

# SimGrid Kernel 101

## Introducing the SimGrid Kernel

Da SimGrid Team

June 13, 2012



# About this Presentation

## Goals and Contents

- ▶ Present Simix, the simulation kernel of SimGrid
- ▶ Show some more details about where our performance comes from

## 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
- ▶ Retrieve them from <http://simgrid.gforge.inria.fr/101>

# 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

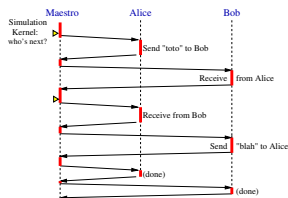
## Example of user code to execute

Alice

```
Send "toto" to Bob
Listen from Bob
```

Bob

```
Listen from Alice
Send "blah" to Alice
```



# SimGrid Internals in a Nutshell

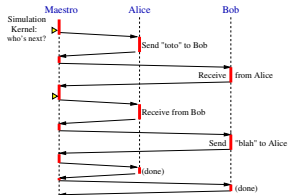
## Example of user code to execute

Alice

```
Send "toto" to Bob
Listen from Bob
```

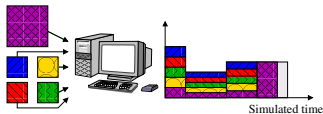
Bob

```
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 Internals in a Nutshell

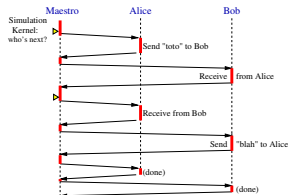
## Example of user code to execute

Alice

```
Send "toto" to Bob
Listen from Bob
```

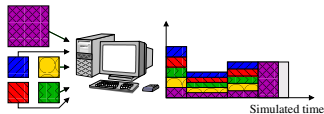
Bob

```
Listen from Alice
Send "blah" to Alice
```



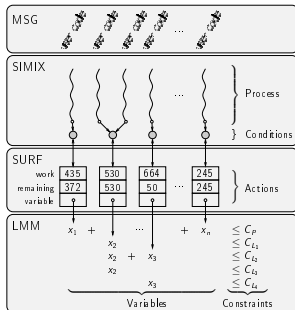
## SimGrid internal Main Loop

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 syntactic sugar
- ▶ **Simix**: Processes, synchro (SimPosix)
- ▶ **SURF**: Resources usage interface
- ▶ **Models**: Action completion computation



# Introduction Example

**function P1**

```
//Compute...  
  Send()  
  ...
```

**end function**

**function P2**

```
//Compute...  
  Recv()  
  ...
```

**end function**

*time*  $\leftarrow 0$

*P<sub>time</sub>*  $\leftarrow \{P_1, P_2\}$

**while** *P<sub>time</sub>*  $\neq \emptyset$  **do**

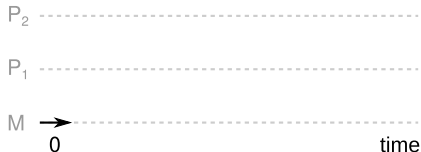
schedule(*P<sub>time</sub>*)

*time*  $\leftarrow$  solve(&*done\_actions*)

*P<sub>time</sub>*  $\leftarrow$  proc\_unblock(*done\_actions*)

**end while**

SimGrid's Main Loop



# Introduction Example

**function P1**

```
//Compute...  
  Send()  
  ...
```

**end function**

**function P2**

```
//Compute...  
  Recv()  
  ...
```

**end function**

$time \leftarrow 0$

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

**while**  $P_{time} \neq \emptyset$  **do**

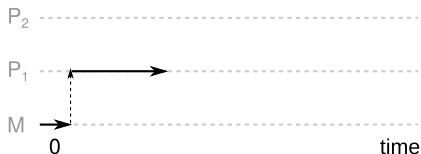
$\text{schedule}(P_{time})$

$time \leftarrow \text{solve}(\&done\_actions)$

$P_{time} \leftarrow \text{proc\_unblock}(done\_actions)$

**end while**

SimGrid's Main Loop



# Introduction Example

**function P1**

```
//Compute...  
  Send()  
  ...
```

**end function**

**function P2**

```
//Compute...  
  Recv()  
  ...
```

**end function**

$time \leftarrow 0$

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

**while**  $P_{time} \neq \emptyset$  **do**

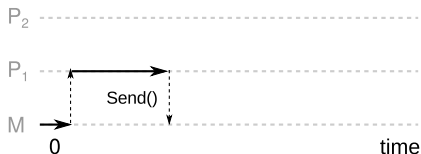
$\text{schedule}(P_{time})$

$time \leftarrow \text{solve}(\&done\_actions)$

$P_{time} \leftarrow \text{proc\_unblock}(done\_actions)$

**end while**

SimGrid's Main Loop



# Introduction Example

**function P1**

```
//Compute...  
  Send()  
  ...
```

**end function**

**function P2**

```
//Compute...  
  Recv()  
  ...
```

**end function**

$time \leftarrow 0$

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

**while**  $P_{time} \neq \emptyset$  **do**

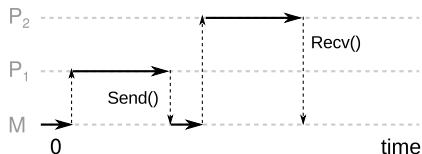
$\text{schedule}(P_{time})$

$time \leftarrow \text{solve}(\&done\_actions)$

$P_{time} \leftarrow \text{proc\_unblock}(done\_actions)$

**end while**

SimGrid's Main Loop



# Introduction Example

**function P1**

```
//Compute...  
  Send()  
  ...
```

**end function**

**function P2**

```
//Compute...  
  Recv()  
  ...
```

**end function**

$time \leftarrow 0$

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

**while**  $P_{time} \neq \emptyset$  **do**

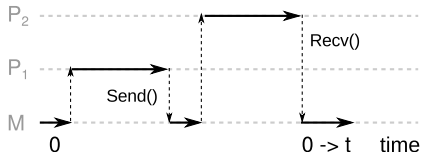
$schedule(P_{time})$

$time \leftarrow solve(\&done\_actions)$

$P_{time} \leftarrow proc\_unblock(done\_actions)$

**end while**

SimGrid's Main Loop



# Introduction Example

**function P1**

```
//Compute...  
Send()  
...
```

**end function**

**function P2**

```
//Compute...  
Recv()  
...
```

**end function**

$time \leftarrow 0$

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

**while**  $P_{time} \neq \emptyset$  **do**

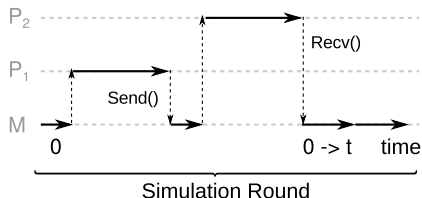
  schedule( $P_{time}$ )

$time \leftarrow \text{solve}(\&done\_actions)$

$P_{time} \leftarrow \text{proc\_unblock}(done\_actions)$

**end while**

SimGrid's Main Loop



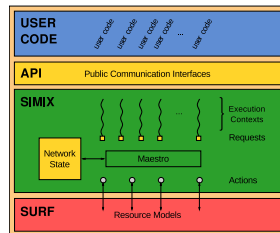
# Simix as an OS (Operating Simulator)

## 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

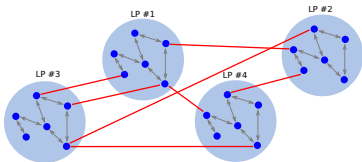
## PlanetSim

- ▶ 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  $\leadsto$  320,000 requests
- ▶ The results are impressive at first glance
  - ▶ 4h10 using two Logical Processes: only 1h06 using 16 LPs
  - ▶ Speedup of 4 using 8 times more resources, that's 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

<b>Simulation Workload</b>	<ul style="list-style-type: none"><li>▶ Granularity, Communication Pattern</li><li>▶ Events population, probability &amp; delay</li><li>▶ #simulation objects, #processors</li></ul>
<b>Simulation Engine</b>	<ul style="list-style-type: none"><li>▶ Parallel protocol, if any:<ul style="list-style-type: none"><li>– Conservative (lookahead, ...)</li><li>– Optimistic (state save &amp; restore, ...)</li></ul></li><li>▶ Event list mgnt, Timing model, ...</li></ul>
<b>Execution Environment</b>	<ul style="list-style-type: none"><li>▶ OS, Programming Language (C, Java, ...), Networking Interface (MPI, ...)</li><li>▶ Hardware aspects (CPU, mem., net)</li></ul>

Classical Parallel Simulation Schema  
[Balakrishnan *et al*]

<b>Simulation Workload</b>	User Code
	Virtualization Layer
	Networking Models
<b>Simulation Engine</b>	
<b>Execution Environment</b>	

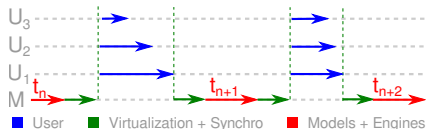
Layered View of  
Dist. App. Simulators

- ▶ The **classical approach** is to distribute the Simulation Engine entirely
- ▶ Hard issues: conservatives  $\leadsto$  too few parallelism; optimistic  $\leadsto$  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



Simulation Workload	User Code
	Virtualization Layer
	Networking Models
Simulation Engine	
Execution Environment	

## Understanding the trade-off

- ▶ Sequential time:  $\sum_{SR} (engine + model + virtu + user)$
- ▶ Classical schema:  $\sum_{SR} \left( \max_{i \in LP} (engine_i + model_i + virtu_i + user_i) + proto \right)$
- ▶ Proposed schema:  $\sum_{SR} \left( engine + model + \max_{i \in WT} (virtu_i + user_i) + sync \right)$
- ▶ 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...**

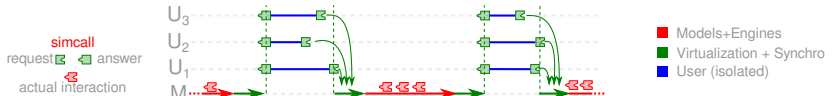
# Enabling Parallel Simulation of Dist.Apps

## 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  $\leadsto$  **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



```
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:    $(t, \text{events}) \leftarrow \text{models\_solve}()$ 
7:    $P_t \leftarrow \text{proc\_to\_wake}(\text{events})$ 
8: end while
```

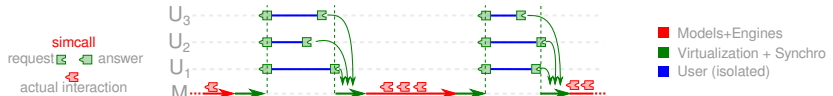
# Enabling Parallel Simulation of Dist.Apps

## 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



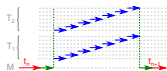
```
1:  $t \leftarrow 0$ 
2:  $P_t \leftarrow P$ 
3: while  $P_t \neq \emptyset$  do
4:   parallel_schedule( $P_t$ )
5:   handle_simcalls()
6:    $(t, \text{events}) \leftarrow \text{models\_solve}()$ 
7:    $P_t \leftarrow \text{proc\_to\_wake}(\text{events})$ 
8: end while
```

- ▶ 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



Ideal Algorithm

## Reducing Synchronization Costs

- ▶ Inter-thread synchronization achieved through system calls (of real OS)
- ▶ Costs of **syscalls** are critical to performance  $\leadsto$  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)

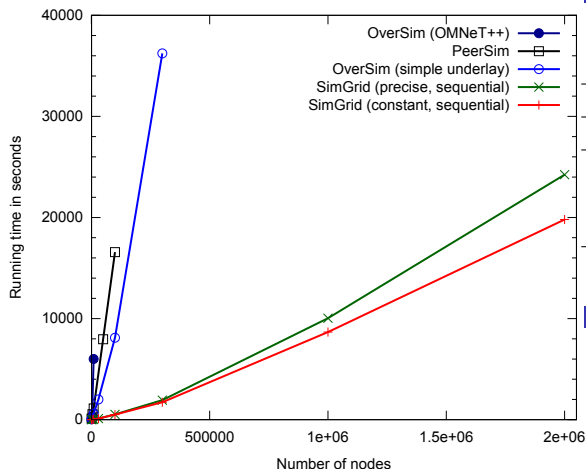
## Measuring intrinsic synchronization costs

- ▶ **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



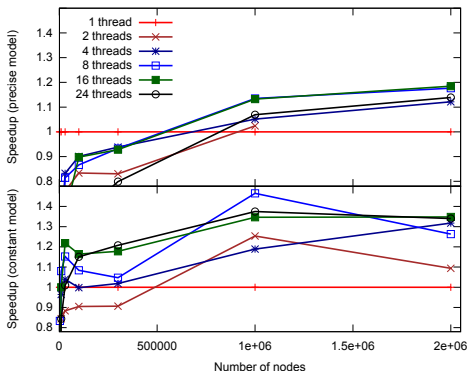
	Size	Time
Omnet++	10k	1h40
PeerSim	100k	4h36
OverSim	300k	10h
SG, precise	10k	130s
	300k	32mn
	2M	6h23
SG, simple	2M	5h30

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

Model	4 threads	8 th.	16 th.	24 th.
Precise	0.28	0.15	0.07	0.05
Constant	0.33	0.16	0.08	0.06

- ▶ Baaaaaad 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  $\neq$  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