Faster simulation of (Coloured) Petri nets using parallel computing

Franck Pommereau, Jordan de La Houssaye

▶ **To cite this version:**

Franck Pommereau, Jordan de La Houssaye. Faster simulation of (Coloured) Petri nets using parallel computing. 38th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2017), Jun 2017, Zaragoza, Spain. pp.37–56, 10.1007/978-3-319-57861-3_4 . hal-01533514

HAL Id: hal-01533514
https://hal.science/hal-01533514

Submitted on 8 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Faster Simulation of (Coloured) Petri Nets Using Parallel Computing

Franck Pommereau and Jordan de la Houssaye

IBISC, University of Évry, 23 boulevard de France, 91037 Évry Cedex, France
franck.pommereau@ibisc.univ-evry.fr, jordan.delahoussaye@ibisc.univ-evry.fr

Abstract. Fast simulation, *i.e.*, automatic computation of sequential runs, is widely used to analyse Petri nets. In particular, it enables for quantitative statistical analysis by observing large sets of runs. Moreover, fast simulation may be used to actually run a Petri net model as a (prototype) implementation of a system, in which case such a net would embed fragments of the code of the system. In both these contexts, being able to perform faster simulation is highly desirable.
In this paper, we propose a way to accelerate fast simulation by exploiting parallel computing, targeting both the multi-core CPUs available nowadays in every laptop or workstation, and larger parallel computers including those with distributed memory (clusters). We design an algorithm to do so and assess in particular its correctness and completeness through its formal modelling as a Petri net whose state space is analysed. We also present a benchmark of a prototype implementation that clearly shows how our algorithm effectively accelerates fast simulation, in particular in the case of large concurrent coloured Petri nets, which is precisely the kind of nets that are usually slow to simulate.

Keywords: Petri nets, fast simulation, parallel computing.

1 Introduction

Fast simulation is a widely used technique that consists in computing a run of a Petri net. Such a run is built automatically, as fast as possible, contrasting with interactive simulation for which the user is requested to chose the next of every step in the run. Fast simulation allows the modeller to make direct observations of the behaviour of a Petri net, just like developers run their programs to observe them. How fast is fast simulation becomes crucial when one wants to build large sets of runs on which various properties may be measured in order to perform quantitative statistical analysis, as done in [4] for example. This is a widely used technique that nicely complements qualitative analysis through state-space analysis (in particular, model-checking). Moreover, considering coloured Petri nets more specifically, one may embed in a Petri net model parts of the functional fragments of system under design, *i.e.*, the Petri net may be equipped with code that performs actual computation and is intended to be integrated into the final system. In such a case, the Petri net is used both for analysis and for simulating

the non-functional parts of the system that may be complicated to implement (which is true in particular for distributed system). Such a model may thus be used as an implementation of the system, or just considered as a prototype that may be simulated to observe runs or perform tests in the environment where the system is expected to operate. In such a case also, being able to accelerate simulation is very interesting.

In this paper we propose a parallel algorithm to accelerate simulation, designed in such a way that it can be implemented using a wide variety of programming languages, and targeting shared memory as well as distributed memory computer. This makes it suitable for being used on laptops or workstations equipped with multi-core CPUs (that is, all of them nowadays) as well as on dedicated computers like clusters. The main idea behind this algorithm is to compute a run of a Petri net from a sequential pseudo-concurrent process (*i.e.*, using *cooperative multitasking* within a single thread of execution) that delegates some computation to external processes through remote procedure calls. We present our algorithm, called *Medusa*, and we assess its properties resorting to formal modelling and analysis, then we present a prototype implementation and a benchmark of its performances. Doing so we show that Medusa is *correct* (it only builds legal runs), *complete* (it can build any run of the simulated Petri net), and *fair* (it has no bias that would favour some runs over others). The benchmark shows that our choice of a cooperative multitasking algorithm requesting worker processes (or threads) can be easily and effectively implemented. Moreover, we observe that Medusa's acceleration grows linearly with the number of processes, up to some point where it amortises, which depends on the intrinsic concurrency in the Petri net and on the work load necessary to compute the successors of a marking through one transition. Consequently, we can conclude that Medusa is well suited for large and concurrent coloured Petri nets, which is exactly the case where faster simulation is the most needed.

The next section provides the definition of the Petri nets we use in the rest of the paper. Then, section 3 presents Medusa algorithm and its formal analysis. Section 4 describes the prototype implementation and its benchmark, and a detailed analysis of the results obtained from the latter. The paper ends on a conclusion with a discussion about related and future works.

2 Petri Nets

To start with, we need to define multisets and various operations between them.

Definition 1. *A* multiset *A over of set X is a function $X \to \mathbb{N}$ where for every $x \in X$, $A(x)$ is the number of occurrences of x in A. For A and B two multisets over X, we define:*

- *$A \leq B$ iff $A(x) \leq B(x)$ for all $x \in X$;*
- *$A + B$ is the multiset over X such that $(A + B)(x) \stackrel{\text{df}}{=} A(x) + B(x)$ for all $x \in X$;*

– $A - B$, *defined iff* $B \leq A$, *is the multiset over* X *such that* $(A - B)(x) \stackrel{\text{df}}{=} A(x) - B(x)$ *for all* $x \in X$.

A (coloured) Petri net involves values, variables and expressions. These objects are defined by a *colour domain* that provides data values, variables, operators, a syntax for expressions, possibly typing rules, etc. For instance, one may use integer arithmetic or Boolean logic as colour domains. Usually, more elaborated colour domains are useful to ease modelling, in particular, one may consider a functional programming language or the functional fragment (expressions) of an imperative programming language. We consider here an abstract colour domain with the following pairwise disjoint sets:

– $\mathbb{D}$ is the set of *data* values;
– $\mathbb{V}$ is the set of *variables*;
– $\mathbb{E}$ is the set of *expressions*, involving values, variables and appropriate operators. Let $e \in \mathbb{E}$, we note by $\mathsf{vars}(e)$ the set of variables from $\mathbb{V}$ involved in e. Moreover, variables or values may be considered as (simple) expressions, *i.e.*, we assume that $\mathbb{D} \cup \mathbb{V} \subset \mathbb{E}$.

We make no assumption about the typing or syntactical correctness of expressions; instead, we assume that any expression can be evaluated, possibly to $\bot \notin \mathbb{D}$ (undefined). More precisely, a *binding* is a partial function $\beta : \mathbb{V} \to \mathbb{D}$. Let $e \in \mathbb{E}$ and β be a binding, we note by $\beta(e)$ the evaluation of e under β. The application of a binding to evaluate an expression is naturally extended to sets and multisets of expressions.

Take for instance $\beta \stackrel{\text{df}}{=} \{x \mapsto 0\}$, we may have distinct evaluations of various expressions depending on the chosen colour domain:

– in Python, C and JavaScript: $\beta(x + 1) = 1$;
– in Python, $\beta(x + \texttt{"hello"}) = \bot$ because of a type exception, in C it is the address of string `"hello"`, while in JavaScript it is the string `"0hello"` because an automatic coercion is performed;
– in Python and JavaScript $\beta(x + y) = \bot$ because of an exception complaining that y is not defined, and in C also but because of a compilation error.

We can now define a variant of coloured Petri nets that is independent of the annotation language.

Definition 2. *A* Petri net *is a tuple* (S, T, ℓ) *where:*

– S *is the finite set of* places*;*
– T*, disjoint from* S*, is the finite set of* transitions*;*
– ℓ *is a* labelling *function such that:*
  • *for all* $s \in S$, $\ell(s) \subseteq \mathbb{D}$ *is the* type *of* s, i.e.*, the values that* s *is allowed to carry as tokens,*
  • *for all* $t \in T$, $\ell(t) \in \mathbb{E}$ *is the* guard *of* t, i.e.*, a condition for its execution,*
  • *for all* $(x, y) \in (S \times T) \cup (T \times S)$, $\ell(x, y)$ *is a multiset over* $\mathbb{E}$ *and defines the* arc *from* x *towards* y.

For all $x \in T \cup P$ we define ${}^\bullet x \stackrel{\text{df}}{=} \{y \in P \cup T \mid \ell(y,x) \neq \emptyset\}$ and $x^\bullet \stackrel{\text{df}}{=} \{y \in P \cup T \mid \ell(x,y) \neq \emptyset\}$. These notations are naturally extended to sets of nodes.

Then we define the dynamic aspect of Petri nets, *i.e.*, the markings and the firing of transitions. Note that we also introduce a notion of *token flow* that will be needed to define our algorithm.

Definition 3. *Let $N \stackrel{\text{df}}{=} (S,T,\ell)$ be a Petri net. A* marking *M of N is a function on S that maps each place s to a finite multiset over $\ell(s)$ representing the tokens held by s. For two markings A and B, we define:*

- *$A \leq B$ iff $A(s) \leq B(s)$ for all $s \in S$;*
- *$A + B$ is the marking such that $(A+B)(s) \stackrel{\text{df}}{=} A(s) + B(s)$ for all $s \in S$;*
- *$A - B$, defined iff $B \leq A$, is the marking such that $(A-B)(s) \stackrel{\text{df}}{=} A(s) - B(s)$ for all $s \in S$.*

Let $t \in T$ be a transition, M a marking and β a binding. The token flow *generated by t and β at M is a pair of markings $sub_{M,t,\beta} \stackrel{\text{df}}{=} \{s \mapsto \beta(\ell(s,t)) \mid s \in S\}$ and $add_{M,t,\beta} \stackrel{\text{df}}{=} \{s \mapsto \beta(\ell(t,s)) \mid s \in S\}$. Then, t is* enabled *for β at M, which is noted by $M[t,\beta\rangle$, iff the following conditions hold:*

- *M has enough tokens,* i.e.*, $sub_{M,t,\beta} \leq M$;*
- *the guard is satisfied,* i.e.*, $\beta(\ell(t))$ is true;*
- *place types are respected,* i.e.*, $add_{M,t,\beta}$ is a valid marking of N.*

If $t \in T$ is enabled for binding β at marking M, then t may fire *and yield a marking M' defined by $M' \stackrel{\text{df}}{=} M - sub_{M,t,\beta} + add_{M,t,\beta}$, which is noted by $M[t,\beta\rangle M'$. Finally, we note by $[M\rangle$ the set of all the markings reachable from a marking M through arbitrary sequences of transitions and bindings.*

3 Medusa: a Concurrent Simulation Algorithm

In order to distinguish the transitions in a Petri net from the corresponding data structures used in Medusa algorithm, we call the latter *players*; by extension, we shall call player as well the activity that handles this data structure (*i.e.*, we may adopt an object-oriented point of view, which fits with our implementation). Moreover, each player belongs to a *team* that consists of the other players with which it is in competition to fire its transitions. More precisely, for every transition we define a corresponding *player* to record information about this transition as follows:

```
1 struct player :
2     trans : transition    # the player's transition
3     team  : set[player]   # its team (a set of players)
4     out   : set[player]   # its output (a set of players)
5     busy  : bool          # is the player currently working?
6     retry : bool          # should the player retry its current work?
```

Then, for each transition (and the corresponding player), we define its *team* as the set of transitions (and players) with which it is in conflict (including the transition itself), and its *output* as the set of transition for which it may produce a token. Formally:

Definition 4. *Let* $N \stackrel{\text{df}}{=} (S, T, \ell)$ *be a Petri net, then for all* $t \in T$ *we define* $\mathsf{team}(t) \stackrel{\text{df}}{=} ({}^\bullet t)^\bullet$ *and* $\mathsf{out}(t) \stackrel{\text{df}}{=} (t^\bullet)^\bullet$*. Let* p *be a player, then* $\mathsf{p.team}$ *is the set of players such that* $\mathsf{team}(\mathsf{p.transition}) = \{\mathsf{q.transition} \mid \mathsf{q} \in \mathsf{p.team}\}$*, and* $\mathsf{p.out}$ *is the set of players such that* $\mathsf{out}(\mathsf{p.transition}) = \{\mathsf{q.transition} \mid \mathsf{q} \in \mathsf{p.out}\}$*.*

Finally, a Petri net *run* is a list r of markings (this may be enriched easily) whose latest item is noted as $\mathsf{r.last}$, and we use an operation "**append** m **to** r" to add a marking m at the end of r. Moreover, [] represents the empty run.

3.1 Concurrency Model

We consider an interleaving concurrency model in which the simulation engine is executed on a single computation unit (*e.g.*, one core of one CPU) on which multiple sequential threads of execution are interleaved. Other computation units exist and are exploited to execute code through *remote procedure calls* (RPC). Moreover, a thread has to explicitly release the control so that another can be scheduled, that is, we assume so called cooperative multitasking. Consequently, there is no need for lock primitives to guarantee consistent accesses to data structures. To express this we use two primitives:

- "**call** $\mathsf{fun}(\cdots)$" invokes a function fun asynchronously, *i.e.*, its execution starts in a new thread but it is not scheduled immediately, instead, the caller is able to continue its own execution. This requires that "fun" returns no value because the caller has no way to get it;
- "**rpc** $\mathsf{fun}(\cdots)$" is similar to **call** but the caller is blocked until the result is returned. Moreover, the execution of fun is performed on another computation unit so that, while the caller is waiting, another thread can be scheduled and executed in the simulation.

The goal is to achieve speedup by allowing this sequential engine to be executed in parallel with the remote procedures it calls. In the implementation, a limited pool of worker processes will be used to execute the RPC calls, which means that not all those calls can be executed in parallel but some are sequentialised. But this has no consequence on the definition of the algorithm.

3.2 Medusa Algorithm

Figure 1 shows Medusa algorithm. The entry point is function $\mathsf{startup}$ that creates a new run and launches in parallel one instance of procedure work for each player. Players are marked as busy and they all share the same run just created.

The main procedure is thus work. Line 9, it computes the token flows available from the current marking by calling remote procedure $\mathsf{getflows}$. Note that this

```
1  def startup (players) :
2      run ← []
3      for player in players :
4          player.busy ← True
5          call work(player, run)
6
7  def work (player, run) :
8      player.retry ← False
9      flows ← {f in rpc getflows(player.trans, run.last) | f.sub ≤ run.last}
10     if player.retry and flows = ∅ :
11         call work(player, run)
12     elif flows = ∅ :
13         player.busy ← False
14     else :
15         choose flow in flows
16         append run.last − flow.sub + flow.add to run
17         player.busy ← False
18         for other in player.team ∪ player.out :
19             if not other.busy :
20                 other.busy ← True
21                 call work(other, run)
22             elif other.busy and other in player.out :
23                 other.retry ← True
```

Fig. 1. Medusa algorithm, where remote procedure getflows(t,m) returns the set of token flows for a transition t at a marking m.

call is initiated using the latest marking in the run, and it may take some time during which other instances of work will be scheduled. When flows are returned and this instance of work is scheduled again, they are filtered by keeping only those that are applicable to the current latest marking. This may not be the same marking as when getflows has been called if another work instance from the same team has fired its transition (thus removing tokens). Line 10, if the player has been told to retry and found no usable flow, work is called again. Otherwise, line 12, if there is no usable flow then the player becomes idle. Finally, line 14, the player has a usable flow and fires its transition by computing a new marking that is added to the run. Then it is marked idle (line 17) and it has to inform all the player in its team or in its output that their input markings has been changed: in the team we have removed tokens, in the output, we have added tokens. If such a player is not busy, it is put to work (lines 20–21), otherwise, if this is a player in the output, it is told to retry because it is waiting for getflows and will retrieve an outdated result since we just added tokens to its input places. Retry is actually attempted (line 13) only is no valid flow is available, this is necessary to avoid a deadlock, but not mandatory since there are already usable flows. Remember that we assume cooperative multitasking, so when an instance of work finds another player to be busy, then the work instance for this latter player *must* by paused because of the call to getflows.

3.3 Formal Analysis

To assess Medusa's expected properties, we have conducted two actions. On the one hand, during our benchmark presented below, every run computed has been checked to be a correct run of the executed Petri net. On the other hand, we have built a formal model of Medusa using the ABCD specification language [15] and analysed its state space, which is reported now. We have not enough room to present the model (available here [16]) but we describe it here in terms of its Petri net semantics, that we call the *model-net* to distinguish it from the *simulated-net* for which it computes a run:

- a place holds the players structures, initially at the state they have at the end of startup, plus a field state to record which step of work this player is currently handling. So, simulating **call** work(player, run) just requires to put player.state at the appropriate value;
- instead of the whole computed run, only run.last is stored, into a dedicated place, because it is the only marking we need and because this allows one to keep a finite state space (provided that the simulated-net itself has a finitely many markings);
- one transition models the beginning of work, picking a player, updating it and putting in another place the flows obtained from getflows (this piece of code is named *rpc*). These flows will be retrieved by one of the following transitions;
- a second transitions models the end of line 9 that filters the flows and the execution of line 11 (code piece *retry*);
- a third transition models also the end of line 9, but now together with line 13 (code piece *idle*);
- finally, a fourth transition models the end of line 9 together with lines 15–23 (code piece *fire*).

Instances of these transitions for the different players are interleaved when the model-net is executed, which models cooperative multitasking.

We have computed the state space of this model-net when its fires the transitions of simulated-nets like those depicted in figure 2. In figure 3, we show the resulting marking graph for simulated-net (3). Slightly larger nets have been analysed as well, with the same results as presented below, but the state space of the model becomes quickly intractable for nets with more that 4-5 transitions. However, the nets depicted in figure 2 have been carefully chosen to focus on the analysed aspects: correctness, completeness, deadlocks, progression, and fairness.

To check these properties, we consider the marking graph G_m of the model-net in which: every node is replaced by the marking of the simulated-net that is stored in run.last in this state of the model-net; every edge that corresponds to the *fire* piece of code is labelled by the transition that is fired (*i.e.*, on black edges in figure 3, the gray parts of the labels are removed); other edges (the gray ones) are labelled by τ. Then, let G_m/τ be G_m in which all the τ transitions have been collapsed (*i.e.*, whenever $x \xrightarrow{\tau} y$ we merge nodes x and y and remove the resulting τ-labelled side-loop). We have checked that for each simulated-net,

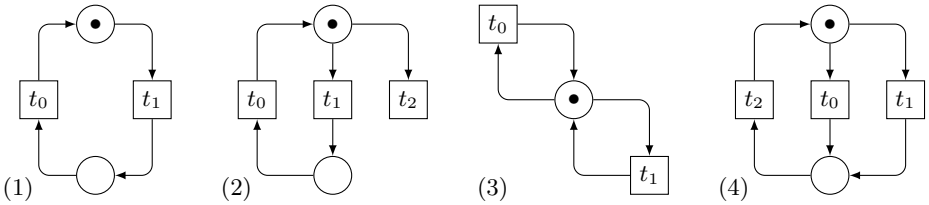

Fig. 2. Examples of simulated-nets used to analyse the model of Medusa. From the left to the right, these simulated-nets yield model-nets with state spaces with respectively 18, 135, 8, and 126 states. The state space for net (3) is depicted in figure 3.

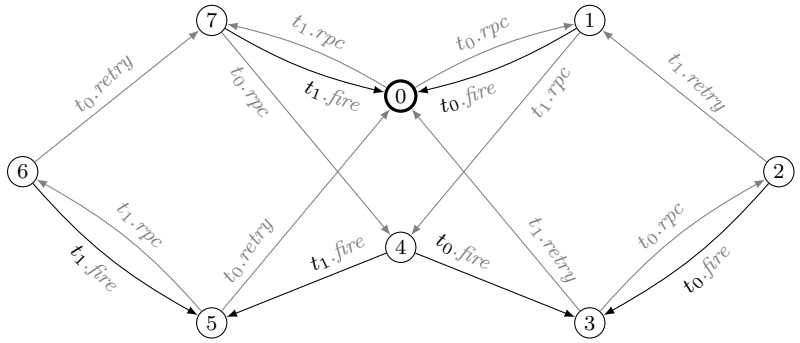

Fig. 3. The state space of the model-net for simulated-net (3) from figure 2 where the initial state is 0. Edges are labelled with the player's transition for which procedure work is active followed by the name of the code piece that is executed. Gray arcs are those whose label is replaced by τ to check weak bisimilarity.

its marking graph G_s and G_m/τ are isomorphic, from which we have that the model-net and the simulated-net are *weak bisimilar* [5]. Consequently, we obtain:

- *correctness*: every run constructed by the model-net is a valid run of the simulated-net;
- *completeness*: for every run of the simulated-net, there is an execution of the model-net that constructs this run;
- *deadlocks equivalence*: the simulated-net and the model-nets have exactly the same deadlocks, if any, *i.e.*, every deadlock in one net corresponds to a deadlock in the other net with the same marking of the simulated-net;
- *progression*: we have checked that there is no loop with only τ edges in G_m, which means that the model-net cannot progress unboundedly without executing an instance of the *fire* piece of code;
- *fairness equivalence*: a fair (resp. unfair) run of the simulated-net can be constructed only by a fair (resp. unfair) run of the model-net because both nets have bisimilar executions and the model-net has to progress. So considering for example G_m for simulated-net (3) that is shown in figure 3, a fair execution of Medusa cannot always favour loops like $0 \leftrightarrow 1$ where transition t_1 is completely excluded, or $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$ where transition t_1 is con-

sidered but never allowed to fire. We will discuss fairness from a quantitative point of view in the next section.

4 Towards an Efficient Implementation

We have made a prototype implementation of Medusa using SNAKES [14] for the Petri net aspects and gevent [2] for the cooperative multitasking aspects. The simulator consists of one main process that executes Medusa and a bunch of worker processes to which computation tasks are delegated through RPC. The implementation of Medusa itself is in Python and is very close to the pseudo-code presented above, the implementation of **call** and **rpc** on the top of gevent requires less than 100 additional lines of code. All this code is available in [16].

We have exercised this prototype on a choice of parametrised models depicted in figure 4. Each model is designed as a reversible net so that we can run arbitrary long simulations without encountering a deadlock. Models StarFlower and HyperLoops have both conflicts and concurrency, while Cycle has no conflict and its concurrency is limited to the number of tokens, finally, Parallel has no conflicts but maximal concurrency. Each model has also a parameter $chroma \geq 0$ that allows to simulate the effect of having coloured Petri nets for which firing transitions takes more time: each transition is equipped with a guard that is always true but spends $chroma/10$ seconds using the CPU intensively. To do so it computes the BZip2 compression of the SHA512 hash of a random value, and repeatedly on the result. This loop involves two CPU-intensive algorithms and using a random seed plus a strong cryptographic hash ensure that we avoid potential effects of the computer's caches.

We have built more than 700 instances of these models with various parameters and ran about 22k simulations of these instances that we have split into five classes according to their number of transitions, as shown in figure 5. Note that we have also ran simulations for much larger nets with up to 115k transitions but there are too few to present smooth results, however, they are so far consistent with what we present here. These simulations allowed to observe various aspects:

- how adding more worker processes speeds-up the simulation;
- how the simulation speed is influenced by the amount of computation required to fire transitions as captured by parameter *chroma*;
- how Medusa behaves when the number of transitions grows;
- how theoretical "fairness equivalence" can be observed on an implementation.

This benchmark have been run on a computer equipped with two 64 bits Intel® Xeon® hexacore CPUs at 2.67GHz, sharing 44Gb of RAM, which allows to run up to 12 processes independently so we have limited our benchmark to 10 worker processes in order to leave enough CPU for Medusa main process and the operating system. On the software side, we have used Debian 8.6 with Linux kernel 4.4.19, Python 2.7.9, gevent 1.1.2 (for cooperative multitasking) [2], gipc 0.6.0 (simplified IPC for gevent) [8], psutil 4.3.1 (measure the CPU and memory usage of processes) [17], and SNAKES 0.9.21 [14].

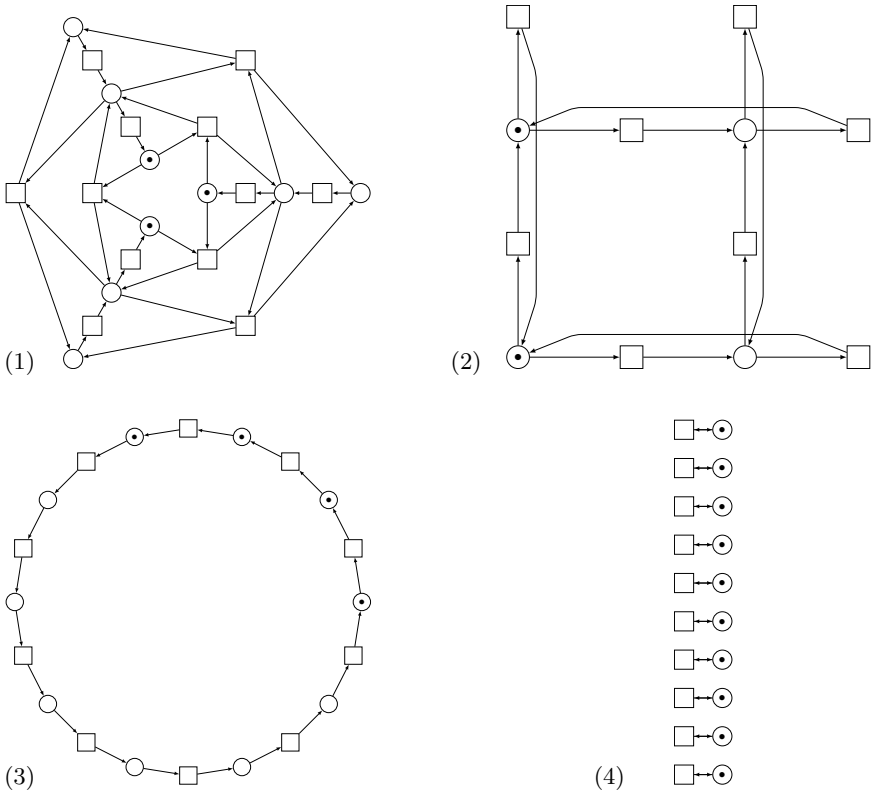

Fig. 4. Instances of the models used for the benchmark. (1) *StarFlower*($depth = 2, num = 3$) is organised as a series of concentric connected rings of places and transitions, where *depth* is the number of rings and *num* is the number of pairs of places and transitions on a ring. (2) *HyperLoop*($dim = 2, width = 2$) is organised as an hyper-cube of dimension *dim* where *width* is the number of transitions on one edge. (3) *Cycle*($length = 10, tokens = 4$) is organised as a loop, where *length* is the number of transitions and *tokens* the number of tokens. (4) *Parallel*($length = 10$) is organised as a series of independent side-loops, where *length* is the number of transitions.

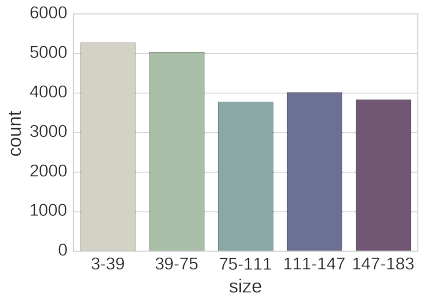

Fig. 5. Distribution of the simulated instances by size (number of transitions).

4.1 Performance Analysis

Let us first remark that the time necessary to build a set of players from a Petri net is always negligible and thus not analysed here. Moreover, by design, Medusa cannot accelerate the simulation of Petri nets that have no concurrency at all because it can only parallelise the firing of concurrent transitions.

That said, figure 6 shows how many transitions per second Medusa can fire, with respect to the number of worker processes and to the size of the nets, for *chroma* $\in \{0, 1\}$, decomposed by model. We observe in general a linear speedup until 5 or 6 processes, where it starts to amortise. For uncoloured models, firing rate is better for smaller nets and it decreases with the size of the nets. For coloured models, rates are of course much lower because every transition firing takes 0.1 second, but larger nets globally have better rates. Moreover, for models with more concurrency between transitions (*i.e.*, larger models or those with less conflicts) Medusa scales better with the number of processes. See in particular the curves for the coloured instances of Parallel.

We explain these observations by the overhead introduced by Medusa that is more important in the uncoloured case than in the coloured case where most of the time is spent computing token flows. This is not surprising that an algorithm that is designed to parallelise computation tasks is more efficient when these tasks take more time. Furthermore, having more concurrent nets yields more such tasks so it allows to better use worker processes that are always fed with token flows to compute, in such a way that we maximise parallel efficiency.

To confirm this, we have programmed a very simple sequential simulator directly based on SNAKES that repeatedly chooses and fires a transition (see its pseudo-code in figure 7). We have exercised this simulator on the same instances and compared it with Medusa (using 6 to 10 worker processes).

The results are provided in figure 8. We can observe that SNAKES simulation is better than Medusa for uncoloured nets (except on model Cycle), but that Medusa is much faster than SNAKES for coloured nets. In the former case, this is explained again by the additional cost of IPC and the relatively bigger complexity of Medusa algorithm, which cannot be compensated by parallel computation because token flows require very little computation time. In the latter case, parallel computation becomes effective and Medusa is actually able to compute in parallel the token flows for several transitions, which is the most clearly observed in the case of Parallel.

Another confirmation that worker processes are efficiently exploited can be obtained by decomposing the time spent in IPC into three classes: (1) waiting: when Medusa is waiting for a worker process to be available, during which another task in Medusa may be active (for instance to fire a transition); (2) communication: the time spent in sending a request from Medusa to a worker process, or in receiving the answer; (3) computation: the time spent by the worker process to compute the token flows, during which Medusa may be busy handling other players. This is shown in figure 9 where we observe that waiting time is largely dominant when there is not enough worker processes, but quickly decreases with the number of processes. Moreover, communication time is always negligible and

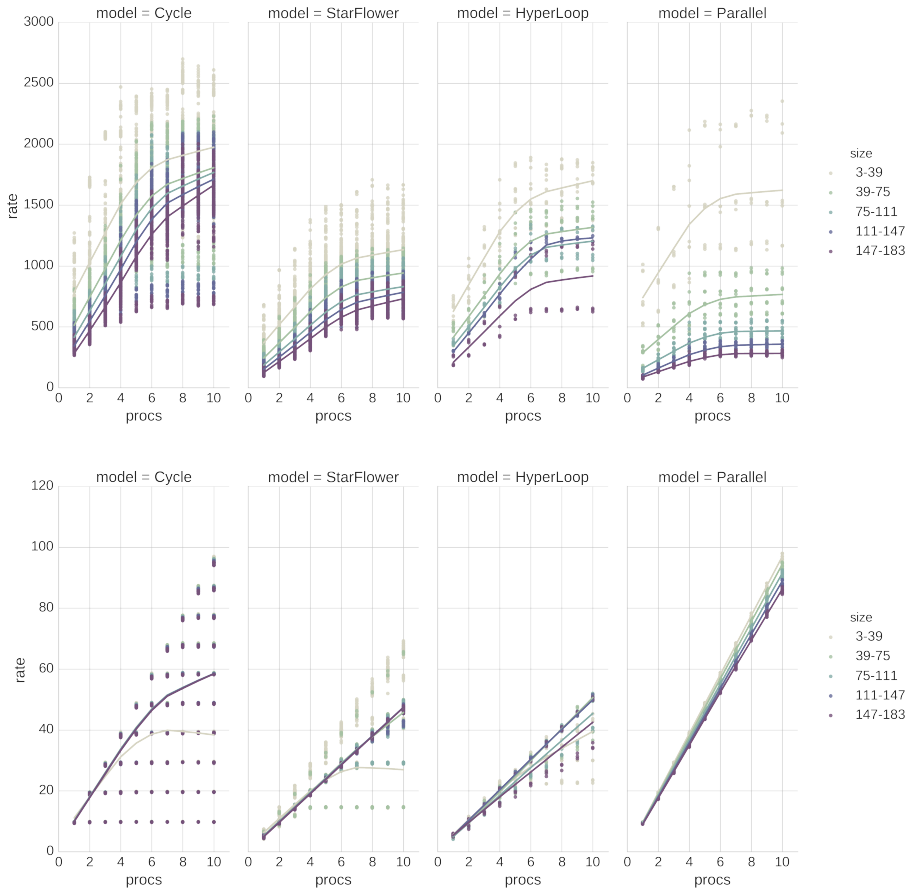

Fig. 6. Firing rates with respect to the number of worker processes, for *chroma* = 0 (top) and *chroma* = 1 (bottom). Higher is better.

not visible in this graph. Note also that we obtain identical plots if we separate cases by *chroma* or net sizes. This waiting time that never disappears shows that Medusa is always fast compared to the worker processes. So, the fact that firing rates do not grow linearly after some point does not come from the overhead of the parallel processing, but it must come from the insufficient intrinsic concurrency in the simulated nets, which does not occur with Parallel.

This is also illustrated on figure 10 where we focus on coloured instances of model Cycle: in this model, the number of tokens is exactly the maximum number of transitions that may fire concurrently (but if several tokens are located in the same place, concurrency is lower because Medusa does not consider auto-concurrency). In figure 10 we clearly observe that Medusa has exactly the same performance than the SNAKES-based simulator when only one token is avail-

```
def simul (net) :
    deadlock ← False
    while not deadlock :    # simulate until a deadlock is found
        deadlock ← True   # assume the current marking is a deadlock
        for trans in shuffle(net.transitions) :  # try to fire each transition
            flows ← getflows(trans, net.marking)
            if flows ≠ ∅ : # choose one flow and fire
                choose flow in flows
                net.marking ← net.marking − flow.sub + flow.add
                deadlock ← False  # actually we did not reach a deadlock
                break             # quit for loop ⇒ restart while loop
```

Fig. 7. The simple SNAKES-based simulator used for comparison with Medusa.

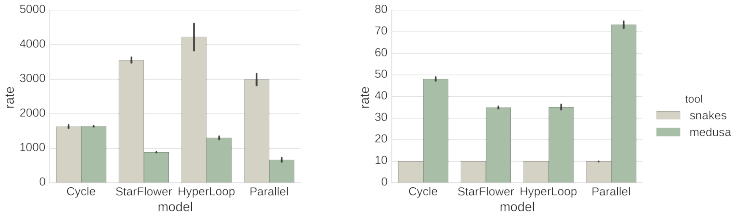

Fig. 8. Compared mean firing rates of SNAKES and Medusa (for 6–10 processes), by models, and for *chroma* = 0 (left) and *chroma* = 1 (right). Higher is better.

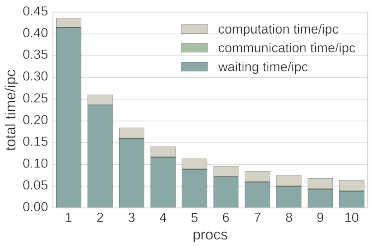

Fig. 9. Decomposition of the time spent in IPCs with respect to the number of worker processes.

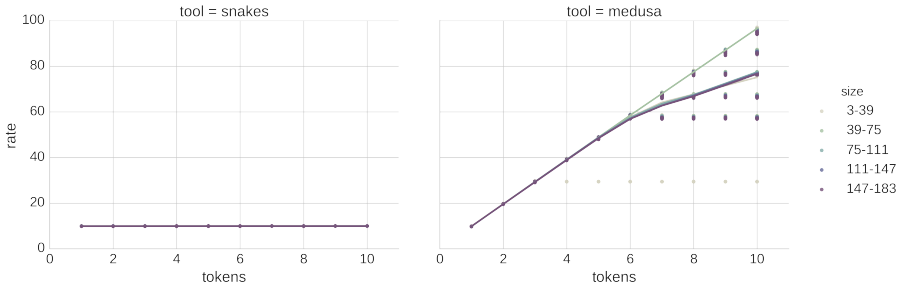

Fig. 10. Mean firing rates of SNAKES and Medusa (for 6–10 processes) on the Cycle model for *chroma* = 1 with respect to the number of tokens.

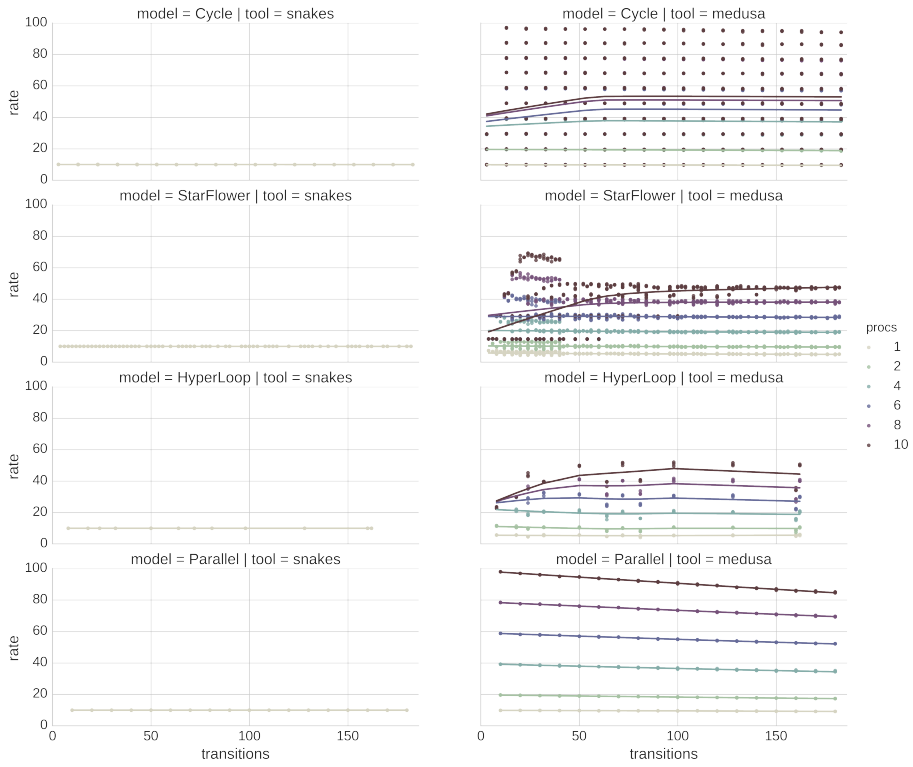

Fig. 11. Mean firing rates of SNAKES and Medusa for *chroma* = 1 with respect to the number transitions, decomposed by tool, model, and number of worker processes.

able (*i.e.*, 10 transitions per second as each transition takes 0.1 second), but it accelerates with the number of tokens.

Finally, in figure 11 we show how Medusa behaves when the size of the simulated Petri net grows. The rate of the SNAKES simulator is plotted also for comparison. As previously observed, adding transitions allows to add concurrency, which yields better rates, especially when we use more processes. We can indeed see how the initial slope is steeper when there are more worker processes. Then, after this initial growth, the rates stabilises and only depends on the number of workers. We can also observe that, for a fixed number of workers, the rates stay constant when the nets get larger. However, model Parallel is different since the rate decreases with the size of the nets. Actually, the smallest instances we have used have 10 concurrent transitions, which is already enough to keep 10 workers fully busy, and indeed, we reach 100 firings per second for 10 workers and 10 transitions whose firing costs 0.1 second each. But the workers are already fully busy and cannot absorb more, so increasing the number of transitions only results in increasing the overhead to manage them. Similarly, for a fixed number of transitions, adding more workers only results in increasing the overhead to manage them as well.

From all these experiments, we observe that for a better overall performance, we need to have: (1) more worker processes to reduce the waiting time; (2) more concurrent Petri nets to always be able to feed these worker processes and use them as much as possible; (3) more CPU-intensive activity when computing token flows to reduce the overhead introduced by Medusa, which is the case with coloured nets that may perform high-level computation in their arcs and in their guards. So we conclude that Medusa is well suited to accelerate the simulation of large coloured Petri nets involving CPU-intensive computation, which is the typical case that requires acceleration (the other cases being naturally fast to simulate).

4.2 Analysing the Design of Medusa

Medusa is designed to be "as concurrent as possible", *i.e.*, every player is given a chance to fire its transition, even if its work may be invalidated by another player firing a transition (which yields a retry, *i.e.*, the execution of lines 10–11 in figure 1).

To observe how this design impacts the execution, we have counted the number of retries performed by Medusa with respect to the number of firings, which is plotted in figure 12. This shows that the number of retries highly depends on the model (from lines 18 and 22 in figure 1: a retry is possible only if a player is working on a transition that is given tokens from another, but may have had tokens stolen by another player in the team). But in any case it remains proportional to (and lower than or equal to) the number of firings.

Fewer retries would be desirable because they correspond to wasted work. But it currently looks like they are hard to avoid without resorting to a higher-level scheduler that would choose which player is given a chance to fire its transition. But doing so is not necessarily more efficient because such an arbiter cannot know in advance which player will actually succeed in firing its transition.

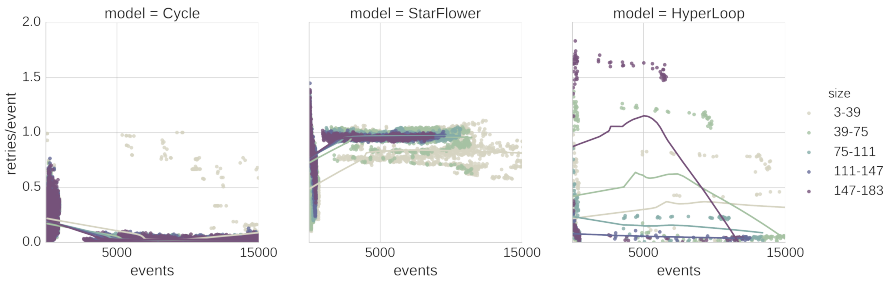

Fig. 12. Number of retries with respect to the number of firings. Lower is better. Model Parallel is not depicted because it requires no retry at all.

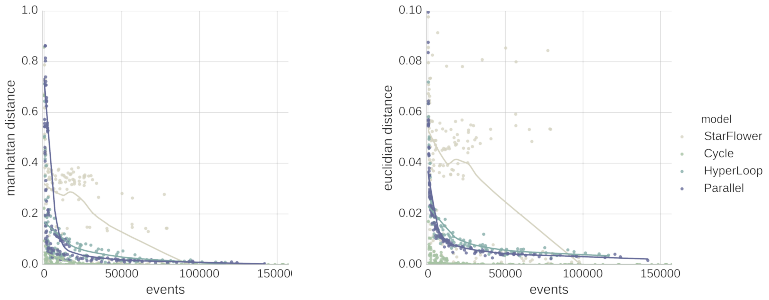

Fig. 13. Distances between SNAKES and Medusa simulations with respect to the number of firings (events). Lower is better.

4.3 Fairness Analysis

To analyse the fairness of Medusa from a quantitative perspective, we have compared it to the SNAKES simulator (presented above, see figure 7) using both small and large instances of each model. Each instance has been simulated using both simulators for growing lengths of simulations (measured as the number of firings). For each pair of simulation, we have counted the frequency of firing for each transition and computed the distance between these two vectors. Both simulators use the same random number generator. The result is rendered in figure 13 from which we observe that, for every model, the two simulators converge to the same distribution of the transitions they choose as the number of firing grows. However, we have used nets whose transitions need all the same time to fire, it is likely that if some transitions need less time than others, they are more likely to be chosen by Medusa that is designed to fire transitions "as soon as possible", which is not the case for a sequential simulator that first chooses a transitions to fire regardless of the time it will take.

5 Conclusion

In this paper we have presented a parallel algorithm to compute a sequential run of a Petri net. This algorithm is based on cooperative multitasking and thus is actually sequential, but it delegates computation to other processes which allows to keep it simple while achieving acceleration by launching computation in parallel processes. This architecture is well suited for multi-core CPUs or multi-CPUs computers, as well as for distributed computers where inter-process communication is made over the network. Moreover, our Python prototype shows that Medusa can be implemented successfully even with a language that is notoriously bad with concurrency and whose default implementation (that we have used) cannot exploit more than one core or CPU.

We have presented a formal analysis of Medusa, as well as a prototype implementation with a benchmark. This allowed to show that Medusa is well suited to accelerate the simulation of large coloured Petri nets with concurrency, that

is, exactly the kind of nets that are usually slow to simulate and for which faster simulation is highly desirable. Moreover, Medusa scales well and is able to execute very large nets with thousands of transitions. We have only limited experiment for these cases but the systematic bench we have presented is still in progress processing larger nets. Preliminary results are fully consistent with what we have presented above. Moreover, we have shown in our benchmark that the firing rate does not depend on the size of the nets (except in pathological cases like model Parallel). Finally, Medusa has been proved correct and complete on a limited but varied set of simulated nets.

We have also identified some limitations of Medusa, and of our analysis. First, it is designed to compute random runs by fire transitions "as soon as possible", consequently, it may be a big change to design a variant with, for instance, priorities between transitions. Such variants have been tested and involve a higher-level scheduling, which works correctly but slows down simulation because we try to fire transitions in one order that is not necessarily available Moreover, on the one hand, we have shown that Medusa does not generate unfair runs of the simulated Petri net from fair runs of the simulation. But on the other hand, unfair runs of the simulation are likely to occur with nets whose transitions do not all require the same computation time considering that Medusa will tend to fire the first one for which this computation completes. Finally, we have seen that Medusa introduces an important overhead on small uncoloured Petri nets and that it cannot accelerate a Petri net that has no concurrency because it is just as concurrent as the simulated Petri net. To do so we would need a radically different algorithm focusing on parallelising the firing of a single transition. This is however not really a limitation but rather a design choice, and actually, having simultaneously two levels of concurrency can be envisaged.

Despite these limitations, we believe that we have demonstrated strong arguments to convince that Medusa has a clear potential to lead to efficient implementations of faster simulators.

5.1 Related Works

The question of executing a Petri net has been long standing. In [11], the author propose a sophisticated solution based on places invariants to extract processes from a Petri net in order to distribute its execution on Ada threads. This is necessary because a naive implementation that would put one thread for each transition would quickly collapse as the number of transitions would grow (threads or processes do not scale well). This highlights a crucial aspect of our proposal: we rely on cooperative multitasking instead of threads, which we have tested to scale happily to hundreds of thousands of pseudo-concurrent activities. Building such a net or loading it in memory takes a lot of time, but we have tested that once this is done, simulation is as fast as with smaller nets, even faster if more transitions means more concurrency.

In [12], the question of parallel simulation of *timed* Petri nets is solved using also a sophisticated partitioning of the Petri net to distribute its nodes on the computation units (CPU + memory), and a complex communication protocol to

synchronise the distributed timed execution in order to ensure that it remains globally correct. The problem solved is substantially distinct from ours: [12] distributes a *timed uncoloured* Petri net and ensures the synchronisation the timed executions of its resulting parts, while we distribute the computation that arises from the execution of a centralised *untimed coloured* Petri net. We end up with a much simpler approach that has probably a much lower overhead and can be executed even on a modest multi-core CPU while [12] is designed for distributed-memory parallel computers.

[19] is a 3-pages paper in which, among other things, the authors describe a parallel simulation engine that relies on partitioning a Petri nets into a set of sub-nets, being then processed in parallel. However, this topic is treated only briefly in a paper that is already very short, so it gives no further details about the parallel algorithm itself. Moreover, we have found no further reference to this paper, including no available software.

Finally, as stated on Renew's web page [18], "[its] simulation engine is capable of true concurrency and supports symmetric multi-processor architectures". Considering that Renew is able to execute Java code attached to the transitions of the Petri nets it simulates [3], this makes it very comparable to SNAKES equipped with Medusa. (Note that, like Renew, SNAKES is able to execute nets-within-nets as shown in [13].) However, we've found no description of this concurrent simulation algorithm but a comparison with Medusa should be very interesting.

5.2 Future Works

In the future, we intend to combine Medusa with the compilation of Petri nets as performed by Neco [6]. In the current prototype, we indeed rely on SNAKES to compute the token flows and so, the simulated net is interpreted (*i.e.*, it is represented as a data structure that is requested and on which generic algorithms are executed). What Neco does is to compile a Petri net into a set of specialised data structures and algorithms so that the Petri net structure disappears and the algorithms can be optimised on a per-transition basis. Experiments have showed that this yields a dramatic improvement in the performances for the computation of state spaces. Neco is currently operating on steps that are coarser than we need, being able to compute the successors of a marking, but not the tokens flows. To make them available to Medusa, we will need to refine Neco.

That done, we should be able to experiment with other finer-grains algorithms. In particular, we envisage to randomise the iterations on the tokens of input places in order to compute a single random token flow, instead computing all the possible flows before to chose one randomly. This should greatly alleviate the work load of worker processes, making them available for more computation (thus reducing the waiting times we have observed in figure 9). But at the same time, this should also increase the number of retries as a single flow is more likely to be invalidated by a concurrent firing than a whole set of flows. In this new algorithm, a retry would however only consist in requesting the next random flow and so should be less time consuming than in the current setting.

We also want to experiment with other implementation languages, we think in particular about the Go language [1,9] that features native cooperative multi-tasking (through so called *goroutines*) with compiler that generates native code that is able to efficiently exploit multi-core CPUs. Go programs are expected to run at the speed of C, even with hundreds of thousands of goroutines started simultaneously. So we expect a Go implementation of Medusa to have a low overhead whose performance on sequential Petri nets could be comparable to a simpler sequential simulator like that outlined in figure 7.

Coming back to our current prototype, we would like to perform a finer analysis of the effect of colours. Choosing parameter *chroma* in $\{0, 1\}$ as considered in this paper turned out to be much too coarse, and we should have designed our benchmark with much smaller delays (*e.g.*, *chroma*/100 seconds instead of *chroma*/10) but when we have had enough data to analyse the benchmark and discovered this, it was to late to restart it from scratch. Furthermore, we should experiment with real cases of coloured Petri nets instead of the artificial ones we have used here. We shall also consider more varied models, in particular those from the model-checking contest [10] should be good candidates as they already come with scaling parameters.

Analysis of fairness should also be refined, first by defining properly what would fairness mean in the context of the "as soon as possible" execution policy on which Medusa has been built. This definition should be general enough to allow for a comparison with other policies, and we would like to adapt Medusa to allow it to be parametrised by such policies.

A more theoretical work will be to provide a formal mechanised proof of Medusa algorithm. We expect to be able to adapt the techniques used in [7] where the authors prove BSP-parallel programs. Such programs are structured as sequences of *super-steps*, each of which being the parallel run of independent sequential blocks of code. The proof technique relies on the fact that the blocks being independent, they may be run sequentially in any order that respect the order of the super-steps; so the proof is reduced to that of a sequential program. Like a BSP program, Medusa algorithm is also organised a set of sequential blocks that are already scheduled sequentially, so we see no major obstacle to reuse the techniques from [7].

Supplementary Material

The ABCD model of Medusa, the scripts to analyse it, and the prototype implementation of Medusa are available as free software under the GNU GPL licence at `http://github.com/fpom/PETRINETS-2017-supplementary` [16].

Acknowledgements

We warmly thank Camille Coti (LIPN) for her help in understanding our hardware and how to exploit it correctly for our benchmark.

References

1. The Go programming language. http://golang.org
2. Bilenko, D., gevent contributors: gevent. http://www.gevent.org
3. Cabac, L., Haustermann, M., Mosteller, D.: Renew 2.5 – towards a comprehensive integrated development environment for Petri net-based applications. In: Proc. of PETRI NETS 2016. LNCS, vol. 9698. Springer (2016)
4. Chaou, S., Utard, G., Pommereau, F.: Evaluating a peer-to-peer storage system in presence of malicious peers. In: Proceedings of HPCS'11. IEEE Computer Society (2011)
5. Fernandez, J.C., Mounier, L.: "On the fly" verification of behavioural equivalences and preorders. In: Proc. of CAV'91. LNCS, vol. 575. Springer (1992)
6. Fronc, Ł., Pommereau, F.: Building Petri nets tools around Neco compiler. In: Proc. of PNSE'13 (2013)
7. Gava, F., Fortin, J., Guedj, M.: Deductive verification of state-space algorithms. In: Proc. of IFM 2013. LNCS, vol. 7940. Springer (2013)
8. Gehrcke, J.P.: gipc: child processes and IPC for gevent. http://gehrcke.de/gipc
9. Kincaid, J.: Google's Go: A new programming language that's Python meets C++. http://techcrunch.com/2009/11/10/google-go-language (2009)
10. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Chiardo, G., Hamez, A., Jezequel, L., Miner, A., Meijer, J., Paviot-Adet, E., Racordon, D., Rodriguez, C., Rohr, C., Srba, J., Thierry-Mieg, Y., Trịnh, G., Wolf, K.: Complete Results for the 2016 Edition of the Model Checking Contest. http://mcc.lip6.fr/2016/results.php (2016)
11. Kordon, F.: Prototypage de systèmes parallèles à partir de réseaux de Petri colorés. Ph.D. thesis, UPMC (1992)
12. Nicol, D.M., Mao, W.: Automated parallelization of timed Petri-net simulations. Journal of Parallel and Distributed Computing (1995)
13. Pommereau, F.: Nets in nets with SNAKES. In: Proc. of MOCA'09. Universität Hamburg, Dept. Informatik, Hamburg (2009)
14. Pommereau, F.: SNAKES: a flexible high-level Petri nets library. In: Proc. of PETRI NETS'15. LNCS, vol. 9115. Springer (2015)
15. Pommereau, F.: ABCD: a user-friendly language for formal modelling and analysis. In: Proc. of PETRI NETS 2016. LNCS, vol. 9698. Springer (2016)
16. Pommereau, F., de la Houssaye, J.: Supplementary material. http://github.com/fpom/PETRINETS-2017-supplementary
17. Rodola, G.: A cross-platform process and system utilities module for Python. http://github.com/giampaolo/psutil
18. The Theoretical Foundations Group of the Department for Informatics of the University of Hamburg: Renew, the reference net workshop — highlights. http://www.informatik.uni-hamburg.de/TGI/renew/highlights.html
19. Wang, B., Zhao, C.: A Petri net simulation kernel with extendibility, convenient modeling and fast simulation engine. In: Proc. of ICCT 2003. vol. 2. IEEE (2003)