SimGrid Kernel 101
Introducing the SimGrid Kernel

Da SimGrid Team

May 27, 2014
About this Presentation

Goals and Contents

▶ Present Simix, the simulation kernel of SimGrid
▶ Show some gore details about where our performance comes from
▶ This is NOT for newcomers but for hardcore SimGrid users (or curious folks)

The SimGrid 101 serie

▶ This is part of a serie of presentations introducing various aspects of SimGrid
  ▶ SimGrid 101. Introduction to the SimGrid Scientific Project
  ▶ SimGrid User 101. Practical introduction to SimGrid and MSG
  ▶ SimGrid User::Platform 101. Defining platforms and experiments in SimGrid
  ▶ SimGrid User::SimDag 101. Practical introduction to the use of SimDag
  ▶ SimGrid User::Visualization 101. Visualization of SimGrid simulation results
  ▶ SimGrid User::SMPI 101. Simulation MPI applications in practice
  ▶ SimGrid User::Model-checking 101. Formal Verification of SimGrid programs
  ▶ SimGrid Internal::Models. The Platform Models underlying SimGrid
  ▶ SimGrid Internal::Kernel. Under the Hood of SimGrid
▶ Get them from http://simgrid.gforge.inria.fr/documentation.html
SimGrid Internals in a Nutshell

Example of user code to execute

Alice

Send "toto" to Bob
Listen from Bob

Bob

Listen from Alice
Send "blah" to Alice
SimGrid Internals in a Nutshell

Example of user code to execute

Alice
Send "toto" to Bob
Listen from Bob

Bob
Listen from Alice
Send "blah" to Alice
SimGrid Internals in a Nutshell

Example of user code to execute

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send &quot;toto&quot; to Bob</td>
<td>Listen from Alice</td>
</tr>
<tr>
<td>Listen from Bob</td>
<td>Send &quot;blah&quot; to Alice</td>
</tr>
</tbody>
</table>

SimGrid internal Main Loop

1. Run every ready user process in row
   - Each wants to consume resources
   - Assign actions on resources
2. Compute share for actions
3. Get earliest finishing action
4. Unlock user code waiting on this action
SimGrid Internals in a Nutshell

Example of user code to execute

Alice
Send "toto" to Bob
Listen from Bob

Bob
Listen from Alice
Send "blah" to Alice

SimGrid internal Main Loop

1. Run every ready user process in row
   - Each wants to consume resources
   - Assign actions on resources
2. Compute share for actions
3. Get earliest finishing action
4. Unlock user code waiting on this action

SimGrid Functional Organization

- **MSG**: User-friendly syntaxic sugar
- **Simix**: Processes, synchro (SimPosix)
- **SURF**: Resources usage interface
- **Models**: Action completion computation
Introduction Example

function $P_1$

// Compute...
Send()
...
end function

function $P_2$

// Compute...
Recv()
...
end function

$time \leftarrow 0$

$P_{time} \leftarrow \{P_1, P_2\}$

while $P_{time} \neq \emptyset$

\begin{align*}
& \text{schedule}(P_{time}) \\
& \quad \quad \quad time \leftarrow \text{solve}(\&done\_actions) \\
& \quad \quad \quad P_{time} \leftarrow \text{proc\_unblock}(done\_actions)
\end{align*}

end while

SimGrid’s Main Loop

$P_2$

$P_1$

$M \rightarrow$ time

0
**Introduction Example**

```plaintext
function P1
  //Compute...
  Send()
  ...
end function

function P2
  //Compute...
  Recv()
  ...
end function

\[
\begin{align*}
time &\leftarrow 0 \\
P_{time} &\leftarrow \{P_1, P_2\} \\
\text{while } P_{time} \neq \emptyset & \text{ do} \\
  & \quad \text{schedule}(P_{time}) \\
  & \quad time \leftarrow \text{solve}(\&\text{done\_actions}) \\
  & \quad P_{time} \leftarrow \text{proc\_unblock}(\text{done\_actions}) \\
\text{end while}
\end{align*}
```

SimGrid’s Main Loop

![Diagram](image_url)
**Introduction Example**

```plaintext
function P1
    //Compute...
    Send()
    ...
end function

function P2
    //Compute...
    Recv()
    ...
end function

\( time \leftarrow 0 \)
\( P_{time} \leftarrow \{ P_1, P_2 \} \)

while \( P_{time} \neq \emptyset \) do
    schedule(\( P_{time} \))
    \( time \leftarrow \text{solve}(&\text{done\_actions}) \)
    \( P_{time} \leftarrow \text{proc\_unblock}(\text{done\_actions}) \)
end while

SimGrid's Main Loop
```

![Diagram](image-url)
function P1
//Compute...
Send()
...
end function

function P2
//Compute...
Recv()
...
end function

time ← 0

P_time ← {P_1, P_2}

while P_time ≠ ∅ do
    schedule(P_time)
    time ← solve(&done_actions)
    P_time ← proc_unblock(done_actions)
end while

SimGrid's Main Loop
function P1 //Compute...  
Send()  
...  
end function

function P2 //Compute...  
Recv()  
...  
end function

time ← 0  
$P_{time} \leftarrow \{P_1, P_2\}$  
while $P_{time} \neq \emptyset$ do  
schedule($P_{time}$)  
$\text{time} \leftarrow \text{solve}(\&\text{done\_actions})$  
$P_{time} \leftarrow \text{proc\_unblock}(\text{done\_actions})$
end while

SimGrid's Main Loop
**Introduction Example**

function P1

//Compute...
Send()
...
end function

function P2

//Compute...
Recv()
...
end function

\[ time \leftarrow 0 \]

\[ P_{\text{time}} \leftarrow \{ P_1, P_2 \} \]

while \( P_{\text{time}} \neq \emptyset \) do

\[ \text{schedule}(P_{\text{time}}) \]

\[ time \leftarrow \text{solve}(\&\text{done_actions}) \]

\[ P_{\text{time}} \leftarrow \text{proc_unblock}(\text{done_actions}) \]
end while

SimGrid’s Main Loop

![Diagram showing the simulation round with arrows indicating Send and Recv actions between P1 and P2.]
Simix as an OS (Operating Simulator)

Requirements

▶ User code to run in a thread-like thing, we control the scheduling
▶ We want portability
→ generic mechanisms; several implementations
▶ We want to run the processes in parallel; we want model-checking
→ Isolate processes from each others
▶ We want it as efficient as possible → That’s what an OS does!

Chosen Design

▶ Processes are perfectly isolated from environment
  simcalls: only way of interacting with others/platform
  The maestro runs that code “in kernel mode”
▶ Processes virtualized with context factories
  Threads (pthread/win); ucontexts; Raw assembly
  Java contexts, Java continuations, Ruby contexts
How efficiently can we simulate P2P Protocols

P2P is a nightmare for the simulator

- People want huge fine grained systems (many events in large platforms)
- As a result, no standard too. Many short lived ones (even one shoot ones)
- If we manage be efficient on this workload, others will be easy

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)

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 here

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: \( \sum_{SR} \left( \max_{i \in LP} \left( \text{engine}_i + \text{model}_i + \text{virtu}_i + \text{user}_i \right) + \text{proto} \right) \)

▶ Proposed schema: \( \sum_{SR} \left( \text{engine} + \text{model} + \max_{i \in WT} \left( \text{virtu}_i + \text{user}_i \right) + \text{sync} \right) \)

▶ Synchronization protocol expensive wrt the engine’s load to be distributed
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...
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 \( \rightsquigarrow \) 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

```
1: t ← 0
2: P_t ← P
3: while \( P_t \neq \emptyset \) do
4:   parallel_schedule(\( P_t \))
5:   handle_simcalls()
6:   (t, events) ← models_solve()
7:   \( P_t \leftarrow \text{proc_to_wake}(events) \)
8: end while
```
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 *simcall* to request creation

```plaintext
1: $t \leftarrow 0$
2: $P_t \leftarrow P$
3: while $P_t \neq \emptyset$ do
4: parallel_schedule($P_t$)
5: handle_simcalls()
6: $(t, events) \leftarrow models\_solve()$
7: $P_t \leftarrow proc\_to\_wake(events)$
8: end while
```

- 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

![Logical View](image1)

![Ideal Algorithm](image2)

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 of
- 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>Tool</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, sequential)</td>
<td>10k</td>
<td>130s</td>
</tr>
<tr>
<td>SimGrid (constant, sequential)</td>
<td>300k</td>
<td>32mn</td>
</tr>
<tr>
<td>SimGrid (2M)</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 18 KiB per process
  
  (User stack: 12 KiB)

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 + Amdahl 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 on Parallel Simulation

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