Proposed Research Topic:

Zero-Config Automatic Parallel Simulation

András Varga, Levente Mészáros
OMNeT++ Core Team
“Proposed Research Topic”

• NOT finished research.
• NOT even research underway.

• A promising research topic for those looking for one.
  – (We see potential in the idea and find it exciting, but we don’t have the resources [mostly, time] to elaborate it in-house.)

• Why?
  – Practically VERY useful
    • Everybody would love their simulations to run X times faster on common hardware!
  – Doable
    • We have already spent some time trying out the idea and proven (at least to ourselves) that it is feasible and the approach outlined here can be made to work.
  – Novel
    • Related research only took of a few years ago
  – Plenty of questions and degrees of freedom
    • publication opportunities!
Two Questions

What is zero-configuration parallel simulation? (and why is it called so?)

Doesn’t OMNeT++ have parallel simulation support already...?
OMNeT++ Parallel Simulation Support

1. Partition the network
   - Each partition will be run in a separate LP (logical process)

Partition – how...?
   - interaction between partitions should be *minimal*
   - link delays across partitions should be *high*
   - workload should be *evenly* distributed
2. Describe this partitioning in omnetpp.ini

[General]
parallel-simulation = true
*.rte[0..4].partition-id = 0
*.rte[5..17].partition-id = 1
*.rte[22].partition-id = 1
*.rte[18..21].partition-id = 2
*.rte[23..24].partition-id = 2
...

Here's the "configuration"
3. Run the simulation on a multiprocessor

- Each partition (logical process, LP) will be a separate simulation process
- Executing on its own CPU (or core)
- Communication over MPI

- Hardware: multicore laptop/desktop, HPC cluster
  (low communication latency is essential, more so than bandwidth)
OMNeT++ Parallel Simulation Support

Limitations:
- Global variables
- Accessing modules in other partitions
- Method calls across partition boundaries
- Simsignal propagation across partition boundaries
- Moving modules to a different partition (?)

Overhead:
- Communication overhead
- Synchronization overhead (lookahead is critical)

Can run on clusters (distributed memory multiprocessors) too, but on multicore CPUs, doesn’t properly take advantage of shared memory
OMNeT++ Parallel Simulation Support

Why synchronization is needed

Example: Two LPs, each of them executing events independently in timestamp order, and sending events to each other.

Already simulated

Message

Local simulation time

BANG!!!

(CausalitY Violation)
Maintaining Event Causality

- The future should not affect the past. That is, processing an event must not have an effect on events with smaller timestamps*.

  - This is the main problem of Parallel Discrete Event Simulation (PDES).

* More precisely, the (timestamp, priority, insertOrder) triplet is used by OMNeT++ for ordering events
PDES Approaches

Conservative

- do not allow causality violations
- example: null-message protocol, a.k.a Chandy-Misra-Bryant

- performance: "lives or dies by the lookahead" (e.g. link delays)
- implementation: straightforward
- chosen by OMNeT++

Optimistic

- allow incausalities, detect them, and repair them by rolling back
- example: Time Warp algorithm

- performance: may suffer from excessive rollbacks
- implementation: complicated protocol (anti-messages etc), laborious implementation (state saving & restoration needs to be implemented in each and every model component, as C++ provides no STM solution)

- so only necessary if Conservative cannot fully utilize the hardware
Diverging From the LP-Based Approach

Why?

- Advances in hardware
  - increase in single-core performance slowed, number of cores steadily increasing instead
    - 4 cores standard, 8/12/16+ cores, etc. available ⇒ HPC clusters less needed
  - memory abounds
    - 8/16G is standard, 32/64G and up easily available ⇒ “distribute memory requirements” argument for LP-based PDES no longer holds

- Limitations of LP approach
  - coding limitations (no access across partitions, etc.)
  - overhead (communication, serialization; unable to take full advantage of shared memory systems)
  - inconvenience (mpi_run, etc)
Multi-Threaded Simulation

Worker threads take events from a shared FES, process them, and insert the resulting events into the FES.

Challenges:
1. Event causality must be kept
2. Concurrent access of data structures (FES, simulation objects)
Event Causality

“What if it exploded right now...?”
“... or 4.3 years ago?”

- Simultaneous events at both cannot affect each other
- Moreover: if time difference < 4.37 years → events cannot affect each other
Visualization: Space-Time Diagram

“Light cone” illustrates which part of the space-time an event can affect.

- A can affect B
- A cannot affect C
Event Coloring

As time progresses: 1. green events stay green; 2. red events may turn green

- **GREEN** events are independent (cannot be affected by any other event)
- **RED** events have dependencies (can be affected by others)
Applying to Simulation

Between modules, if only interaction is message passing:

\[\text{distance}(A,B) = 100\text{ms} \]
\[\text{distance}(B,A) = \infty\]

“100 light-milliseconds distance A-to-B”

\[\text{distance}(C,D) = \text{<total delay on shortest path>}\]
* using delay as metric
Zero-Config Parallel Simulation

• Meaning of coloring:
  – **Green** events can be executed in concurrently
  – **Red** events cannot

• During simulation:
  – **Worker threads** process **green** events
  – **Colorer** continually works on turning more events **green**
Mapping to Hardware

Coloring algorithm may run continuously in the background. [when done, wait for change in FES]

Separate thread/core can be dedicated to coloring

Coloring algorithm can be parallel in itself (if that’s the bottleneck)
Coloring Algorithm

Pseudocode:

for each red event in the FES:
    if it’s not in any other event’s “light cone”:
        mark it as green

A little more formally:

for each red event E1 in module M1 in the FES:
    T := (minimum of arrivalTime(E2) + distance(M2, M1) \n        for each event E2 in module M2 before E1 in the FES)
    if arrivalTime(E1) < T:
        mark E1 as green

T: time of earliest possible effect from other modules
distance(M2, M1): total delay on shortest path from M2 to M1
The distance() Function

• Precompute
  – Then keep up-to-date with topology changes

• Store as matrix
  – Requires $N^2$ space for $N$ modules
  – *Optimization possibility*: represent zero-delay module groups as one entry (row/col)
    • In INET, almost all modules within a host or router form such a zero-delay group → reduces matrix size
Non-Message Dependencies

- **Method call: instantaneous effect**
  - Action: “A performs B->f( )”
  - Setters: A→B dependence: distance(A,B)=0
    - like a zero-delay A→B message sending
  - Getters: B→A dependence: distance(B,A)=0
  - Mixed: mutual dependence

- **Global variable: instantaneous effect**
  - A writes, B reads: A→B dependence

- **Signals**
  - Listeners are “method calls in disguise”
    - as emit() indirectly invokes listeners
  - For all E emitter and L listener pairs: E→L dependence, i.e. distance(E,L)=0
Implementing the Colorer

- Pseudocode shows a naïve algorithm
  - Looks at all events every time
    - For performance, it should be incremental

- Issue:

Worker thread:
msg = fes->pop();
mod->handleMessage(msg);

handleMessage(msg) {
  delete msg /
  send(msg,..) /
  scheduleAt(t,msg)
}

Colorer: Removing an event and adding consequence events should happen atomically!

Resolution: Colorer must work on a view of the FES, not on the FES itself!
Worker Thread Scheduling

How should worker threads pick from the pool of green events?

- Grabbing >1 event at a time may reduce blocking overhead
- If \#events > \#cores:
  - in which order to serve them?
  - order affects performance
- If \#green < \#cores:
  - eager assignment?
  - being eager may not always be the best strategy
    - may pay off to wait for new events that contribute more to simulation progress
- Further observations?
Concurrent Access

• FES is under heavy concurrent access
  – locking
  – lock-free data structures

• Simulation model and state
  – Challenge: Cross-module method calls
    • Relaxing: If events within a (compound) module are NOT processed concurrently, inter-node accesses don’t need to be protected
  – Simsignals: Method calls in disguise!
    • Emitting a signal indirectly invokes the listeners
    • Listeners need to be protected against concurrent accesses
  – Model code needs to be instrumented for zero-config parsim!

• Simulation kernel and infrastructure
  – If model is static -- no protection needed
  – Dynamic module creation and other model changes
  – Result filters/recorders also need to be protected
“Are there enough green events in “normal” simulations?”

**Experiment:** We added a simple version of Colorer to an otherwise vanilla OMNeT++ INET simulation.

**Result:** Usually 4-5 green events in the FES in a network of 4 LANS, 4 hosts/LAN. This was a small simulation; we expect the number of green events to scale linearly with the size of the simulation ⇒ enough to keep all CPU cores busy.
Research Questions

Topics open to research:
• Choice of FES data structure
• Efficient Colorer algorithm
• Worker thread scheduling
If you are interested, please contact us!

END

(QUESTIONS?)