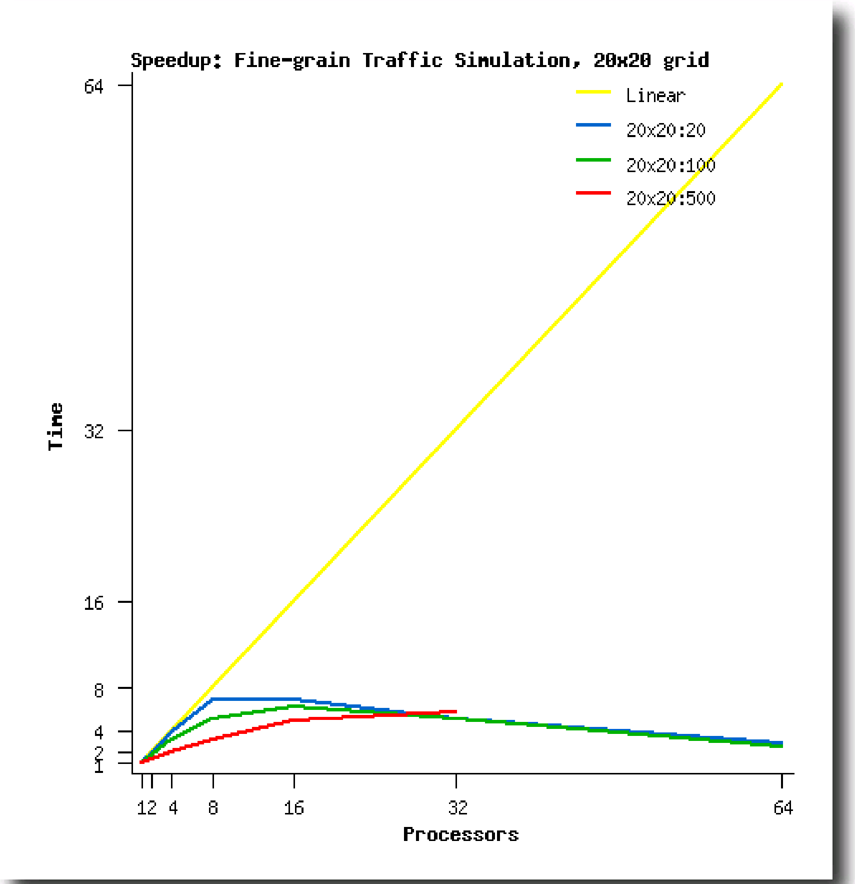


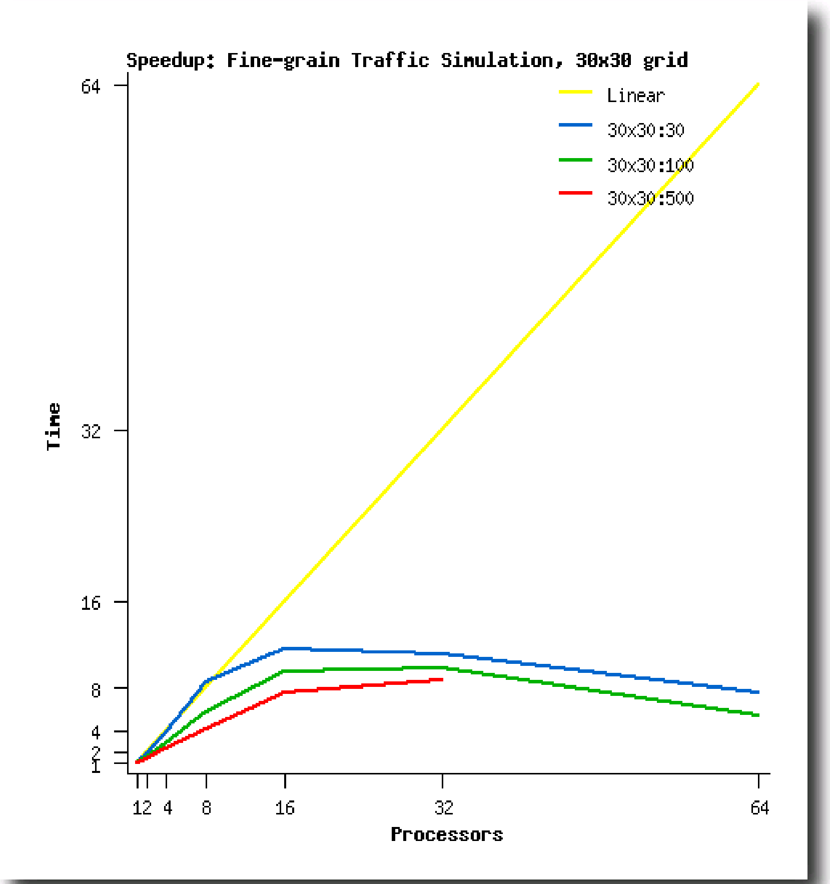
Speedup: Medium-grained Traffic Simulation



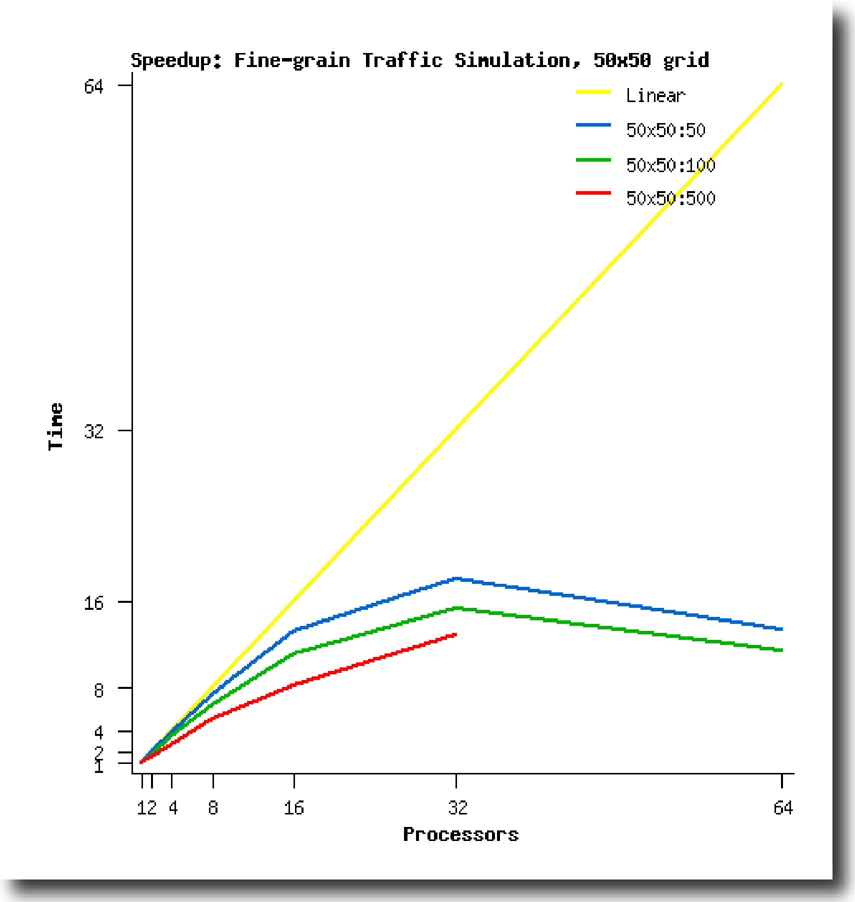


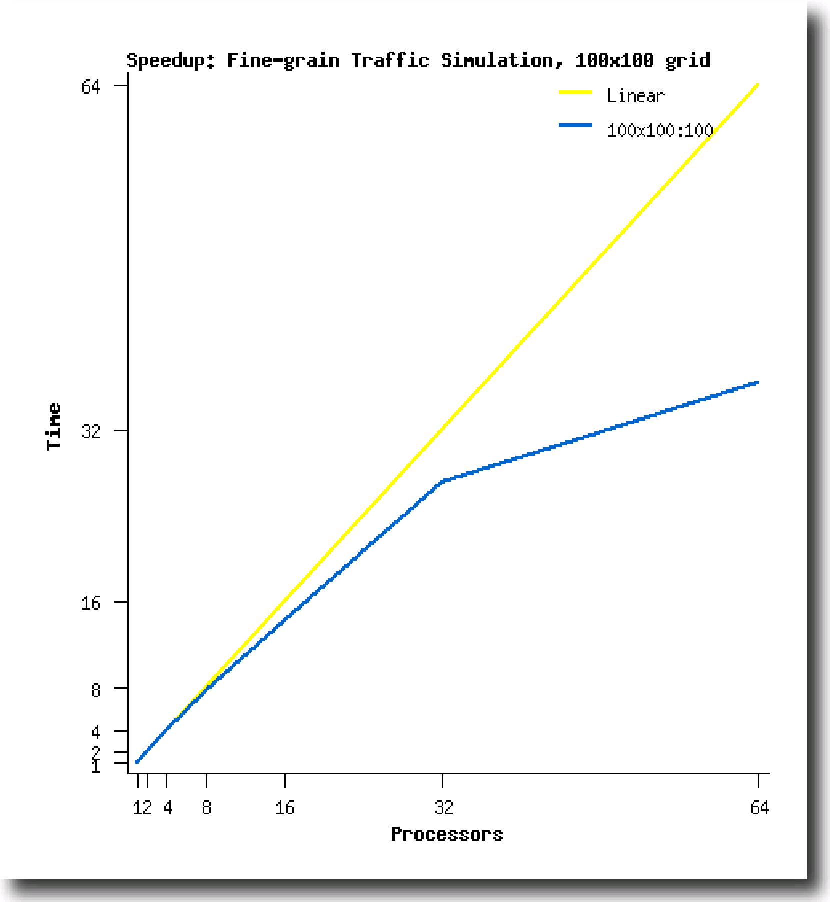
Speedup: Fine-grained Traffic Simulation



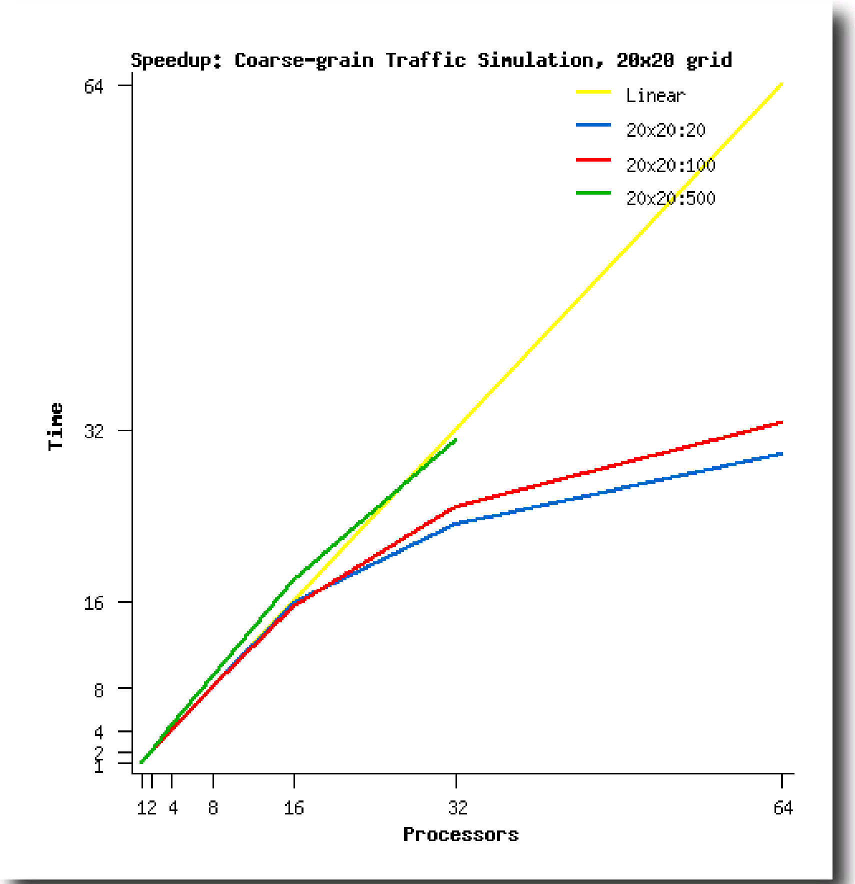


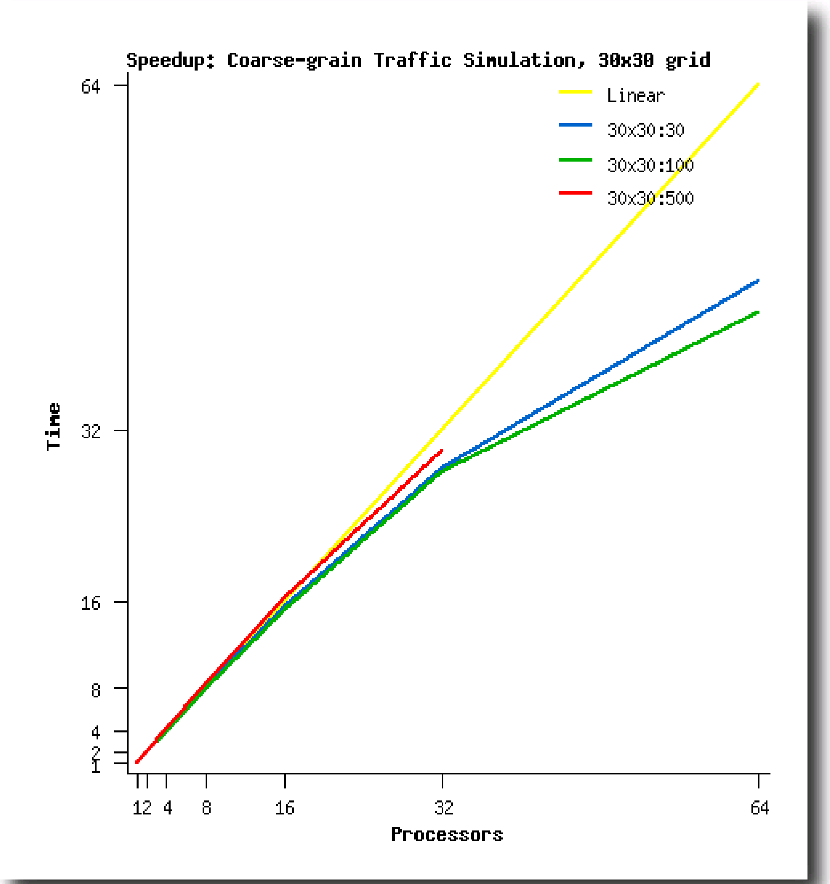
Speedup: Fine-grained Traffic Simulation



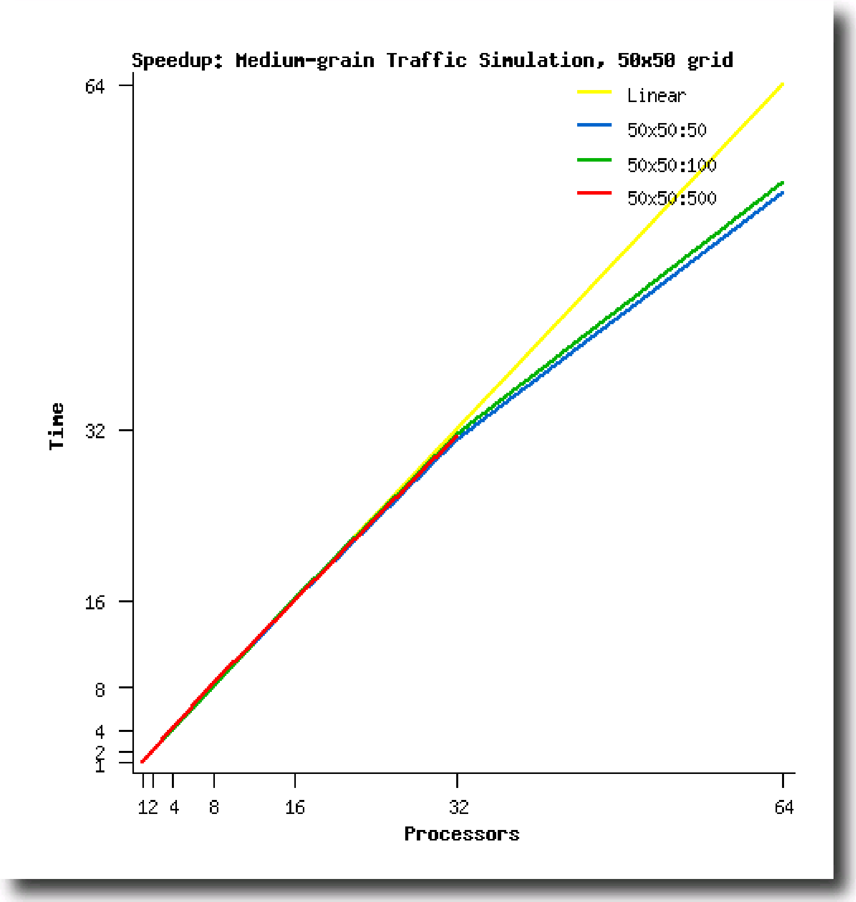


Speedup: Coarse-grained Traffic Simulation





Speedup: Coarse-grained Traffic Simulation



POSE

Load Balancing

- Ordinary LB strategies insufficient: cannot balance to minimize idle time!
- Should take into account type of load: forward execution, speculative computation, or rollbacks
- *Simulation priority load balancing*: determines execution priorities for simulation objects and balances to even out the priority load
- *Execution priority* has four determiners: *object virtual time*, *execution forecast*, *speculative forecast*, and *rollback overhead*
- Given execution priorities for each object, find A_i, the average execution priority on processor i. Priority load P_i on processor i is $(o_i + w)/A_i$. o_i is the number of objects on P_i, and w is a weight.

POSE

Load Balancing

- Given priority loads for all processors, how should we design our LB strategy?

- A strategy can even out the priority loads on all processors, and/or it can strive for *mix quality* on all processors
  - What does it mean to be *priority balanced*?
  - What *migrates*?
  - How *thorough* should the strategy be?
  - *When* should the load balancer be invoked?

Load Balancing

- Two target strategies:
  - "Perfect" Load Balancing Strategy (PLBS) : attempts to achieve nearly identical P_i on all processors and good mix quality; migrates whatever is necessary to improve the load; execution priority update is constant; changes trigger imbalance check; rebalance performed whenever slight imbalance is detected
  - "Quick" Load Balancing Strategy (QLBS): prepares for future balance by moving medium and low priority objects; invoked periodically; requests priority updates from objects; checks for imbalance above a generous threshold; moves lightweight objects to get imbalance below threshold