Faster Simulation of (Coloured) Petri Nets Using Parallel Computing

Jordan de la Houssaye Franck Pommereau

IBISC, University of Évry / Paris-Saclay

PETRI NETS 2017 — Zaragoza — June 28th 2017

Outline

Introduction

Medusa

Formal analysis

Benchmarks

Conclusion

Faster fast simulation

Compute traces of Petri nets faster

- for statistical analysis
- to use a model as a prototype. . .
- . . . or even as an implementation

Exploit parallelism

- multi-core CPUs
- mutli-CPU architectures
- distributed computing (clusters)

Contribution

- a parallel algorithm
- formal analysis
- benchmark

Faster fast simulation

Compute traces of Petri nets faster

- for statistical analysis
- to use a model as a prototype...
- ... or even as an implementation

Exploit parallelism

- multi-core CPUs
- mutli-CPU architectures
- distributed computing (clusters)

Contribution

- a parallel algorithm
- formal analysis
- benchmark

Faster fast simulation

Compute traces of Petri nets faster

- for statistical analysis
- to use a model as a prototype. . .
- . . . or even as an implementation

Exploit parallelism

- multi-core CPUs
- mutli-CPU architectures
- distributed computing (clusters)

Contribution

- a parallel algorithm
- formal analysis
- benchmark

Outline

Introduction

Medusa

Formal analysis

Benchmarks

Conclusion

Concurrency model

- simulation is sequential on single CPU
  - uses cooperative multitasking (no locks needed)
  - “**call** fun$(\cdots)$”
    - starts “fun$(\cdots)$” in a new thread
    - does not give control (actual start is delayed)
  - “**rpc** fun$(\cdots)$”
    - remotely calls an instance of “fun$(\cdots)$”
    - gives control immediately, until a result is available
- remote procedure calls are realised on the other CPUs
  - in parallel to the simulation
  - implementation uses a (limited) pool of worker processes

Token flows

We computes successor markings through flows
= pairs of markings to add/remove from the current one

Concurrency model

- simulation is sequential on single CPU
  - uses cooperative multitasking (no locks needed)
  - "**call** fun($\cdots$)"
    - starts "fun($\cdots$)" in a new thread
    - does not give control (actual start is delayed)
  - "**rpc** fun($\cdots$)"
    - remotely calls an instance of "fun($\cdots$)"
    - gives control immediately, until a result is available
- remote procedure calls are realised on the other CPUs
  - in parallel to the simulation
  - implementation uses a (limited) pool of worker processes

Token flows

We computes successor markings through flows
= pairs of markings to add/remove from the current one

Players and teams

- each transition is simulated by a player
- each player has
    - a team: the other players with which it has a conflict
    - an output: the other players for which it may produce tokens

```
struct player :
    trans : transition
    team  : set[player]
    out   : set[player]
    busy  : bool
    retry : bool
```

```
def startup (players) :
    run ← []
    for player in players :
        player.busy ← True
        call work(player, run)
```

Algorithm

```
def work (player, run) :
    player.retry ← False
    flows ← {f in rpc getflows(player.trans, run.last) | f.sub $\leq$ run.last}
    if player.retry and flows $= \emptyset$ :
        call work(player, run)
    elif flows $= \emptyset$ :
        player.busy ← False
    else :
        choose flow in flows
        append run.last $-$ flow.sub $+$ flow.add to run
        player.busy ← False
        for other in player.team $\cup$ player.out :
            if not other.busy :
                other.busy ← True
                call work(other, run)
            elif other.busy and other in player.out :
                other.retry ← True
```

Outline

- Introduction
- Medusa
- Formal analysis
- Benchmarks
- Conclusion

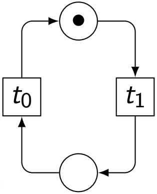
Model-net N_m models Medusa

- 4 transitions

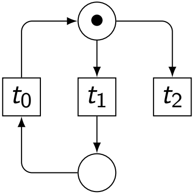
rpc	start of work, up to **rpc**
retry	return from **rpc** + **if** = player got no flows but has to retry
idle	return from **rpc** + **elif** = player got no flows and becomes idle
fire	return from **rpc** + **else** = player has at least one flow and can fire its transition

- 3 places
  1. players structures
  2. computed flows
  3. trace (only its latest marking)

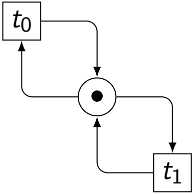
Simulated-nets N_s



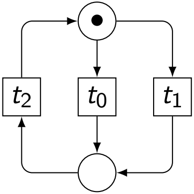
(1)



(2)



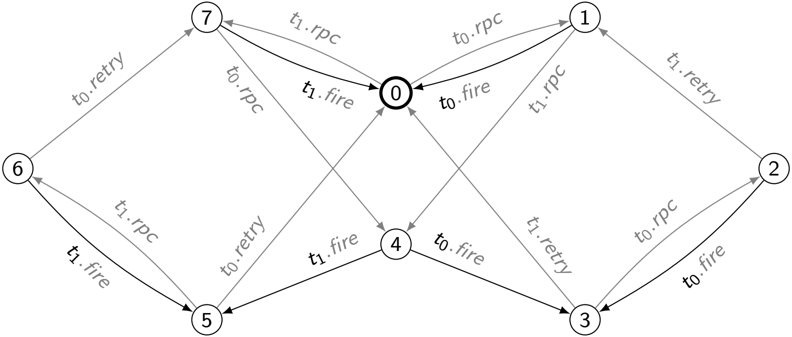
(3)



(4)

Marking graph G_m

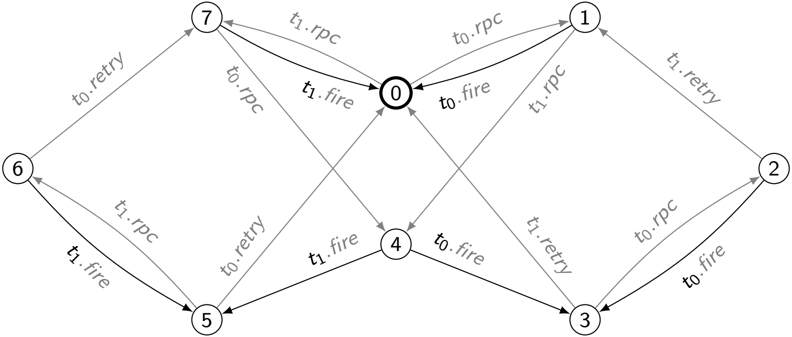
here for net (3)



- rpc, retry and idle are internal actions $\mapsto \tau$
- let G_m/τ be G_m in which gray arcs have been collapsed

Marking graph G_m

here for net (3)



- rpc, retry and idle are internal actions $\mapsto \tau$
- let G_m/τ be G_m in which gray arcs have been collapsed

Results

- G_m/τ always isomorphic to G_s (marking graph of N_s)
- G_s and G_m are weak bisimilar

- correctness: every run of the N_m is a correct run of N_s
- completeness: every run of N_s exists in N_m
- deadlock equivalence: N_m and N_s have the same deadlocks

- progression: no τ-loop in $G_m \Rightarrow$ a fire always eventually occurs

Outline

Introduction

Medusa

Formal analysis

Benchmarks

Conclusion

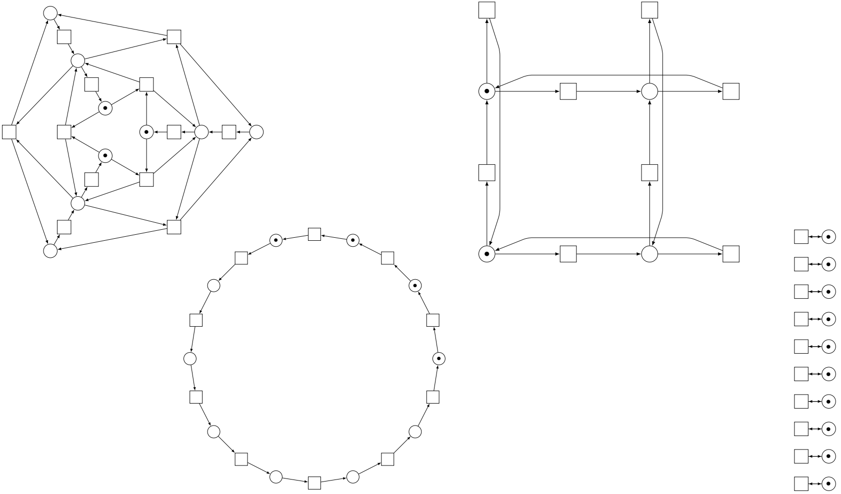
Prototype implementation

- Python
- SNAKES for Petri nets stuff
- gevent for cooperative multitasking
- less than 150 lines of code

For the benchmarks:

- 4 parametrised models (next slide)
- P/T nets + simulated colours (eat CPU for a given amount of time)

Models used



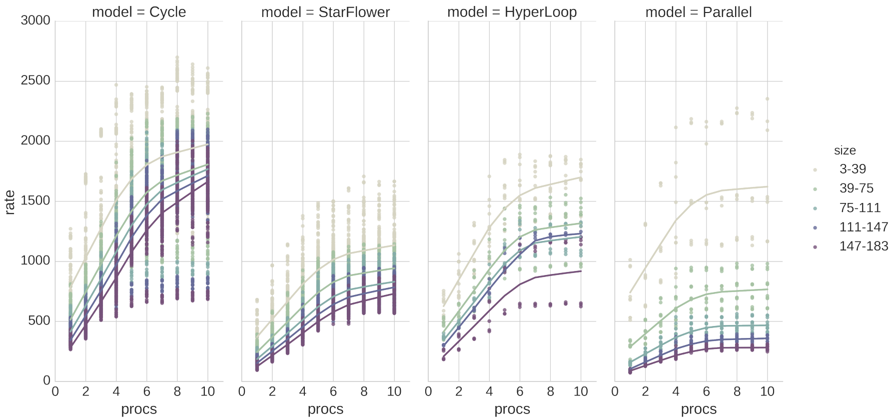
starflower

hyperloop

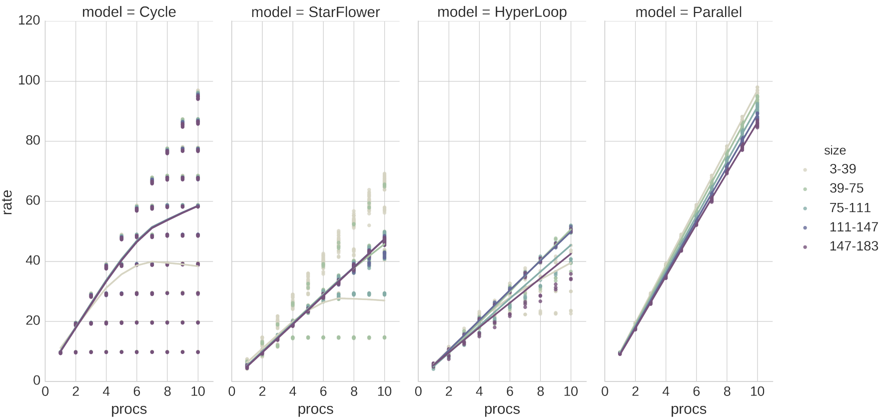
cycle

parallel

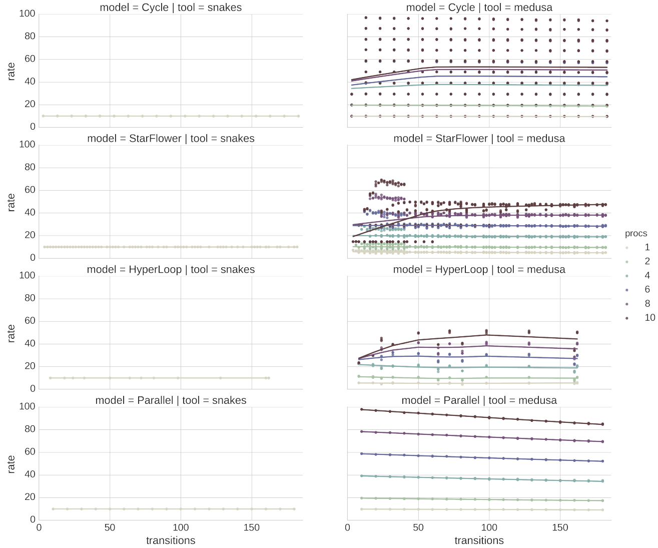
Medusa does not like uncoloured nets



But Medusa loves large coloured nets



Medusa efficiently eats your CPUs



Outline

Introduction

Medusa

Formal analysis

Benchmarks

Conclusion

Achievements and future work

Contribution

- simple yet non-trivial parallel simulation algorithm
- formally analysed
- portable design from multicore to clusters
- efficient implementation is attainable
- encouraging experimental results

Perspectives

- use Neco's compilation technique to improve RPC-side computation
- finer-grained algorithm (ex: compute one flow at a time)
- better analysis of the influence of colours
- investigate fairness more thoroughly (thanks to reviewer 1)
- formal proof

Thank you. Questions?

Introduction

Medusa

Formal analysis

Benchmarks

Conclusion