Parallel Simulation of Peer-to-Peer Systems

Martin Quinson, Cristian Rosa, Christophe Thiéry
(presented by Arnaud Legrand)

Université de Lorraine, France

ANR 08 SEGI 022
ANR 11 INFRA 13

CCGrid 2012
Context: Simulation of Peer-to-Peer Systems

Large Scale Distributed Systems exist already

- P2P, Grids, Clouds, Volunteer Computing, ... 
- But these systems remain very hard to study (performance, correction)

Classical Scientific Pillars Apply

- Theoretical Approach: Mathematical study of algorithms
- Experimental Science: Study applications on scientific instrument
- Computational Science: Simulation of a system model

Performance Study ~ Experimentation

- Theory still mandatory, but everything’s NP-hard
- Experimental Facilities: Real applications on Real platform \( (in \text{ vivo}) \)
  - 😊 No bias; 😞 Difficult; no Experimental Control. Reproducibility?
- Simulation: Prototypes of applications on system’s Models \( (in \text{ silico}) \)
  - 😞 Experimental bias? 😊 Simple. Perfect control/reproducibility;

Most studies in the literature are conducted through simulation
Parallel Simulation: Three decades of literature

- All sort of simulations have been parallelized: continuous, discrete event
- For all sort of domains: physics, biology, economics, social science, engineering

Yet, Mainstream Simulators of P2P Systems are not Parallel

- Astonishing since scalability is their main goal
- How to leverage multi-core architecture?

Objectives of this talk (and agenda)

- Contrast classical parallelisation schema and classical dist.apps simulation
- Propose a new parallelisation schema adapted to our settings
- Propose specific techniques to efficiently implement that schema
- Present some experimental results using the SimGrid framework
State of the Art on P2P simulators

No de facto simulator

- Huge amount of short lived simulators (not to say one shot)
- This jeopardizes research works that become obsolete when the tool disappears
- Many survey authors wish that a standard tool emerges eventually

PeerSim

- Simple enough to get adapted, but no network in model (abstracted)
- Query-cycle mode (application as automata): 10^6 nodes; DES: 10^3
- Query-cycle: user-unfriendly way to express dist. apps; DES: sequential

OverSim

- Scalable: 10^5 nodes using simplistic network models
- Realistic: can leverage the omNET++ packet-level simulator
- Simplistic models are sequential, parallel omNET++ seemingly never used

PlanetSim

- Parallel execution, but query-cycle mode only (embarrassingly parallel)
Parallel P2P simulators: the dPeerSim attempt

**dPeerSim**
- Parallel implementation of PeerSim/DES (not by PeerSim main authors)
- Classical parallelization: spreads the load over several Logical Processes (LP)

![Diagram of parallel processes](image)

**Experimental Results**
- Uses Chord as a standard workload: e.g. 320,000 nodes \(\sim\) 320,000 requests
- The result are impressive at first glance
  - 4h10 using two Logical Processes: only 1h06 using 16 LPs
  - Speedup of 4 using 8 times more resources, that really not bad
- But this is to be compared to sequential results
  - The same simulation takes 47 seconds in the original sequential PeerSim
  - (and 5 seconds using the precise network models of SimGrid in sequential)
## Parallel Simulation vs. Dist. Apps Simulators

### Simulation Workload
- Granularity, Communication Pattern
- Events population, probability & delay
- #simulation objects, #processors

### Simulation Engine
- Parallel protocol, if any:
  - Conservative (lookahead, ...)
  - Optimistic (state save & restore, ...)
- Event list mgnt, Timing model...

### Execution Environment
- OS, Programming Language (C, Java...), Networking Interface (MPI, ...)
- Hardware aspects (CPU, mem., net)

### Classical Parallel Simulation Schema
[Balakrishnan et al]

- The **classical approach** is to distribute the Simulation Engine entirely
- Hard issues: conservatives \(\sim\) too few parallelism; optimistic \(\sim\) roll back
- From our experience, most of the time is in so called “simulation workload”
  - User code executed as threads, that are scheduled according to simulation
  - The user code itself can reveal resource hungry: numerous / large processes
**Main Idea of this Work**

**Split at Virtualization, not Simulation Engine**
- Virtualization contains threads (user’s stack)
- Engine & Models remains sequential

- **Understanding the trade-off**
  - **Sequential time:** \( \sum_{SR} (\text{engine} + \text{model} + \text{virtu} + \text{user}) \)
  - **Classical schema:** \( \sum_{SR} \left( \max_{i \in LP} (\text{engine}_i + \text{model}_i + \text{virtu}_i + \text{user}_i) + \text{proto} \right) \)
  - **Proposed schema:** \( \sum_{SR} \left( \text{engine} + \text{model} + \max_{i \in WT} (\text{virtu}_i + \text{user}_i) + \text{sync} \right) \)
  - Synchronization protocol expensive wrt the engine’s load to be distributed

---

Quinson, Rosa, Thiéry  Parallel Simulation of Peer-to-Peer Systems.
Enabling Parallel Simulation of Dist.Apps

Challenge: Allow User-Code to run Concurrently

- DES simulator full of shared data structures: how to avoid race conditions?
- Fine-locking would be difficult and inefficient; wouldn’t avoid app-level races
  - A: \textit{recv}, B: \textit{send}, C: \textit{send}; Which \textit{send} matches the \textit{recv} from A in simulation?
  - Depends on execution order in host system \(\sim\) simulation not reproducible...
Enabling Parallel Simulation of Dist. Apps

Challenge: Allow User-Code to run Concurrently

- DES simulator full of shared data structures: how to avoid race conditions?
- Fine-locking would be difficult and inefficient; wouldn’t avoid app-level races
  - A: recv, B: send, C: send; Which send matches the recv from A in simulation?
  - Depends on execution order in host system \( \sim \) simulation not reproducible...

Solution: OS-inspired Separation Simulated Processes

- Mediate any interaction of processes with their environment, as in real OSes
  e.g. don’t create a new process directly, but issue a simcall to request creation

```plaintext
1: t ← 0
2: Pt ← P
3: while Pt ≠ ∅ do
4:   parallel_schedule(Pt)
5:   handle_simcalls()
6:   (t, events) ← models_solve()
7:   Pt ← proc_to_wake(events)
8: end while
```

Quinson, Rosa, Thiéry  Parallel Simulation of Peer-to-Peer Systems.
Enabling Parallel Simulation of Dist.Apps

Challenge: Allow User-Code to run Concurrently

▶ DES simulator full of shared data structures: how to avoid race conditions?
▶ Fine-locking would be difficult and inefficient; wouldn’t avoid app-level races
  ▶ A: recv, B: send, C: send; Which send matches the recv from A in simulation?
  ▶ Depends on execution order in host system \( \leadsto \) simulation not reproducible...

Solution: OS-inspired Separation Simulated Processes

▶ Mediate any interaction of processes with their environment, as in real OSes e.g. don’t create a new process directly, but issue a \textit{simcall} to request creation

\begin{verbatim}
1: \( t \leftarrow 0 \)
2: \( P_t \leftarrow P \)
3: \textbf{while} \( P_t \neq \emptyset \) \textbf{do}
4: \hspace{1cm} parallel_schedule(\( P_t \))
5: \hspace{1cm} handle_simcalls()
6: \hspace{1cm} (\( t, events \)) \leftarrow models_solve()
7: \hspace{1cm} \( P_t \leftarrow proc_to_wake(events) \)
8: \textbf{end while}
\end{verbatim}

▶ Processes isolated from each others
  ▶ Simcalls data locally stored
▶ Simcalls handled centrally once users blocked
  ▶ Arbitrary fixed order for reproducibility
Efficient Parallel Simulation

Leveraging Multicores

- P2P involve millions of user processes, but dozens of cores at best
- Having millions of System threads is difficult (when possible)
- Co-routines (Unix ucontexts, Windows fibers): highly efficient but not parallel
- N:M model used: millions of coroutines executed on few threads

Reducing Synchronization Costs

- Inter-thread synchronization achieved through system calls (of real OS)
- Costs of syscalls are critical to performance \(\sim\) save all possible syscalls
- Assembly reimplementation of ucontext: no syscall on context switch
- Synchronize only at scheduling round boundaries using futexes
- Dynamic load distribution: hardware fetch-and-add next process’ index
Microbenchmarking Synchronization Costs

Rq: P2P and Chord are ultra fine grain: this is thus a worst case scenario

Comparing our user context containers

- pthreads hit a scalability limit by 32,000 processes (amount of semaphores)
- System contexts and ASM contexts have no hard limit (beside available RAM)
- pthreads are about 10 times slower than our own ASM contexts
- ASM contexts are about 20% faster than system ones
  (only difference: avoid any syscalls on user context switches)

Measuring intrinsic synchronization costs

- Disabling parallelism at runtime: no noticeable performance change
- Enabling parallelism over 1 thread: 15% performance drop off
- Demonstrate the difficulty although the careful optimization
Sequential Performance in State of the Art

- **Scenario:** Initialize Chord, and simulate 1000 seconds of protocol
- **Arbitrary Time Limit:** 12 hours (kill simulation afterward)

### Largest simulated scenario

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Omnet++</td>
<td>10k</td>
<td>1h40</td>
</tr>
<tr>
<td>PeerSim</td>
<td>100k</td>
<td>4h36</td>
</tr>
<tr>
<td>OverSim</td>
<td>300k</td>
<td>10h</td>
</tr>
<tr>
<td>SimGrid (precise)</td>
<td>10k</td>
<td>130s</td>
</tr>
<tr>
<td>SimGrid (constant)</td>
<td>300k</td>
<td>32mn</td>
</tr>
<tr>
<td>SimGrid (constant)</td>
<td>2M</td>
<td>6h23</td>
</tr>
<tr>
<td>SimGrid (simple)</td>
<td>2M</td>
<td>5h30</td>
</tr>
</tbody>
</table>

### Memory Usage

- 2M precise nodes: 32 GiB
- That is 18kiB per process
  
  (User stack: 12kiB)

Extra complexity to allow parallel execution don’t impact sequential perf
Benefits of the Parallel Execution

- Speedup ($\frac{t_{seq}}{t_{par}}$): up to 45%
- More efficient with simple model:
  - Less work in engine + Amhdal law
- Speedup depends on thread amount
  - 8 threads (of 24 cores) often better
  - Synch costs remain hard to amortize
  - They depend on thread amount

Parallel Efficiency ($\frac{\text{speedup}}{\#\text{cores}}$) for 2M nodes

<table>
<thead>
<tr>
<th>Model</th>
<th>4 threads</th>
<th>8 th.</th>
<th>16 th.</th>
<th>24 th.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precise</td>
<td>0.28</td>
<td>0.15</td>
<td>0.07</td>
<td>0.05</td>
</tr>
<tr>
<td>Constant</td>
<td>0.33</td>
<td>0.16</td>
<td>0.08</td>
<td>0.06</td>
</tr>
</tbody>
</table>

- Baaaaad efficiency results
- Remember, P2P and Chord: Worst case scenarios

Yet, first time that Chord’s parallel simulation is faster than best known sequential
Conclusions and Future Work

Context  DES of dist apps constitute a specific simulation workload
  ▶ Yet, scalability considered as main goal to P2P simulators (at least)
Problem  Classical parallelisation is suboptimal (spatial decomposition)
  ▶ Optimistic’s rollbacks difficult with complex network models
  ▶ Pessimistic look ahead limited because P2P app topology ≠ network one
  ⇒ dPeerSim: 2 LPs: 4h; 16 LPs: 1h, but 47 seconds sequential without LPs
Proposal  Better to keep central engine and leverage virtualization threads
  ▶ Making this possible mandates an OS-inspired separation of processes
  ▶ Making this efficient for P2P mandates to reduce synchros to bare minimum
Evaluation  Implemented in SimGrid (http://simgrid.gforge.inria.fr)
  ▶ Still orders of magnitude faster than PeerSim and OverSim in sequential
  ▶ Parallel execution (somehow) beneficial for (very) large amount of processes

Take home message
  ▶ Parallel P2P simulator mandates creative approaches and careful optimization

Future work
  ▶ Further technical improvements (automatic tuning thread amount; Java bindings)
  ▶ Attempt distribution (beyond memory limit and for HPC tasks)
  ▶ Leverage this tool to conduct nice studies