Performance & Correctness Assessment of Distributed Systems

PhD Defense of Cristian Rosa

Université Henri Poincaré – Nancy 1, France

24/10/2011
Introduction

Distributed System

A system that consists of multiple autonomous computing entities that interact towards the solution of a common goal.

Some Examples:
- Facebook: 500 Millions of users
- eBay: idem Facebook + Money
- eMule, BitTorrent, Amazon

Distributed Systems are critical to many applications!
Complexity of Distributed Systems

- **Grid Computing**
  - Infrastructure for computational science
  - Heterogeneous computing resources, static network topology
  - Main issue: process as much jobs as possible
  - Example: LHC Computing Grid – 500K cores, 140 computing centers in 35 countries

- **Peer-to-Peer Systems**
  - Exploit resources at network edges
  - Heterogeneous computing resources, dynamic network topology
  - Main issue: deal with intermittent connectivity, anonymity
  - Example: BitTorrent, SETI@Home
Challenges of Distributed Systems

- **Lack of knowledge about the global state**
  ▸ control decisions based only on local knowledge

- **Lack of common time reference**
  ▸ impossible to order the events of different entities

- **Non-determinism**
  ▸ evolution of all non-local state impossible to predict

In general, distributed systems are badly understood!
Assessing Distributed Systems

Distributed Systems characteristics are hard to assess:

- **Performance**
  Must maximize it, but definition differs between systems

- **Correctness**
  Hard to find and reproduce bugs, lack of guarantees
Assessing the Performance of Distributed Systems

- **Theoretical approach**
  - 😊 absolute answer
  - 😞 often simplistic, time consuming, experienced users

- **Real executions**
  - 😊 accurate, real experimentation bias
  - 😞 difficult to instrument, limited to a few scenarios

- **Simulations**
  - 😊 relative simple to use, many scenarios, fast
  - 😞 lack of real experimentation effects, requires validated models
Assessing the Correctness of Distributed Systems

- **Direct Experimentation**
  - 😊 no false positives
  - 😞 very limited, bugs hard to find and reproduce, difficult to instrument

- **Proofs**
  - 😊 complete guarantee of correctness independent of system size
  - 😞 non automatic, time consuming, experienced users

- **Model Checking**
  - 😊 automatic, relatively simple to use, counter-examples
  - 😞 state spaces grows exponentially with system size
Comparison of Methodologies

<table>
<thead>
<tr>
<th></th>
<th>Execution</th>
<th>Simulation</th>
<th>Proofs</th>
<th>Model checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Assessment</td>
<td>☹</td>
<td>☺☺</td>
<td>☹</td>
<td>☹</td>
</tr>
<tr>
<td>Experimental Bias</td>
<td>☹☺</td>
<td>☺</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Experimental Control</td>
<td>☹</td>
<td>☺☺</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Correctness Verification</td>
<td>☹☺</td>
<td>☹</td>
<td>☺☺</td>
<td>☺</td>
</tr>
<tr>
<td>Ease of use</td>
<td>☹</td>
<td>☺☺</td>
<td>☹</td>
<td>☺</td>
</tr>
</tbody>
</table>

Simulation and Model Checking complement each other:

- Simulation to assess the performance
- Model Checking to verify correctness
- Both run automatically
- Low usability barrier

Often, simulators and model checkers require different system descriptions
Model Checking idea:
- Exhaustive exploration of state space
- Check validity of every state

In a distributed setting:
- Run all interleavings of communications
  - interception of communication events
  - control the events’ happening ordering
Model Checking Versus Simulation

Simulation idea:
- The platform is an additional parameter
- Compute a single run of the system

In a distributed setting:
- Always the same trace with timings
- interception of communications events
- delay the events according to the models
Simulation idea:
- The platform is an additional parameter
- Compute a single run of the system

In a distributed setting:
- Always the same trace with timings
- interception of communications events
- delay the events according to the models
Objective
Develop the theory and tools for the efficient performance and correctness assessment of distributed systems.

Approach
Make model checking possible in the SimGrid simulation framework. Not reinvent the wheel: SimGrid is fast, scalable, and validated.

Contributions
- Correctness assessment:
  - SimGridMC: a dynamic verification tool for distributed systems
  - Custom reduction algorithm to deal with state space explosion
- Performance assessment:
  - Parallelization of the simulation loop for CPU bound simulations
  - Criteria to estimate the potential benefit of parallelism
1 Introduction

2 Bridging Simulation and Verification
   - The SimGrid Framework
   - SIMIXv2.0
   - Experiments

3 SimGrid MC
   - Architecture
   - Coping With The State Explosion
   - Experiments

4 Parallel Execution
   - Architecture
   - Cost Analysis
   - Experiments

5 Conclusion and Future Work
The SimGrid Framework

A collection of tools for the simulation of distributed computer systems

Main characteristics:
- Designed as a scientific measurement tool (validated models)
- It simulates real programs (written in C and Java among others)

Experimental workflow:

Distributed System Implementation + Resource Models + Experimental Setup = Scientific Results
A Simulation Example

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

function P2
    //Compute...
    Recv()
    ...
end function
A Simulation Example

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

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

time ← 0
P_time ← \{P_1, P_2\}
while P_time \neq \emptyset do
  schedule(P_time)
  time ← solve(done_actions)
  P_time ← proc_unblock(done_actions)
end while

SimGrid’s Main Loop
A Simulation Example

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

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

time ← 0
P_time ← \{ P_1, P_2 \}
while \( P_{time} \neq \emptyset \) do
    schedule(\( P_{time} \))
    time ← solve(\&done_actions\)
    P_time ← proc_unblock(done_actions)
end while

SimGrid’s Main Loop

P_2

P_1

M 0 time
A Simulation Example

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
A Simulation Example

**SimGrid’s Main Loop**

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

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

- `time ← 0`
- `P_{time} ← \{P_1, P_2\}`
- `while P_{time} \neq \emptyset do`
  - `schedule(P_{time})`
  - `time ← solve(&done_actions)`
  - `P_{time} ← proc_unblock(done_actions)`
- `end while`

P1  +-----------------------------+
    |                             |
    v                             |
+-----------------------------+  P2
    |                             |
    v                             |
+-----------------------------+  M
    |                             |
    v                             |
+-----------------------------+  0

```
```
A Simulation Example

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

P2

P1

M

0

Send()

Recv()

time
A Simulation Example

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
A Simulation Example

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(\( P_{time} \))
  time ← solve(\&\ done\_actions\)
  \( P_{time} \leftarrow \) proc_unblock(\( done\_actions \))
end while

SimGrid’s Main Loop

P1
Send()
P2
Recv()
M
0
Simulation Round

0 → t

C. Rosa (UHP – Nancy 1) PhD Defense 24/10/2011 12 / 42
The Architecture

USER CODE

API
Network Applicative State

SIMIX

{Execution Contexts
Conditions

SURF
Resource Models
A Simulation Example in More Detail

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

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

time ← 0
P_{time} ← P
while P_{time} ≠ ∅ do
    schedule(P_{time})
    time ← solve(&done_actions)
    P_{time} ← proc_unblock(done_actions)
end while

SimGrid’s Main Loop
```

```
+---------------------+---------------------+
| P2                  | P1                  |
|                    |                      |
| P_{time}            | P_{time}            |
|                    |                      |
| Send()              | Send()              |
+---------------------+---------------------+

0 0 -> t

C. Rosa (UHP – Nancy 1) PhD Defense 24/10/2011 14 / 42
Limitations of the Architecture

This architecture not good enough:

- Late interception of network operations
- Lack of control over the network state
Limitations of the Architecture

This architecture not good enough:

- Late interception of network operations
  ➔ Lack of control over the network state

- User processes modify the shared state
  ➔ Parallel execution hard to achieve
  ➔ Renders reproducibility impossible
Limitations of the Architecture

This architecture not good enough:

- Late interception of network operations
  \implies Lack of control over the network state

- User processes modify the shared state
  \implies Parallel execution hard to achieve
  \implies Renders reproducibility impossible
Limitations of the Architecture

This architecture not good enough:

- Late interception of network operations
  → Lack of control over the network state

- User processes modify the shared state
  → Parallel execution hard to achieve
  → Renders reproducibility impossible
A New Architecture for SimGrid

SIMIXv2.0

A new virtualization module designed to overcome previous limitations.

(Joint work with Christophe Thiéry).

Inspired by operating system design concepts:

- **Strict layered design:**
  - Processes, IPC, and synchronization primitives
  - Encapsulated shared state

- **System call like interface:**
  - Interaction with platform mediated through “requests”
  - The simulator answers the requests
The New Architecture

USER CODE

API
Network Applicative State

SIMIX

SURF
Resource Models

USER CODE

API
Public Communication Interfaces

SIMIX
Network State
Maestro

SURF
Resource Models

Execution Contexts
Requests
Actions
A Simulation Example with SIMIXv2.0

SIMIX Main Loop

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

SIMIXv2.0 Main Loop

\[
\begin{align*}
time & \leftarrow 0 \\
P_{time} & \leftarrow \{P_1, P_2\} \\
\text{while } P_{time} \neq \emptyset & \text{ do} \\
\quad & \text{schedule}(P_{time}) \\
\quad & \text{handle\_requests()} \\
\quad & time \leftarrow \text{solve}(\&done\_actions) \\
\quad & P_{time} \leftarrow \text{proc\_unblock}(done\_actions) \\
\text{end while}
\end{align*}
\]
Master-Slaves experiment:

- SIMIXv2.0 14% faster on average (with no loses)
- Gains due to: simplification of code, less dynamic data
1 Introduction

2 Bridging Simulation and Verification
   - The SimGrid Framework
   - SIMIXv2.0
   - Experiments

3 SimGrid MC
   - Architecture
   - Coping With The State Explosion
   - Experiments

4 Parallel Execution
   - Architecture
   - Cost Analysis
   - Experiments

5 Conclusion and Future Work
Overview of SimGridMC

SimGridMC
A dynamic verification tool for SimGrid programs.

Design Goals:
- Verification of *unmodified* SimGrid programs
- Find bugs triggered by program’s nondeterministic behavior
- Designed as a debugging tool
- Capable of handling nontrivial programs
- Simple to use by SimGrid users
An Example of Bug

Message Delivery Order Bug

P1(){
    Send(1,P3);
}

P2(){
    Send(2,P3);
}

P3(){
    Recv(&x,*);
    Recv(&y,*);
    ASSERT(x<y);
}
An Example of Bug

Message Delivery Order Bug

\begin{verbatim}
P1(){
    Send(1, P3);
}

P2(){
    Send(2, P3);
}

P3(){
    Recv(&x,*);
    Recv(&y,*);
    ASSERT(x<y);
}
\end{verbatim}
An Example of Bug

Message Delivery Order Bug

P1()
  Send(1, P3);
}

P2()
  Send(2, P3);
}

P3()
  Recv(&x,*);
  Recv(&y,*);
  ASSERT(x<y);
}
Main characteristics of SimGridMC:

- **Exploration:**
  - Explicit-state
  - Verification of local assertions
  - It actually executes the code

- **Roll-backs:**
  - Stateless approach (replay)
  - No visited state storing nor hashing

- **Reduction techniques to cope with state space explosion**
The Exploration Loop

Explored Interleavings: \( \langle a, b, c, d \rangle, \langle a, b, d, c \rangle, \ldots \)
The State Explosion Problem

P1()
  Send(&x,P2);
}
P2()
  Recv(&y,P1);
}
P3()
  Send(&r,P4);
}
P4()
  Recv(&q,P3);
}

They are all the same happened-before relation!
To explore different partial orders we must interleave dependent transitions

\[ D(t_a, t_b) = \neg I(t_a, t_b) \]

How do we get the predicate \( D \)?

- Using the semantics of the transitions
- "Independence theorems" for each pair of transitions
No communication API in SimGrid had a formal specification. Manual specification required (tedious, time consuming)
No communication API in SimGrid had a formal specification. Manual specification required (tedious, time consuming)

The solution explored in this thesis:
- A core set of four basic networking primitives (SIMIXv2.0 IPC)
- User-level APIs written on top of these
- Full formal specification of their semantics (in TLA+)
- Theorems of independence between certain primitives
- State-space exploration at primitives’ level
The Communication Model of the IPC

Communication model based on mailboxes:
- processes post send/receive request into mailboxes
- requests queued/matched in FIFO order

Four primitives:
- `Send` – asynchronous send request
- `Recv` – asynchronous receive request
- `WaitAny` – block until completion of a communication
- `TestAny` – test for completion without blocking

Can express large parts of MPI, GRAS (socket), and MSG (CSP)
Independence Theorems

- 6 theorems of the form:

\[ I(A, B) \triangleq \text{Enabled } A \land \text{Enabled } B \Rightarrow \land \ A \Rightarrow (\text{Enabled } B)' \]
\[ \land \ B \Rightarrow (\text{Enabled } A)' \]
\[ \land \ A \cdot B \equiv B \cdot A \]

- Proofs expand definitions and use commutativity

- The following actions are independent:
  - Local actions with any other action
  - Send and Recv
  - Two Send or two Recv in different mailboxes
  - Wait or Test for the same communication
Processes multiple of 3 receive a message from their next two successors

```c
if (rank % 3 == 0) {
    MPI_Recv(&val1, MPI_ANY_SOURCE);
    MPI_Recv(&val2, MPI_ANY_SOURCE);
    MC_assert(val1 > rank);
    MC_assert(val2 > rank);
} else {
    MPI_Send(&rank, (rank / 3) * 3);
}
```

<table>
<thead>
<tr>
<th>#P</th>
<th>Without reductions</th>
<th>With reductions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Time</td>
</tr>
<tr>
<td>3</td>
<td>520</td>
<td>0.247 s</td>
</tr>
<tr>
<td>6</td>
<td>&gt;10560579</td>
<td>&gt;1 h</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chord Experiments

Chord P2P DHT protocol

SimGrid implementation:
- 500 lines of C (MSG interface)
- Spotted a bug in big instances
Chord Experiments

Chord P2P DHT protocol

SimGrid implementation:
- 500 lines of C (MSG interface)
- Spotted a bug in big instances

SimGrid MC with two nodes:
- DFS: 15600 states - 24s
- DPOR: 478 states - 1s
- Simple Counter-example!
- One line fix
1. Introduction

2. Bridging Simulation and Verification
   - The SimGrid Framework
   - SIMIXv2.0
   - Experiments

3. SimGrid MC
   - Architecture
   - Coping With The State Explosion
   - Experiments

4. Parallel Execution
   - Architecture
   - Cost Analysis
   - Experiments

5. Conclusion and Future Work
Motivation of Parallelization Work

- Scaling-up memory bound simulations; easy $\rightsquigarrow$ more RAM
- Speedup CPU bound simulations; difficult
  - Processors increase almost only in parallel power (cores)
  - The simulation problem is really hard to parallelize

We envision two scenarios:
- Applications with processes that perform big computations
- Applications with a large amount of processes
Avoiding out of order events:
- **Conservative**: no out of order event can happen
  - 😞 very platform dependent (latency)
- **Optimistic**: rewind to a consistent state on out of order events
  - 😞 expensive checkpoints
Parallelization of the Simulation Loop

Our Approach

Keep the simulation sequential but parallelize some steps of the main loop
Parallelization of the Simulation Loop

Our Approach

Keep the simulation sequential but parallelize some steps of the main loop

Parallel Main Loop

\[
\begin{align*}
t & \leftarrow 0 \\
P_{time} & \leftarrow P \\
\text{while } P_{time} \neq \emptyset & \text{ do} \\
& \quad \text{parallel_schedule}(P_{time}) \\
& \quad \text{handle_requests()} \\
& \quad t \leftarrow \text{surf.solve}(& \text{done.actions}) \\
& \quad P_{time} \leftarrow \text{process.unblock}(\text{done.actions}) \\
\text{end while}
\end{align*}
\]

This is possible thanks to the shared state encapsulation of SIMIXv2.0
### Cost of parallel execution

**Sequential execution**

\[
\sum\limits_{t_i} \left( C_{surf}(R, M) + C_{smx}(|P_{t_i}|) + C_{usr}(P_{t_i}) \right)
\]

**Parallel execution**

\[
\sum\limits_{t_i} \left( C_{surf}(R, M) + C_{smx}(|P_{t_i}|) + C_{thr}(|T|) + \max_{w \in T} \left( C_{usr}(P^w_{t_i}) \right) \right)
\]
Good Parallelization Scenarios

\[ K \cdot C_{thr}(|T|) + \max_{w \in T}(C_{usr}(P^w_t)) < C_{usr}(P_{t_i}) \]
Good Parallelization Scenarios

\[ K \cdot C_{thr}(|T|) + \max_{w \in T}(C_{usr}(P^w_t)) < C_{usr}(P_t) \]

This can happen when

\[ \sum_{p \in P_t} C(p) \to \infty \]
Experimental Results I

Good scenario: Parallel Matrix Multiplication

- 9 nodes (3x3 grid)
- Matrices of size 1500 (doubles)
- Simulation results (LV08):
  - Sequential execution: 31s
  - Parallel execution (4T): 11s (speedup = x2.8)
Experimental Results II

Chord: SimGrid vs. OverSim

300,000 nodes
- OverSim (simple): 10h
- SimGrid (LV08): 38 min

2,000,000 nodes (SG only)
- Seq (LV08): 7h40
- 24T (LV08): 6h55 (x1.30)
- Seq (Const): 5h42
- 24T (Const): 4h (x1.45)
Conclusions I

Correctness Assessment:

- Novel approach that integrates a simulator and a model checker
- SimGridMC a model checker for unmodified distributed C programs
- Effective state reduction with support for multiple APIs
- Capable of finding bugs in realistic programs like Chord
Conclusions II

Performance Assessment:

- Classical parallelization approaches are not well suited

- Alternative approach: parallelize user processes

- Cost analysis of the approach

- SimGrid is scalable, accurate, and fast
Future Work

Correctness assessment:
- Implement and evaluate a stateful exploration
- Add support for *liveness* properties verification
- Experiment with a hybrid roll-back mechanism (checkpoint + replay)

Performance assessment:
- Simulation and model checking combined (performance checking)
- Parallelize other steps of the simulation loop
- Refinement of the communication primitives
S. Merz, M. Quinson, and C. Rosa.
Simgrid MC: Verification Support for a Multi-api Simulation Platform.

C. Rosa, S. Merz, and M. Quinson.
A Simple Model of Communication APIs – Application to Dynamic Partial-order Reduction.
High Performance Computing
- Lead CS and IT world’s research
- Homogeneous nodes with many cores, high-speed local links
- Main issue: do the biggest possible numerical simulations
- Example: K Computer – 548,352 Cores, Riken, Japan

Cloud Computing
- Large infrastructures underlying commercial Internet
- Heterogeneous computing resources, static network topology
- Main issue: optimize costs, keep up with the load
- Example: Amazon’s Cloud
Asynchronous Communication Bug

P1()
    c = iSend("ok",P2);
    Wait(c);
}

P2()
    c1 = iRecv(&buff,P1);
    c2 = iSend(&buff,P2);
    Wait(c1);
    Wait(c2);
}

P3()
    c = iRecv(&buff,P2);
    Wait(c);
    ASSERT(buff=="ok");
}
Asynchronous Communication Bug

P1()
    c = iSend("ok",P2);
    Wait(c);
}

P2()
    c1 = iRecv(&buff,P1);
    c2 = iSend(&buff,P2);
    Wait(c1);
    Wait(c2);
}

P3()
    c = iRecv(&buff,P2);
    Wait(c);
    ASSERT(buff=="ok");
}
Asynchronous Communication Bug

P1()
    c = iSend("ok",P2);
    Wait(c);
}

P2()
    c1 = iRecv(&buff,P1);
    c2 = iSend(&buff,P2);
    Wait(c1);
    Wait(c2);
}

P3()
    c = iRecv(&buff,P2);
    Wait(c);
    ASSERT(buff=="ok");
}
The Model

The states are the global states of the system
The transitions are the communication actions
The exploration consists of interleaving the communication actions
SimGridMC’s Architecture

The Architecture

Communication API

P₀  P₁  …  Pₙ

MC

NETWORK

INITIAL STATE
Approximating Dependency

D can be over-approximated by a $D'$ such that

$$D(A, B) \Rightarrow D'(A, B)$$

If we don’t know if $l(t_i, t_j)$ holds we assume $D'(t_i, t_j)$ (for soundness).
SimGridMC
Independence Theorems

**Theorem**

Any two *Send* and *Recv* transitions are independent.

\[
\forall A, B \in \text{Proc}, rdv_1, rdv_2 \in \text{RdV}, \&x, \&y \in \text{Addr}, c_1, c_2 \in \text{Addr} : \\
I(\text{Send}(A, rdv_1, \&x, c_1), \text{Recv}(B, rdv_2, \&y, c_2))
\]
SimGridMC
Independence Theorems

Theorem

Any two Send and Recv transitions are independent.

∀A, B ∈ Proc, rdv₁, rdv₂ ∈ RdV, &x, &y ∈ Addr, c₁, c₂ ∈ Addr :
I(Send(A, rdv₁, &x, c₁), Recv(B, rdv₂, &y, c₂))
Theorem

Any two Send and Recv transitions are independent.

∀A, B ∈ Proc, rdv₁, rdv₂ ∈ RdV, &x, &y ∈ Addr, c₁, c₂ ∈ Addr :
I(Send(A, rdv₁, &x, c₁), Recv(B, rdv₂, &y, c₂))
Theorem

Any two Send andRecv transitions are independent.

∀A, B ∈ Proc, rdv₁, rdv₂ ∈ RdV, &x, &y ∈ Addr, c₁, c₂ ∈ Addr :
I(Send(A, rdv₁, &x, c₁), Recv(B, rdv₂, &y, c₂))
Theorem

Any two Send and Recv transitions are independent.

\[
\forall A, B \in \text{Proc}, \ rdv_1, \ rdv_2 \in \text{RdV}, \ &x, \ &y \in \text{Addr}, \ c_1, c_2 \in \text{Addr} : \ I(\text{Send}(A, \ rdv_1, \ &x, \ c_1), \ \text{Recv}(B, \ rdv_2, \ &y, \ c_2))
\]
Theorem

Any two Send and Recv transitions are independent.

$$\forall A, B \in \text{Proc}, rdv_1, rdv_2 \in \text{RdV}, \& x, \& y \in \text{Addr}, c_1, c_2 \in \text{Addr} :$$

$$I(\text{Send}(A, rdv_1, \& x, c_1), \text{Recv}(B, rdv_2, \& y, c_2))$$
Theorem

Any two Send and Recv transitions are independent.

∀A, B ∈ Proc, rdv₁, rdv₂ ∈ RdV, &x, &y ∈ Addr, c₁, c₂ ∈ Addr :
I(Send(A, rdv₁, &x, c₁), Recv(B, rdv₂, &y, c₂))
Theorem

Any two Send and Recv transitions are independent.

\[ \forall A, B \in \text{Proc}, \text{rdv}_1, \text{rdv}_2 \in \text{RdV}, \& \text{x, \& y} \in \text{Addr}, c_1, c_2 \in \text{Addr} : I(\text{Send}(A, \text{rdv}_1, \& \text{x, c}_1), \text{Recv}(B, \text{rdv}_2, \& \text{y, c}_2)) \]
Any two Send and Recv transitions are independent.

∀A, B ∈ Proc, rdv₁, rdv₂ ∈ RdV, &x, &y ∈ Addr, c₁, c₂ ∈ Addr :
I(Send(A, rdv₁, &x, c₁), Recv(B, rdv₂, &y, c₂))
Parallelization of the Simulation Loop

Classical Parallelization Approaches

There are two classical parallelization approaches:

- **Space Decomposition**

  - Multiple time lines

  Risk of out of order events
  - Conservative: advance when no event out of order can happen
  - Optimistic: rewind to a consistent state when out of order events happen

  Time divided in intervals
  - Intervals simulated in parallel
  - Must guess initial states
  - Re-computation needed when states do not match
Parallelization of the Simulation Loop
Classical Parallelization Approaches

There are two classical parallelization approaches:

- **Space Decomposition**
  - Multiple time lines
  - Risk of out of order events
  - Conservative: advance when no event out of order can happen
  - Optimistic: rewind to a consistent state when out of order events happen
Parallelization of the Simulation Loop
Classical Parallelization Approaches

There are two classical parallelization approaches:

- **Space Decomposition**
  - Multiple time lines
  - Risk of out of order events
  - Conservative: advance when no event out of order can happen
  - Optimistic: rewind to a consistent state when out of order events happen

- **Time Decomposition**
  - Time divided in intervals
  - Intervals simulated in parallel
There are two classical parallelization approaches:

- **Space Decomposition**
  - Multiple time lines
  - Risk of out of order events
  - Conservative: advance when no event out of order can happen
  - Optimistic: rewind to a consistent state when out of order events happen

- **Time Decomposition**
  - Time divided in intervals
  - Intervals simulated in parallel
  - Must guess initial states
  - Re-computation needed when states do not match
Resolution of the Model

A simulation defines a discretization of the simulated time:

\[ T_{M,R} : E \rightarrow [0, t], \text{ with } t \in \mathbb{R} \]

Each event has a timestamp that is computed using the resource models.

Resolution (\( \varepsilon \))

The minimal time increment possible between two timestamps.
The model resolution $\varepsilon$ has an impact on the size of $P_{t_i}$

\[
\begin{align*}
|P_{t_1}| &= 1, & |P_{t_2}| &= 1 \\
|P_{t_3}| &= 2, & |P_{t_4}| &= 1 \\
|P_{t_5}| &= 1, & |P_{t_6}| &= 1
\end{align*}
\]
The model resolution $\varepsilon$ has an impact on the size of $P_{t_i}$.

<table>
<thead>
<tr>
<th>$P_{t_1}$</th>
<th>$P_{t_2}$</th>
<th>$P_{t_3}$</th>
<th>$P_{t_4}$</th>
<th>$P_{t_5}$</th>
<th>$P_{t_6}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Higher resolution

Lower resolution
Impact of the resolution $\varepsilon$ on $|P_{t_i}|$

- Chord 100,000 nodes
- Simulation of 1000s
- 25,000,000 messages exchanged

<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>$10^{-5}$</th>
<th>$10^{-3}$</th>
<th>$10^{-1}$</th>
<th>Constant network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average $</td>
<td>P_{t_i}</td>
<td>$</td>
<td>10</td>
<td>44</td>
</tr>
</tbody>
</table>
Chord: SimGrid vs. OverSim

Chord 2,000,000 nodes

- $\varepsilon = 10^{-1}$
- Sequential execution: 8h15
- Parallel execution: 7h15
- speedup = 1.13 (24 threads)