Under consideration for publication in Formal Aspects of Computing

GPU-accelerated Steady-state Computation of Large Probabilistic Boolean Networks

Andrzej Mizera

Department of Infection and Immunity
Luxembourg Institute of Health
&
Luxembourg Centre for Systems Biomedicine
University of Luxembourg
e-mail: andrzej.mizera@uni.lu

Jun Pang

Faculty of Science, Technology and Communication &
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
e-mail: jun.pang@uni.lu

Qixia Yuan

Faculty of Science, Technology and Communication
University of Luxembourg
e-mail: qixia.yuan@uni.lu

Abstract. Computation of steady-state probabilities is an important aspect of analysing biological systems modelled as probabilistic Boolean networks (PBNs). For small PBNs, efficient numerical methods to compute steady-state probabilities of PBNs exist, based on the Markov chain state-transition matrix. However, for large PBNs, numerical methods suffer from the state-space explosion problem since the state-space size is exponential in the number of nodes in a PBN. In fact, the use of statistical methods and Monte Carlo methods remain the only feasible approach to address the problem for large PBNs. Such methods usually rely on long simulations of a PBN. Since slow simulation can impede the analysis, the efficiency of the simulation procedure becomes critical. Intuitively, parallelising the simulation process is the ideal way to accelerate the computation. Recent developments of general purpose graphics processing units (GPUs) provide possibilities to massively parallelise the simulation process. In this work, we propose a *trajectory-level parallelisation framework* to accelerate the computation of steady-state probabilities in large PBNs with the use of GPUs. To maximise the computation efficiency on a GPU, we develop a dynamical data arrangement mechanism for

Correspondence and offprint requests to: Jun Pang, Computer Science and Communications, Faculty of Science, Technology and Communication, University of Luxembourg. Postal address: 6, avenue de la Fonte, L-4362 Esch-sur-Alzette, Luxembourg.
A preliminary version of this work was presented at the 2nd International Symposium on Dependable Software Engineering: Theories, Tools, and Applications [MPY16c].

handling different size PBNs with a GPU. Specially, we propose a *reorder-and-split* method to handle both large and dense PBNs. Besides, we develop a specific way of storing predictor functions of a PBN and the state of the PBN in the GPU memory. Moreover, we introduce a strongly connected component (SCC)-based network reduction technique to further accelerate the computation speed. Experimental results show that our GPU-based parallelisation gains approximately a 600-fold speedup for a real-life PBN compared to the state-of-the-art sequential method.

Keywords: Probabilistic Boolean networks, biological networks, computational modelling, discrete-time Markov chains, simulation, statistical methods, graphics processing unit (GPU).

1. Introduction

Systems biology aims to model and analyse biological systems using mathematical and computational methods from a holistic perspective in order to provide a comprehensive, system-level understanding of cellular behaviour. Advances in biology provide new biological knowledge and promote the interest in the comprehension of the functioning of larger and larger cellular systems. This requires the use of computational modelling and formal analysis, which faces a significant challenge due to the state-space size of the system under study. Developed in 2002 by Shmulevich et al. [SD10, TMP$^+$13], probabilistic Boolean networks (PBNs) is a well-suited framework for modelling large-size biological systems. Originally, the framework of PBNs is introduced as a probabilistic generalisation of the standard Boolean networks (BNs) framework to model gene regulatory networks (GRNs). The framework of BNs can incorporate rule-based dependencies between genes and allow the systematic study of global network dynamics. The framework of PBNs not only has the advantage of BNs, but also is capable of dealing with uncertainty, which naturally occurs in the study of biological systems.

One of the key aspects of analysing biological systems, especially those modelled as PBNs, is the comprehensive understanding of their long-run (steady-state) behaviour. This is vital in many contexts, e.g., attractors of a GRN are considered to characterise cellular phenotypes [Kau69]. There have been a lot of studies in analysing the steady-state behaviours of biological systems modelled as PBNs. As the dynamics of a PBN can be viewed as a discrete-time Markov chain (DTMC), they can be studied with the use of the rich theory of DTMCs. Relying on this, many numerical methods exist to compute steady-state probabilities for small-size PBNs [SGH$^+$03, TMP$^+$14]. In the case of large-size PBNs, however, numerical methods face the state-space explosion problem. The use of statistical methods and Monte Carlo methods is then proposed to estimate the steady-state probabilities. These methods require simulating the PBN under study for a certain length and the simulation speed is an important performance factor of these approaches. For large PBNs and long trajectories, a slow simulation speed could render these methods infeasible as well. For example, it would be too slow to compute a steady-state probability with a runtime of more than one month. Therefore, we aim to improve the computation speed for estimating the steady-state probability of a set of states in a PBN. We expect to have speedups of hundreds of times compared to existing state-of-the-art methods. A natural way to address this problem is to parallelise the simulation process.

Recent improvements in the computing power of the general purpose graphics processing units (GPUs) enable massive parallelisation of this process. In this work, we propose a *trajectory-level parallelisation framework* to accelerate the computation of steady-state probabilities in large PBNs with the use of GPUs. The architecture of a GPU is very different from that of a central processing unit (CPU) and the performance of a GPU-based program is highly related to how the synchronisation between cores is implemented and how memory access is managed. Our framework reduces the time needed for synchronisation by allowing each core to simulate one trajectory. Regarding the memory management, we contribute in four aspects. We first develop a dynamical data arrangement for handling different size PBNs with a GPU to maximise the computation efficiency on a GPU for relatively small-size PBNs. We then propose a specific way of storing predictor functions of a PBN and the state of the PBN in the GPU memory to reduce the memory consumption and to improve the access speed. Thirdly, we take special care of large and dense networks using our reorder-and-split method so that our parallelisation framework can efficiently handle large and dense

networks. Lastly, we develop a network reduction technique which can significantly reduce the unnecessary memory usage as well as the amount of required computations. We show with experiments that our GPU-accelerated parallelisation gains a speedup of more than two orders of magnitude compared to the state-of-the-art sequential method.

Structure of the paper. We present preliminaries on PBNs and the architecture of GPUs in Section 2. The difficulties of parallelising the simulation of a PBN and how to overcome them are discussed in Section 3. We then propose a strongly connected component (SCC)-based network reduction technique in Section 4. We evaluate our GPU implementation in Section 5 and conclude our paper with some discussions in Section 6.

2. Preliminaries

2.1. Probabilistic Boolean networks (PBNs)

A PBN $G(X, \mathcal{F}, C)$ consists of a list of binary-valued nodes (also known as genes) $X = (x_1, x_2, \ldots, x_n)$, a list of function sets $\mathcal{F} = (F_1, F_2, ..., F_n)$, and a list of probability distributions $C = (C_1, C_2, \ldots, C_n)$. For each $i \in \{1, 2, ..., n\}$, the set $F_i = \{f_1^{(i)}, f_2^{(i)}, \ldots, f_{\ell(i)}^{(i)}\}$ is a collection of Boolean functions, known as *predictor functions* of node x_i, where $\ell(i)$ is the number of predictor functions for node x_i. Each $f_j^{(i)}$ is a Boolean function defined using a subset of the nodes, referred to as *parent nodes* of x_i. At each time point t, the value of each node x_i is updated with one of its predictor functions. The predictor function is selected in accordance with a probability distribution $C_i = (c_1^{(i)}, c_2^{(i)}, \ldots, c_{\ell(i)}^{(i)})$, where the individual probabilities are the *selection probabilities* for the respective elements of F_i and they sum to 1. We use $x_i(t)$ to denote the value of node x_i at time point t, and $s(t) = (x_1(t), x_2(t), \ldots, x_n(t))$ to denote the state of the PBN at time point t. The state space of the PBN is $S = \{0, 1\}^n$ and it is of size 2^n. PBNs can be divided into two types according to the way of selecting predictor functions: *independent* PBNs and *dependent* PBNs. In independent PBNs, the predictor functions for different nodes are chosen independently of each other; while in dependent PBNs, the predictor functions for all the nodes are chosen simultaneously. The selected predictor functions for all the nodes form one *context*. Due to the dependent relationship in selecting the predictor functions, a dependent PBN has a context switching probability distribution specifying the probability of a context being selected. In this paper, we focus on accelerating the computation of steady-state probabilities of independent PBNs. However, our method can be easily applied to dependent PBNs by slightly changing the process for selecting the predictor functions: we select predictor functions for all the nodes simultaneously based on the context switching probability distribution instead of the selection probability distribution C_i. In general, the update of a PBN can be performed in two different modes, i.e., *synchronous* mode and *asynchronous* mode.

- In the synchronous mode, the values of all the nodes are updated synchronously at each time step. The transition from state $s(t)$ to state $s(t+1)$ is performed by randomly selecting a predictor function for each node x_i from F_i and by applying those selected predictor functions to update the values of all the nodes synchronously. Let $f(t)$ be the combination of all the selected predictor functions at time point t. The transition of state $s(t)$ to $s(t+1)$ can then be denoted as

$$s(t+1) = f(t)(s(t)). \tag{1}$$

- In the asynchronous mode, one node is randomly selected at each time step (normally with uniform distribution) and only the value of this selected node is updated. The transition from state $s(t)$ to state $s(t+1)$ is performed by three steps: first randomly selecting a node, then randomly selecting a predictor function for the selected node, and lastly applying the selected predictor function to update the value of the selected node. Let $f_i(t)$ be the randomly selected function of the randomly selected node x_i at time point t and $s_i(t)$ be the parent nodes of function $f_i(t)$. The transition of state $s(t)$ to $s(t+1)$ can then be denoted as

$$s(t+1) = (x_1(t), x_2(t), \ldots, x_{i-1}(t), f_i(t)(s_i(t)), x_{i+1}(t), \ldots, x_n(t)). \tag{2}$$

A PBN can therefore be viewed as a discrete-time Markov chain (DTMC) with state space $S = \{0, 1\}^n$ and transition relation defined by either Equation 1 or Equation 2.

In a PBN with *perturbations*, a perturbation probability $p \in (0, 1)$ is introduced and the dynamics of

a PBN is guided with both perturbations and predictor functions: at each time point t, the value of each node x_i is flipped independently with probability p; and if no flip happens, either the value of each node x_i is updated with selected predictor functions synchronously in the synchronous update mode or the value of a randomly selected node is updated with the selected predictor function in the asynchronous update mode. Let $\boldsymbol{\gamma}(t) = (\gamma_1(t), \gamma_2(t), \ldots, \gamma_n(t))$ be a perturbation vector, where each element is an independent Bernoulli distributed random variable with parameter p, i.e., $\gamma_i(t) \in \{0, 1\}$ and $\mathbb{P}(\gamma_i(t) = 1) = p$ for all t and $i \in \{1, 2, \ldots, n\}$. Extending Equation 1, the transition from $s(t)$ to $s(t+1)$ in synchronous PBNs with perturbations is given by

$$s(t+1) = \begin{cases} s(t) \oplus \gamma(t) & \text{if } \gamma(t) \neq 0 \\ f(t)(s(t)) & \text{otherwise,} \end{cases} \tag{3}$$

where $\oplus$ is the element-wise exclusive or operator for vectors. And extending Equation 2, the transition from $s(t)$ to $s(t+1)$ in asynchronous PBNs with perturbations is given as

$$s(t+1) = \begin{cases} s(t) \oplus \gamma(t) & \text{if } \gamma(t) \neq 0 \\ (x_1(t), x_2(t), \ldots, x_{i-1}(t), f_i(t)(s_i(t)), x_{i+1}(t), \ldots, x_n(t)) & \text{otherwise.} \end{cases} \tag{4}$$

According to Equations 3 and 4, from any state the system can move to any other state with one transition due to perturbations. Therefore, the underlying Markov chain is in fact irreducible and aperiodic. Thus, the dynamics of a PBN with perturbations can be viewed as an ergodic DTMC [SD10]. Based on the theory of ergodic DTMCs, the long-run dynamics of a PBN with perturbations is governed by a unique limiting distribution, convergence to which is independent of the choice of the initial state.

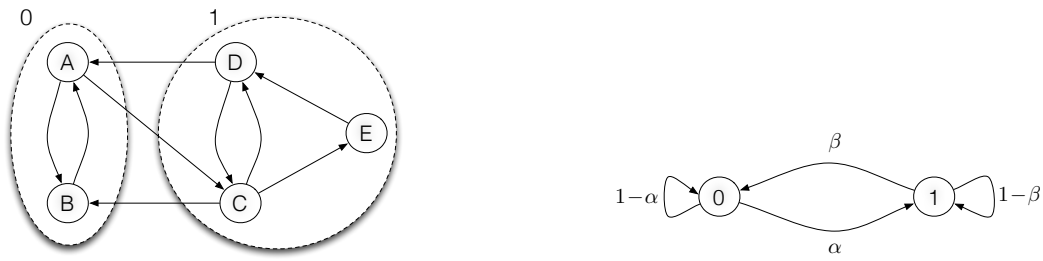
The density of a PBN is measured with the number of predictor functions and the number of parent nodes for each predictor function. For a PBN G, its density is defined as $\mathcal{D}(G) = \frac{1}{n}\sum_{i=1}^{M} \phi(i)$, where n is the number of nodes in G, M is the total number of predictor functions in G, and $\phi(i)$ is the number of parent nodes for the ith predictor function.

2.2. Steady-state Analysis of PBNs

The steady-state probability of a PBN can be computed via either a numerical method or a simulation-based method. However, when it comes to a large PBN, only the simulation-based method is feasible. We briefly introduce a simulation-based method called the two-state Markov chain approach [RL92] for steady-state computation of a PBN.

The two-state Markov chain approach [RL92] is a method for approximate computation of the steady-state probability for a subset of states of a DTMC. This approach splits the states of an arbitrary DTMC into two parts, referred to as two meta states. One part is composed of the states of interest, labelled 1, and the other part is its complement, labelled 0. Denote the steady-state probability of meta state 1 as q. One can estimate q by performing simulations of the original Markov chain. To make the estimation, an abstraction of the original DTMC into a two-state Markov chain is required. Let $\{Z_t\}_{t\geqslant 0}$ be a binary 0-1 process, where Z_t represents the meta state in which the original Markov chain is at time t. In general, $\{Z_t\}_{t\geqslant 0}$ is not a Markov chain. However, as argued in [RL92], it is reasonable to assume that the dependency in $\{Z_t\}_{t\geqslant 0}$ falls off rapidly with lag. Hence, a new process $\{Z_t^{(k)}\}_{t\geqslant 0}$, where $Z_t^{(k)} = Z_{tk}$, will be approximately a first-order Markov chain for k large enough. A procedure for determining appropriate k is given in [RL92]. The first-order Markov chain consists of the two meta states with transition probabilities α and β between them. We illustrate in Fig. 1 the construction of a two-state Markov chain from a 5-state Markov chain.

The estimation of q, denoted as $\hat{q}$, can be obtained by performing simulation of the original DTMC. The key point is to determine a suitable length of the trajectory. Two requirements need to be considered. Firstly, the abstracted two-state Markov chain should converge close enough to the steady-state distribution $\pi = (\pi_0, \pi_1)$. Formally, we require a t large enough such that $|\mathbb{P}[Z_t^{(k)} = i \,|\, Z_0^{(k)} = j] - \pi_i| < \epsilon$ for a given $\epsilon > 0$ and all $i, j \in \{0, 1\}$. t is known as the "burn-in" period where the samples should be discarded. As given in [RL92], the value of t is given by a function $m(\alpha, \beta)$, where $m(\alpha, \beta) = \log\left(\frac{\epsilon(\alpha+\beta)}{\max(\alpha,\beta)}\right) / \log\left(|1-\alpha-\beta|\right)$. Secondly, the estimation $\hat{q}$ should satisfy $\mathbb{P}[q - r \leqslant \hat{q} \leqslant q + r] \geqslant s$, where r is the required precision and s is a specified confidence level. This condition determines the sample size n required to compute $\hat{q}$. Similar to



(a) Original DTMC (b) Two-state DTMC

Fig. 1. Conceptual illustration of the two-state Markov chain construction. (a) The state space of the original DTMC is split into two meta states: meta state 0 (states A and B) and meta state 1 (states D, C and E). The two meta states are marked with dashed ellipses. (b) Projecting the behaviour of the original chain on the two meta states results in a binary (0-1) stochastic process which can be approximated as a first-order, two-state Markov chain.

Algorithm 1 The Two-state Markov chain approach

procedure ESTIMATEPROBABILITY(m_0, n_0, ϵ, r, s)
  $\mathcal{M} := m_0$; $\mathcal{N} := n_0$; $l := \mathcal{M} + \mathcal{N}$;
  generate an initial trajectory of length l abstracted to the two meta states;
  repeat
    extend the trajectory by $\mathcal{M} + \mathcal{N} - l$ states;
    $l := \mathcal{M} + \mathcal{N}$;
    estimate α, β based on the last $\mathcal{N}$ elements of the extended trajectory;
    $\mathcal{M} := \left\lceil \log\left(\frac{\epsilon(\alpha+\beta)}{\max(\alpha,\beta)}\right) / \log\left(|1-\alpha-\beta|\right) \right\rceil$, $\mathcal{N} := \left\lceil \frac{\alpha\beta(2-\alpha-\beta)}{(\alpha+\beta)^3} \frac{\left(\Phi^{-1}(\frac{1}{2}(1+s))\right)^2}{r^2} \right\rceil$
  until $\mathcal{M} + \mathcal{N} \leq l$
  estimate the probability of meta state 1 from the last $\mathcal{N}$ elements of the trajectory.
end procedure

the burn-in period, the sample size is given by a function $n(\alpha, \beta)$, where $n(\alpha, \beta) = \frac{\alpha\beta(2-\alpha-\beta)}{(\alpha+\beta)^3} \frac{\left(\Phi^{-1}(\frac{1}{2}(1+s))\right)^2}{r^2}$. Considering the two conditions together, we can get the total required trajectory length of the original DTMC as $M + N$, where $M = 1 + (\lceil m(\alpha, \beta) \rceil - 1)k$ and $N = 1 + (\lceil n(\alpha, \beta) \rceil - 1)k$. Often, $k = 1$ is large enough. For simplification, we assume $k = 1$ in the following of this study. However, in the real case, we follow the procedure in [RL92] to estimate an appropriate value of k. Denote the burn-in steps and sample size as $\mathcal{M}$ and $\mathcal{N}$ respectively, given $k = 1$. We outline the steps in Algorithm 1. The two arguments m_0 and n_0 are the initial burn-in period and the initial sample size, respectively. In each iteration of the algorithm, the burn-in steps $\mathcal{M}$ and the sample size $\mathcal{N}$ are re-estimated. The iteration continues until $\mathcal{M} + \mathcal{N}$ is not bigger than the current trajectory length. For more details on this approach, a derivation of the formulas for $m(\alpha, \beta)$ and $n(\alpha, \beta)$, and a discussion regarding a proper choice of n_0, we refer to [MPY15b]. Note that the initial choice of m_0 can be arbitrary, as it does not affect the accuracy of the final estimation.

2.3. GPU architecture

We review the basics of GPU architecture and its programming approach, i.e., the common unified device architecture (CUDA) released by NVIDIA.

At the physical hardware level, an NVIDIA GPU usually contains tens of streaming multiprocessors (SMs, also abbreviated as MPs), each containing a fixed number of streaming processors (SPs), a fixed number of *registers*, and fast *shared memory* as illustrated in Figure 2, with N being the number of MPs.

Accessing registers and shared memory is fast, but the size of these two types of memory is very limited. In addition, a large size *global memory*, a small size *texture memory*, and *constant memory* are available outside the MPs. Global memory has a high bandwidth (240 GB per second in our GPU), but also a high

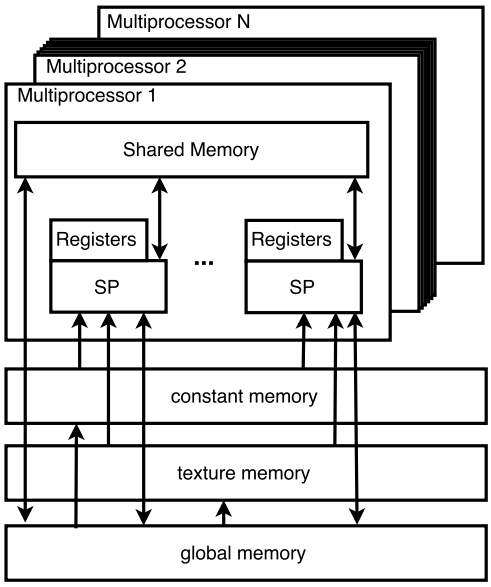


Fig. 2. Architecture of a GPU.

latency. Accessing global memory is usually orders of magnitude slower than accessing registers or shared memory. Constant memory and texture memory are memories of special type which can only store read-only data. Accessing constant memory is most efficient if all threads are accessing exactly the same data, while texture memory is better for dealing with random access. We refer to registers and shared memory as *fast memory*; global memory as *slow memory*; and constant memory and texture memory as *special memory*.

At the programming level, the programming interface CUDA is in fact an extension of C/C++. A segment of code to be run in a GPU is put into a function called a *kernel*. The kernels are then executed as a grid of blocks of threads. A thread is the finest granularity in a GPU and each thread can be viewed as an execution of the kernel. A block is a group of threads executed together in a batch. Each thread is executed in an SP and threads in a block can only be executed in one MP. One MP, however, can launch several blocks in parallel. Communications between threads in the same block are possible via shared memory. NVIDIA GPUs use a processor architecture called single instruction multiple thread (SIMT), i.e., a single instruction stream is executed on a group of 32 threads, called a *warp*. Threads within a warp are bound together, i.e., they always execute the same instruction. Therefore, branch divergence can occur within a warp: if one thread within a warp moves to the 'if' branch of an 'if-then-else' sentence and the others choose the 'else' branch, then actually all the 32 threads will "execute" both branches, i.e., the thread moving to the 'if' branch will wait for other threads when they execute the 'else' branch and vice versa. If both branches are long, then the performance penalty is huge. Therefore, branches should be avoided as much as possible for reasons of performance. Moreover, the data accessing pattern of the threads in a warp should be taken care of as well. We consider the access pattern of shared memory and global memory in this work. Accessing shared memory is most efficient if all threads in a warp are fetching data in the same position or each thread is fetching data in a distinct position. Otherwise, the speed of accessing shared memory is reduced by the so-called *bank conflict*. Accessing global memory is most efficient if all threads in a warp are fetching data in a *coalesced* pattern, i.e., all threads in a warp are reading data in adjacent locations in global memory. In principle, the number of threads in a block should always be an integral multiple of the warp size due to the SIMT architecture; and the number of blocks should be an integral multiple of the number of MPs since each block can only be executed in one MP.

An important task for GPU programmers is to hide latency. This can be done via the following four ways:

1. increase the number of active warps;
2. reduce the access to global memory by caching the frequently accessed data in fast memory, or in constant memory or texture memory, if the access pattern is suitable;

3. reduce bank conflict of shared memory access;
4. coalesce accesses to the global memory to use the bandwidth more efficiently.

However, the above four methods often compete with one another due to the restrictions of the hardware resources. For example, using more shared memory would restrict the number of active blocks and hence the number of active warps is limited. Therefore, a trade-off between the use of fast memory and the number of threads has to be considered carefully. We discuss this problem and provide our solution to it in Section 3.2.

3. PBN Simulation in a GPU

In this section, we present how simulation of a PBN is performed in a GPU, while addressing the problems identified at the end of Section 2. More specifically, we discuss in Subsections 3.1–3.3 how in general the simulation of a PBN can be performed efficiently in a GPU; in Subsection 3.4, we take special care of large and dense PBNs, and demonstrate our reorder-and-split method for handling the large memory required in the dense network.

3.1. Trajectory-level parallelisation

In general, there are two ways of parallelising the PBN simulation. One way is to update all nodes synchronously, i.e., each GPU thread only updates one node of a PBN; the other way is to simulate multiple trajectories simultaneously. The first way requires synchronisation among the threads, which is time-consuming in the current GPU architecture. Besides, this approach does not work for the asynchronous update mode since only one node is updated at each time point. Therefore, in our implementation, we take the second way and simulate multiple trajectories concurrently. In order to use samples from multiple trajectories to compute the steady-state probabilities of a PBN, we propose to combine the Gelman & Rubin method [GR92] with the two-state Markov chain approach [RL92, MPY17].

The Gelman & Rubin method [GR92] is an approach for monitoring the convergence of multiple trajectories. It starts from simulating 2ψ steps of $\omega \geq 2$ independent trajectories in parallel. The first ψ steps of each trajectory, known as the burn-in period, are discarded from it. The last ψ elements of each trajectory are used to compute the within-chain (W) and between-chain (B) variance[1], which are used to estimate the variance of the steady state distribution ($\hat{\sigma}^2$). Next, the potential scale reduction factor $\hat{R}$ is computed with $\hat{\sigma}^2$. $\hat{R}$ indicates the convergence to the steady state distribution. The trajectories are considered as converged and the algorithm stops if $\hat{R}$ is close to 1; otherwise, ψ is doubled, the trajectories are extended, and $\hat{R}$ is recomputed. We list the steps of this approach in Algorithm 2. This algorithm takes the number of trajectories ω and an initial length ψ_0 as input, and has two outputs. The first output is ω trajectories and the second output is the length of the final ψ, which equals to half of each trajectory's length. For further details of this method and the discussion on the choice of the initial states for the ω trajectories we refer to [GR92].

Once convergence is reached, the second halves of the trajectories are merged into one sample, and the two-state Markov chain approach is applied to estimate the required sample length L based on the merged sample. Since the convergence is assured, we propose to skip the iterative computation of the burn-in period in the two-state Markov chain approach to maximise the speed-up. We assume that the two-state Markov chain abstraction has also converged to its steady-state distribution when the simulated trajectories of the original Markov chain have converged. Denote the current burn-in period of the two-state Markov chain approach as m. Then initially, m equals to ψ, the burn-in size of the Gelman & Rubin method. Now the stop criterion for the two-state Markov chain approach becomes that the estimated required sample length L is not bigger than the actual size of the merged sample. If the stop criterion is not satisfied, the multiple trajectories are extended in parallel to provide a sample of required length. The above assumption holds in most cases based on our computational experiments (data not shown); however, it does not hold in general. Hence, we add another step to guarantee that the two-state Markov chain abstraction in each trajectory

[1] We use within-chain and between-chain instead of within-trajectory and between-trajectory because within-chain and between-chain are commonly used in the literature

Algorithm 2 The Gelman & Rubin method

procedure GENERATECONVERGEDCHAINS(ω, ψ_0)
  $\psi := \psi_0$;
  $chains$:= generate in parallel ω trajectories of length ψ;
  repeat
    Extend each trajectory in $chains$ by ψ;
    for $i = 1..\omega$ **do**
      μ_i := mean of the last ψ values of $chains[i]$;
      s_i := standard deviation of the last ψ values of $chains[i]$;
    end for
    $\mu := \frac{1}{\omega}\sum_{i=1}^{\omega}\mu_i$;
    $B := \frac{\psi}{\omega-1}\sum_{i=1}^{\omega}(\mu_i-\mu)^2$; $W := \frac{1}{\omega}\sum_{i=1}^{\omega}s_i^2$; *//Between and within variance*
    $\hat{\sigma}^2 := (1-\frac{1}{\psi})W + \frac{1}{\psi}B$; *//Estimate the variance of the stationary distribution*
    $\hat{R} := \sqrt{\hat{\sigma}^2/W}$; *//Compute the potential scale reduction factor*
    $\psi := 2\cdot\psi$;
  until $\hat{R}$ is close to 1
  return $(chains, \psi/2)$; *//ψ equals to the length of each chain*
end procedure

Algorithm 3 The Parallelised two-state Markov chain approach

procedure ESTIMATEINPARALLEL($\omega, \psi_0, \epsilon, r, s$)
  $(chains, \psi)$:= GENERATECONVERGEDCHAINS(ω, ψ_0);
  $m := \psi$; $n := \psi$; $E := 0$; ab_sample := NULL;
  repeat
    repeat
      $chains$:= extend in parallel each trajectory in $chains$ by E samples;
      $n := n + E$; $sample\_size := \omega \cdot n$;
      $sample := chains(m{+}1, \ldots, m{+}n)$; *//obtain samples between m+1 and m+n in all trajectories*
      abstract $sample$ to the 0-1 chain and append it to ab_sample ;
      Estimate α, β from ab_sample;
      Compute $\mathcal{N}$ as in Line 8 Algorithm 1;
      $E := \lceil (sample\_size - \mathcal{N})/\omega \rceil$;
    until $E \leq 0$
    compute $\mathcal{M}$ as in Line 8 Algorithm 1;
    if $\mathcal{M} > m$ **then**
      $E := \mathcal{M} - m$; $m := \mathcal{M}$; $n := n - E$;
    end if
  until $E \leq 0$
  Estimate the prob. of meta state 1 from ab_sample;
end procedure

has also converged. Once the stop criterion is satisfied, we compute the burn-in period $\mathcal{M}$ of the two-state Markov chain. The assumption is verified true if $\mathcal{M}$ is smaller or equal to m. In the other case, we continue to discard additional $\mathcal{M} - m$ elements in the beginning of each trajectory and re-run the two-state Markov chain approach based-on the modified samples. Notice that the number of the burn-in steps does not depend on the number of trajectories, but only on the values of α and β in the two-state Markov chain. Given the real values (or a good estimate) of α and β, the number of burn-in steps is fixed. We describe this process in Algorithm 3 and refer to [MPY16d] for a detailed description of this combination. Note that merging is performed in a CPU and no synchronization is required. We show in Figure 3 the workflow for computing steady-state probabilities based on trajectory-level parallelisation.

Each shaded box represents a kernel to be parallelised in a GPU. The first and second shaded boxes perform the same task except that trajectories in the first shaded box are abandoned while those in the

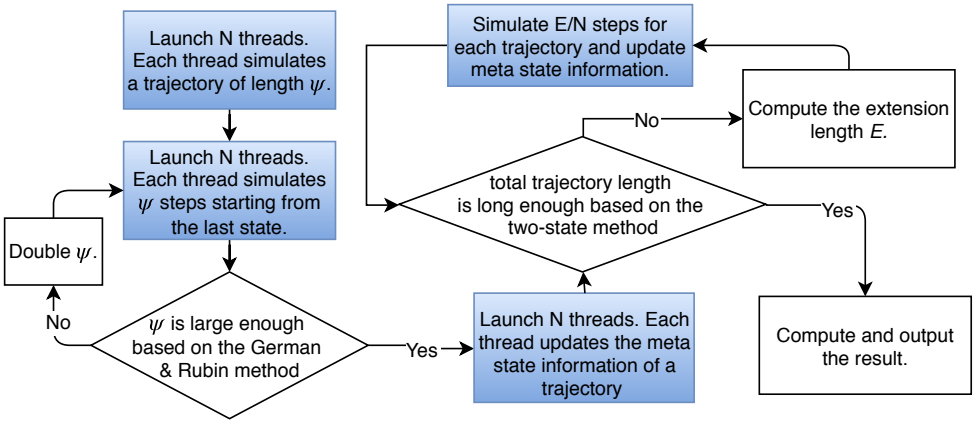


Fig. 3. Workflow of steady-state analysis using trajectory-level parallelisation.

second shaded box are stored in global memory. The second shaded box may be iterated several times and only samples generated in the last iteration will be used for estimating the steady-state probabilities. This is due to the requirement of the Gelman & Rubin method [GR92] that only the second half samples in each trajectory (as shown in Line 8 of Algorithm 3) are used for computing steady-state probabilities. Based on the last ψ samples simulated in the second shaded box, the third shaded box computes the *meta state* information required by the two-state Markov chain approach [MPY15b]. The two-state Markov chain approach determines whether the samples are large enough based on the meta state information. If not enough, the last, fourth kernel is called repeatedly to extend samples; otherwise, the steady-state probability is computed.

The key part of the four kernels is the simulation process. We describe in Algorithm 4 the process for simulating one step of a PBN in a GPU. The four inputs of this algorithm are respectively the number of nodes n, the Boolean functions F, the extra Boolean functions $extraF$, and the current state S. The extra Boolean functions are generated due to that we optimise the storage of Boolean functions and split them into two parts in order to save memory (see Section 3.3 for details). Due to this optimisation, an 'if' statement (lines 14 to 17) has to be added. This 'if' statement fetches the Boolean function stored in the second part ($extraF$). The probability that this statement is executed is very small due to the way we split the Boolean functions and the time cost of executing this statement is also very small. Therefore, by paying a small penalty in terms of computational time, we are able to store Boolean functions in fast memory and in total gain significant speedups.

3.2. Data arrangement

As mentioned in Section 2.3, a suitable strategy for hiding latency should be carefully considered for a GPU program. Since the simulation process requires accessing the PBN information (in a random way) in each simulation step and the latency cost for frequently accessing data in slow memory is huge, caching this information in fast and special memory results in a more efficient computation compared to allowing more active warps. Therefore, we first try to arrange all frequently accessed data in fast and special memory as much as possible; then, based on the remaining resources we calculate the optimal number of threads and blocks to be launched. Since the size of fast memory is limited and the memory required to store a PBN varies from PBN to PBN, a suitable data arrangement policy is necessary. In this section, we discuss how we dynamically arrange the data in different GPU memories for different PBNs.

In principle, frequently accessed data should be put in fast memory. We list all the frequently used data and how we arrange them in GPU memories in Table 1. As the size of the fast memory is limited and has different advantages for different data accessing modes, we save different data in different memories. Namely, the read-only data that are always or most likely accessed simultaneously by all threads in a warp, are put in constant memory; other read-only data are put in shared memory if possible; and the rest of the data are put in registers if possible. Since the memory required to store the frequently used data varies a lot

Algorithm 4 Simulate one step of a PBN in a GPU

```
1: procedure SimulateOneStep(n, F, extraF, p, S)
2:     perturbed := false;
3:     for (i := 0; i < n; i++) do
4:         if rand() < p then perturbed := true; S[i/32] := S[i/32] ⊕ (1 ≪ (i%32));
5:         end if
6:     end for
7:     if perturbed then return S;
8:     else
9:         set array nextS to 0;
10:        for (i := 0; i < n; i++) do
11:            index := nextIndex(i);                    //sample the Boolean function index for node i
12:            compute the entry of the Boolean function based on index and S;
13:            v := F[index];
14:            if entry > 31 then                                              //entry starts with 0
15:                get index of the Boolean function in extraF;                //see Section 3.3
16:                v := extraF[index]; entry := entry%32;
17:            end if
18:            v := v ≫ entry; nextS[i/32] := nextS[i/32] | ((v&1) ≪ (i%32));
19:        end for
20:    end if
21:    S := nextS; return S.
22: end procedure
```

Table 1. Frequently accessed data arrangement.

data	data type	stored in
random number generator	CUDA built in	registers
node number	integer	constant memory
perturbation probability	float	constant memory
cumulative number of functions	short array	constant memory
selection probabilities of functions	float array	constant memory
indices of positive nodes	integer array	constant memory
indices of negative nodes	integer array	constant memory
cumulative number of parent nodes	short array	shared memory
Boolean functions	integer array	shared memory
indices for extra Boolean functions	short array	shared memory
parent nodes indices for each function	short array	shared/texture memory
current state	integer array	registers/global memory
next state	integer array	registers/global memory

from PBN to PBN, we propose to use a dynamic decision process to determine how to arrange some of the frequently accessed data, i.e., the data shown in the last three rows of Table 1. The dynamic process calculates the memory required to store all the data for a given PBN and determines where to put them based on their memory size. If the shared memory and registers are large enough, all the data will be stored in these two fast memories. Otherwise, they will be placed in the global memory. For the data stored in the global memory, we use two ways to speed up their access. One way is to use texture memory to speed up the access for read-only data, e.g., the parent node indices for each function. The other way is to optimise the data structure to allow a coalesced accessing pattern, e.g., the current state. We explain this in details in Section 3.3. This dynamical arrangement of data allows our program to explore the computational power of a GPU as much as possible, leading to larger speedups for relatively small sparse networks.

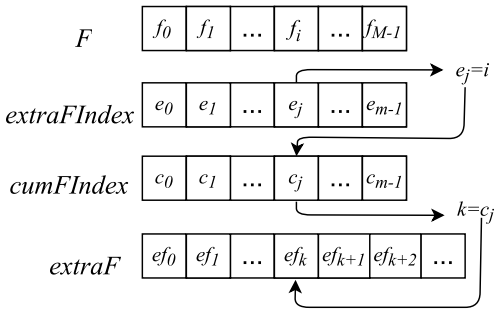


Fig. 4. Demonstration of storing Boolean functions in integer arrays.

3.3. Data optimisation

As mentioned in Section 2.3, a GPU usually has a very limited size of fast memory and the latency can vary significantly depending on how the memory is accessed, e.g., accessing shared memory with or without bank conflict. Therefore, we optimise the data structures for two important pieces of data, i.e., the Boolean functions (stored as truth tables) and the states of a PBN, to save space and to maximise the access speed.

Optimisation on Boolean functions. A direct way to store a truth table is to use a Boolean array, which consumes one byte to store each element. Accessing an element of the truth table can be directly made by providing the index of the Boolean array. Instead, we propose to use a primitive 32-bit integer (4 bytes) type to store the truth table. Each bit of an integer stores one entry of the truth table and hence the memory usage can be reduced by 8 in maximum: 4 bytes compared to 32 bytes of a Boolean array. A 32-bit integer can store a truth table of at most 32 elements, corresponding to a Boolean function with max. 5 parent nodes. Since for real biological systems the number of parent nodes is usually small [HSIO06], in most cases one integer is enough for storing the truth table of one Boolean function. In the case of a truth table with more than 32 elements, additional integers are needed. In order to save memory and quickly locate a specific truth table, we save the additional integers in a separate array. More precisely, we use a 32-bit integer array F of length M to store the truth tables for all the M Boolean functions and the ith ($i \in [0, M-1]$) element of F stores only the first 32 elements of the ith truth table. If the ith truth table contains more than 32 elements, the additional integers are stored in an extra integer array *extraF*. In addition, two index arrays *extraFIndex* and *cumExtraFIndex* are needed to store the index of the ith truth table in *extraF*. Each element of *extraFIndex* stores one index value of the truth table which requires additional integers. The length of *extraFIndex* is at most M. Each element of *cumExtraFIndex* stores the cumulative number of additional required integers for all the truth tables whose indices are stored in *extraFIndex*.

As an example, we show how to store a truth table with 128 elements in Figure 4. We assume that this 128-element truth table is the ith one among all M truth tables and that it is the jth one among those m truth tables that require additional integers to store. Therefore, its first 32 (0-31th) elements are stored in the ith element of F and its index i is stored in the jth element of *extraFIndex*, denoted as e_j. The jth element of *cumExtraFIndex*, denoted as c_j, stores the total number of additional integers required to store the $j-1$ truth tables whose indices are stored in the first $j-1$ elements of *extraFIndex*. Let $cumExtraFIndex[j] = k$. The kth, $(k+1)$th, and $(k+2)$th elements of *extraF* store the 32-127th elements of the ith truth table. After storing the truth tables in this way, accessing the δth element of the ith truth table can be performed in the following way. When $\delta \in [0, 31]$, $F[i]$ directly provides the information and when $\delta \in [32, 127]$, three steps are required: 1) search the array *extraFIndex* to find the index j such that $extraFIndex[j]$ equals to i, 2) fetch the jth value of array *cumFIndex* and let $k = cumFIndex[j]$, 3) the integer $extraF[k + (\delta - 32)/32]$ contains the required information. Since in most cases the number of parent nodes is very limited, the array *extraFIndex* is very small. Hence, the search of the index j in the first step can be finished very quickly. In the rare case where the *extraFIndex* array would be large, e.g., M is large and the length of *extraFIndex* would be close to M, it is preferable to store *extraFindex* as an array of length M and let $extraFindex[i]$ store the entry in $cumFIndex$ for the ith truth table so that the search phase of the first step is eliminated. The required memory for storing this truth table is reduced from 128 bytes (stored as Boolean arrays) to 20 bytes (6 integers to store the truth table and 2 shorts to store the index). In addition to saving memory, the above

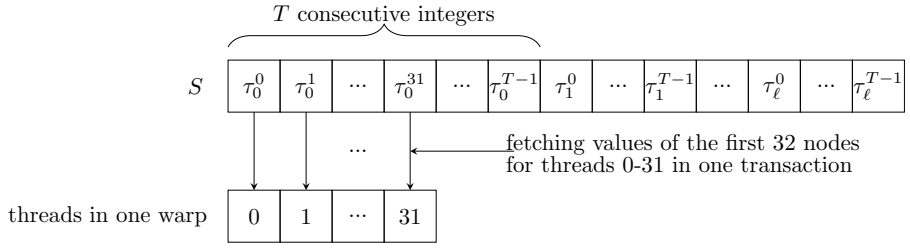


Fig. 5. Storing states in one array and coalesced fetching for threads in one warp.

optimisation can also reduce the chances of bank conflict in shared memory due to the fact that accessing any entry of a truth table is performed by fetching only one integer in array F in most cases. Accessing the elements in *extraFIndex* requires additional memory fetching; however, as mentioned before, the chance for such cases to happen is very small in a real-life PBN and the gained memory space and improved data fetching pattern can compensate for this penalty.

Optimisation on PBN states. The optimisation of the data structure for states is similar to that for Boolean functions, i.e., states are stored as integers and each bit of an integer represents the value of a node. Therefore, a PBN with n nodes requires $\lceil n/32 \rceil$ integers ($4 * \lceil n/32 \rceil$ bytes) to be stored, compared to n bytes when stored as a Boolean array. During the simulation process, the current state and the next state of a PBN have to be stored. As shown in Table 1, the states are put in registers whenever possible, i.e., when the number of nodes is smaller than 129. In the case of a PBN with nodes number equal to or larger than 129, the global memory has to be used due to the limited register size (shared memory are used to store other data and would not be large enough to store states in this case). To reduce the frequency of accessing global memory, one register (32 bits) is used to cache the integer that stores the values of 32 nodes. Updating of the 32 node values is performed via the register and stored in the global memory with a single access only once all the 32 node values are updated in the register. Moreover, states for all the threads are stored in one large integer array S in the global memory and we arrange the content of this array to allow for a coalesced accessing pattern. More specifically, starting from the 0th integer, every consecutive T integers store the values of 32 nodes in the T threads (assuming there are T threads in total). Figure 5 shows how to store states of a PBN with n nodes for all the T threads in an integer array S and how the 32 threads in the first warp fetch the first integer in a coalesced pattern. We denote τ_i^j as the ith integer to store values of 32 nodes for thread j and let $\ell = \lceil n/32 \rceil$. For threads in one warp, accessing the values of the same node can be performed via fetching the adjacent integers in the array S. This results in a coalesced accessing pattern of the global memory. Hence, all the 32 threads in one warp can fetch the data in a single data transaction.

3.4. Node-reordering for large and dense networks

The above mentioned data arrangements and optimisation methods work quite well if the network is relatively sparse or small. However, when the network is both large and dense, the space required for storing the Boolean functions becomes so huge that they cannot be handled by the fast memories. Moreover, when the network is too dense, the number of parent nodes for each Boolean function is very likely to exceed 5. As a result, the Boolean function requires extra integers to be stored as discussed in Section 3.3, leading to inefficient access of Boolean functions.

To overcome the above mentioned problem, we propose a reorder-and-split method to handle the Boolean functions and their parent node indices for a large and dense network. The method consists of the following two steps. First, we reorder the nodes in an ascending order based on the number of their Boolean functions. Secondly, we split the ordered nodes into two parts. This split is based on the available amount of shared memory, i.e., the first part contains the first m nodes, where m is the maximum number of nodes whose Boolean functions and parent node indices can be stored in the fast memory. By reordering the nodes, we put the nodes with fewer Boolean functions in the fast memory. As a result, we can put more nodes in fast memory. Therefore, accessing slow memory in each simulation step is reduced. By splitting, we maximise the usage of the fast memory to store the Boolean functions so that the access to slow memory is minimised. Besides, since the chance that a Boolean function may have more than five parent nodes becomes higher in

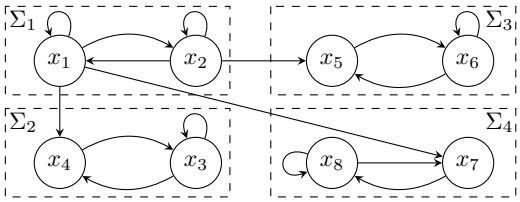


Fig. 6. SCC-based network reduction.

a dense network, it is very likely that *extraF* is required to store a Boolean function if the Boolean functions are stored as discussed in Section 3.3. As GPU instructions in a warp are performed simultaneously, even if only one out of 32 threads is accessing the *extraF*, the other 31 threads have to wait for this access. Therefore, the advantage for the optimisation of storing Boolean functions in Section 3.3 disappears or even becomes a disadvantage. Instead of the above optimisation, we propose to store a Boolean function in consecutive elements of the same array when the elements of a Boolean function is more than 32. In this way, we only need two arrays to store the Boolean function information: one array F to store the Boolean functions and one array *startIndexF* to store the starting index of each Boolean function.

4. Strongly connected component (SCC)-based network reduction

The set of states whose steady-state probability is to be computed is usually specified by specifying the values for a subset of node, referred to as the nodes of interest. The values of the remaining nodes are not considered when computing the steady-state probability. If a non-interest node is not an ancestor node of any node of interest, then such a node would not affect the values of the nodes of interest. We call such a node an *irrelevant* node. Removing the irrelevant nodes will not affect the computation of steady-state probabilities if the perturbations of these irrelevant nodes are considered. In [MPY16b], a leaf-based network reduction method was proposed to remove the irrelevant nodes and hence to reduce the amount of computations required for PBN simulation. In this section, we present a strongly connected component (SCC)-based network reduction technique to improve the performance. Our method differs from the leaf-based network reduction method by removing not only the leaf nodes, but also any other node that does not affect the nodes of interest. In other words, our method can remove all the irrelevant nodes. Please be noted that our SCC-based method takes the SCCs on the level of network structure, i.e., the dependency graph of nodes. Therefore, the detection of SCCs does not face the state-space explosion problem and can be finished quickly. We first give the standard graph-theoretical definition of an SCC:

Definition 1 (SCC). Let $\mathcal{G}$ be a directed graph and $\mathcal{V}$ be its vertices. A strongly connected component (SCC) of $\mathcal{G}$ is a maximal set of vertices $C \subseteq \mathcal{V}$ such that for every pair of vertices u and v, there is a directed path from u to v and vice versa.

We take a PBN G and convert it, i.e., its network structure, to a graph $\mathcal{G}$ by taking the nodes of G as the vertices in $\mathcal{G}$ and by drawing edges from the parent nodes to the child nodes in each of the Boolean functions. We then detect SCCs for the graph $\mathcal{G}$. By treating the SCCs as new vertices, we obtain a new graph which is in fact a directed acyclic graph (DAG). In this DAG, we keep only the following two types of SCCs: either an SCC that contains nodes of interest or an SCC that is an ancestor of a first type SCC. Nodes in the remaining SCCs are removed.

Example 1. Figure 6 shows the graph of a BN with 8 nodes $x_1, x_2, \ldots, x_8$. The BN is decomposed into four SCCs $\Sigma_1, \Sigma_2, \Sigma_3$, and Σ_4. Assume only node x_7 is of interest, then the nodes in the SCC Σ_2 and Σ_3 can be removed since these two SCCs neither contain the nodes of interest nor are ancestors of an SCC with nodes of interest. Notably, the leaf-based network reduction method will not remove any node in this graph since there is no leaf node in this graph.

Let us call the nodes removed by the above mentioned SCC-based network reduction technique as *redundant nodes*. Since those redundant nodes do not affect the nodes of interest, the simulation of the nodes of interest will not be affected in a PBN without perturbations after applying this network reduction technique. In the case of a PBN with perturbations, perturbations of the redundant nodes need to be considered.

Algorithm 5 Checking perturbations of redundant nodes in a PBN

```
1: procedure CheckRedundantNodes(λ)
2:     if rand() > λ then return true;
3:     else return false;
4:     end if
5: end procedure
```

Updating states with Boolean functions will only be performed when there is no perturbation in both the redundant nodes and the non-redundant nodes. Whether at least one redundant node is perturbed can be checked in constant time irrespective of the number of redundant nodes as described in Algorithm 5. The input λ is the probability that no perturbation happens in all the redundant nodes. It is calculated by $\lambda = (1-p)^{\ell}$, where p is the perturbation probability for each node and ℓ is the number of redundant nodes in the PBN. With the consideration of their perturbations, the redundant nodes can be removed without affecting the simulation of the non-redundant nodes also in a PBN with perturbations. Since the redundant nodes are not of interest, results of analyses performed on the simulated trajectories of the reduced network, i.e., containing only non-redundant nodes, will be the same as performed on trajectories of the original network, i.e., containing all the nodes. Algorithm 5 can be easily incorporated into Algorithm 4 with two small changes: firstly, add the parameter λ to the inputs of Algorithm 4 at line 1; secondly, after line 7 of Algorithm 4, add the following line "**if** CheckRedundantNodes(λ) **then return** S;" .

5. Evaluation

We evaluate our GPU-based parallelisation framework for computing steady-state probabilities of PBNs on both randomly generated networks and on a real-life biological network. The evaluation contains three parts. We first evaluate the performance of our framework on randomly generated networks in Section 5.1. This evaluation includes the performance for relatively sparse networks as well as dense networks. Then, we demonstrate the performance of our SCC-based network reduction technique in Section 5.2. Lastly, we evaluate our framework on a real-life biological network. All the experiments are performed on high performance computing (HPC) machines, each of which contains a CPU of Intel Xeon E5-2680 v3 @ 2.5 GHz and an NVIDIA Tesla K80 Graphic Card with 2496 cores @824MHz. The program is written in a combination of both Java and C, and the initial and maximum Java virtual machine heap sizes are set to 4GB and 11.82GB, respectively. The C language is used to program operations on GPUs due to the fact that no suitable Java library is currently provided for programming operations on NVIDIA GPUs. When launching the GPU kernels, the kernel configurations (the number of threads and blocks) are dynamically determined as mentioned in Section 3.2.

5.1. Randomly generated networks

We first evaluate our framework on relatively sparse networks. This evaluation is performed on 380 PBNs, which are generated using the tool ASSA-PBN [MPY15a, MPY16a]. The node numbers of these networks are form the set $\{100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750, 800, 850, 900, 950, 1000\}$. For each of the 380 networks, we compute one steady-state probability using both the sequential two-state Markov chain approach and our GPU-accelerated parallelisation framework. We set the three precision requirements of the two-state Markov chain approach, i.e., the confidence level s, the precision r, and the steady-state convergence parameter ϵ to 0.95, 5×10^{-5}, and 10^{-10} respectively. The computation time limit is set to 10 hours. In the end, we obtain 366 pairs of valid results. The remaining 14 pairs are invalidated due to the time out of the sequential version of the two-state Markov chain approach (not of the parallel version!). We say a valid pair of results is comparable if the difference between the two estimated probabilities is no bigger than $2 \times r$ (10^{-4}). Among the 366 results, 355 (96.99%) are comparable. This number is in agreement with the used confidence level of $s = 95\%$ for the precision of the estimation of the steady-state probabilities: the sequential and the parallel versions independently estimate the probabilities with $s = 95\%$, so the interval $\pm r$ around the estimated value contains the true value in 95% of the cases; therefore, the percentage of comparable pairs should be no less than 90%.

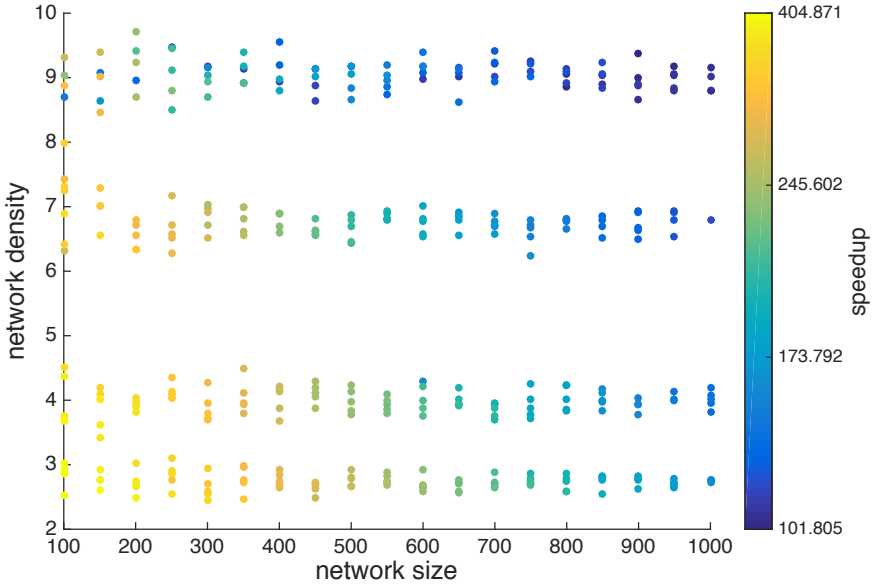


Fig. 7. Speedups of GPU-accelerated steady-state computation.

We compute the speedups of the GPU-accelerated parallelisation framework with respect to the sequential two-state Markov chain approach for those 366 valid results with the formula $speedup = \frac{s_{pa}/t_{pa}}{s_{se}/t_{se}}$, where s_{pa} and t_{pa} are respectively the sample size and time cost of the parallelisation framework, and s_{se} and t_{se} are respectively the sample size and time cost of the sequential approach. The speedups are plotted in Figure 7. As can be seen from this figure, we obtain speedups approximately between 102- and 405-fold. There are some small gaps in the densities of the generated networks, e.g., there are no networks with density between 5 and 6. These gaps are due to the way the networks are randomly generated, i.e., one cannot force the ASSA-PBN tool to generate a PBN with a fixed density, but can only provide the following information to affect the density: the number of nodes, the maximum (minimum) number of functions for each node, and the maximum (minimum) number of parent nodes for each function. However, even with the gaps, the tendency of the changes of speedups with respect to densities can be well observed. In fact, this observation is similar to that for the network size. With the network size decreasing and the density decreasing, our GPU-accelerated parallelisation framework gains higher speedups. This is due to our dynamic way of arranging data for different size PBNs: data for relatively small[2] and sparse networks can be arranged in the fast memory alone. As mentioned in Section 3.3, a state for a network with less than 129 nodes can be stored in fast memory.

To present the details on the obtained results, we select 8 pairs among the 366 results and show in Table 2 the computed probabilities, the sample size (in millions), and the time cost (in seconds) for computing the steady-state probabilities using both the sequential two-state Markov chain approach and the GPU-accelerated parallelisation framework. In all the cases, the numbers of burn-in steps are very small compared to the sample size; therefore, they are not shown in the table. Note that the results of the two methods are shown in columns titled "s." and "–"; the columns titled "+" are used for demonstrating results of the network reduction technique discussed in the next section. The speedup of the GPU-accelerated parallelisation framework with respect to the sequential method is shown in the column titled "–". The two approaches generated comparable results using similar length of samples while our GPU-accelerated parallelisation framework shows speedups of more than two orders of magnitude. All detailed results for the 380 networks can be found at `http://satoss.uni.lu/software/ASSA-PBN/benchmark/`.

[2] In fact all the networks used in this subsection should be called large-size PBNs since the state space of the network with the smallest size has $2^{100} \approx 10^{30}$ states. The networks with 100 nodes are relatively small compared to other networks.

Table 2. Speedups of GPU-accelerated steady-state computation of 8 randomly generated networks. The results shown here are based on 10 runs. The probability, sample size and time are shown in the format of "average(standard deviation)", where average is the average value of the 10 runs and standard deviation is the standard deviation of the 10 runs. The networks are indexed with numbers and the detailed information of each network is shown in the table after. The probability means the steady-state probability for being in one set of randomly selected states; "s." is short for the sequential two-state Markov chain approach; "–" means the GPU-accelerated parallel approach without the network reduction technique applied; and "+" means the GPU-accelerated parallel approach with the network reduction technique applied.

#	probability ($\times 10^{-5}$)			sample size (million)			time (s)			speedup	
	s.	–	+	s.	–	+	s.	–	+	–	+
1	24405(2)	24404(3)	24406(1)	358(4)	358(4)	366(5)	3595(34)	7(1)	4.6(0.3)	401	585
2	8831(2)	8831(3)	8830(1)	150(3)	153(3)	151(3)	939(29)	4(5)	2.9 (0.4)	215	323
3	20528(1)	20529(3)	20528(2)	494(3)	491(3)	494(2)	9833(22)	60(6)	38.2(2.4)	163	257
4	12002(1)	12002(2)	12003(1)	316(3)	316(2)	318(2)	7611(31)	28(3)	15.7(1.4)	276	487
5	13708(1)	13707(3)	13708(1)	542(2)	543(3)	540(3)	14625(117)	125(7)	79.8(2.0)	118	183
6	5797(2)	5798(2)	5797(1)	260(3)	260(3)	259(2)	8590(79)	36(4)	23.1(1.1)	237	371
7	17796(2)	17797(3)	17797(1)	990(3)	996(3)	992(3)	34109(81)	328(4)	214(1.4)	105	160
8	14674(2)	14674(2)	14675(1)	844(4)	843(4)	844(3)	30589(87)	183(4)	108(1.7)	167	283

#	# nodes	# redundant nodes	density	#	# nodes.	# redundant nodes	density
1	100	36	2.53	5	700	231	7.08
2	100	31	7.31	6	700	269	2.64
3	400	131	7.14	7	1000	331	7.09
4	400	148	2.75	8	1000	338	2.73

Table 3. Speedups of GPU-accelerated steady-state computation with the reorder-and-split method applied. As in Table 2, the results shown here are based on 10 runs. "+" means with the reorder-and-split method applied; while "–" menas without the method applied.

# node	density	probability ($\times 10^{-5}$)		sample size ($\times 10^{5}$)		time (s)		speedup
		–	+	–	+	–	+	
500	16.93	10275(3)	10276(2)	3287(3)	3290(6)	513(3)	77.4(0.4)	6.63
600	16.71	8992(2)	8991(3)	3170(6)	3169(6)	634(3)	90.7(0.2)	6.99
700	16.26	13129(2)	13130(3)	5324(8)	5319(1)	1346(9)	180.5(0.9)	7.45
800	16.46	9157(3)	9156(2)	4430(12)	4434(10)	1487(9)	175.8(0.4)	8.47
900	16.39	12739(2)	12739(2)	6694(10)	6693(9)	2736(19)	301(1)	9.09
1000	16.87	16529(3)	16531(3)	9374(19)	9367(9)	4921(18)	479.0(0.4)	10.27

We continue to demonstrate the performance of our framework on large and dense networks. Using the tool ASSA-PBN, we generate 30 large and dense networks whose nodes number are in the set $\{500, 600, 700, 800, 900, 1000\}$. In this evaluation, we compare how our reorder-and-split method as discussed in Section 3.4 performs compared to the cases when the reorder-and-split method is not applied. Therefore, for each of the 30 networks, we compute one steady-state probability using both the GPU-accelerated parallelisation framework with and without the reorder-and-split method applied. The three precision parameters were kept the same as in the previous evaluation. We repeat the computation 10 times. In the end, we get 10 pairs of valid results for each of the 30 networks. We select the results for 6 networks and show them in Table 3. It is obvious from this table that applying our reorder-and-split method can improve the performance of the GPU-accelerated parallelisation framework by several times. Moreover, the improved performance of the reorder-and-split method increases with the number of nodes. This reflects the fact that the advantages of our reorder-and-split method become more pronounced with the network size increased.

Table 4. Speedups of GPU-accelerated steady-state computation of a real-life apoptosis network. As in Table 2, the results shown here are based on 10 runs. The three different types of methods calculate the same probabilities for the given precision; therefore, the probabilities are shown in one column. "s." represents the sequential two-state Markov chain approach; "–" represents the GPU-accelerated parallel approach without applying the network reduction technique; and "+" represents the GPU-accelerated parallel approach with the network reduction technique applied.

steady-state R C F	probability ($\times 10^{-5}$)	sample size (million)			time (s)			speedup	
		s.	–	+	s.	–	+	–	+
0 1 1	324(0)	591(3)	591(3)	593(2)	3906(13)	9.4(0.7)	5.8(0.1)	412	671
1 1 1	99005(0)	1810(3)	1810(2)	1811(1)	11476(17)	28.2(0.6)	17.6(0.6)	406	653
1 0 1	559(0)	1017(3)	1037(4)	1039(6)	6661(3)	15.8(0.9)	10.2(0.5)	431	670
1 1 0	108(0)	198(3)	203(3)	203(3)	1282(1)	3.4(0.9)	1.8(0.3)	385	740
* 1 1	99329(0)	1232(4)	1238(3)	1237(4)	7974(24)	19.3(0.3)	11.9(0.3)	415	675
* 1 0	109(0)	201(3)	205(3)	205(3)	1094(8)	3.4(0.3)	1.9(0.3)	328	585
* 0 1	562(0)	1030(4)	1037(2)	1037(2)	6737(22)	16.2(0.3)	9.8(0.3)	418	691

5.2. Performance of SCC-based network reduction

In this section, we evaluate the performance of our SCC-based network reduction technique. We use the 8 selected networks shown in Table 2 to perform this evaluation. We calculate 8 steady-state probabilities of the 8 networks using the GPU-accelerated parallelisation framework with the SCC-based network reduction technique applied and show the results in columns with title "+". In the last column, we show the speedup of the parallelisation framework with the SCC-based network reduction technique applied with respect to the sequential two-state Markov chain approach. It is calculated based on the formula $speedup_{SCC} = \frac{s_{SCC}/t_{SCC}}{s_{se}/t_{se}}$, where s_{SCC} and t_{SCC} are respectively the sample size and time cost of the parallelisation framework with the SCC-based network reduction technique applied and s_{se} and t_{se} are respectively the sample size and time cost of the sequential approach. The results in Table 2 show that, by applying our SCC-based network reduction technique, the performance of our GPU-accelerated framework can be further improved. Given around 35% of redundant nodes, the computational speed is improved by about 1.6 times (the number is obtained by comparing the speedups in the last two columns). We made similar observation for the real-life network demonstrated in the next section.

5.3. An apoptosis network

We have analysed a PBN model of an apoptosis network using the sequential two-state Markov chain approach in [MPY15b]. The apoptosis network was originally published in [SSV+09] as a BN model and cast into the PBN framework in [TMP+14]. The PBN model (as shown in Figure 8) contains 91 nodes and 107 Boolean functions. The selection probabilities of the Boolean functions were fitted to experimental data in [TMP+14]. We took the 20 best fitted parameter sets and performed the influence analyses for them. Although we managed to finish this analysis in an affordable amount of time due to an efficient implementation of a sequential PBN simulator, the analysis was still very expensive in terms of computation time since the required trajectories were very long and we needed to compute steady-state probabilities for a number of different states.

In this work, we re-perform part of the influence analyses from [MPY15b] using our GPU-accelerated parallel two-state Markov chain approach. In the influence analysis, we consider the PBN with the best fitted values and we aim to compute the *long-term influences* on complex2 from each of its parent nodes: RIP-deubi, complex1, and FADD, in accordance with the definition in [SDKZ02]. In order to compute this long-term influence, seven different steady-state probabilities are required. We show in the first column of Table 4 the values of the nodes of interest for seven steady-state probabilities. The three numbers or "*" with two numbers respectively represent the values of the three genes RIP-deubi, complex1, and FADD: 0 represents active; 1 represents inactive; and "*" represents irrelevant. We compute the seven different steady-state probabilities using three different methods: the sequential two-state Markov chain approach, the GPU-accelerated parallelisation framework without the SCC-based network reduction technique applied, and

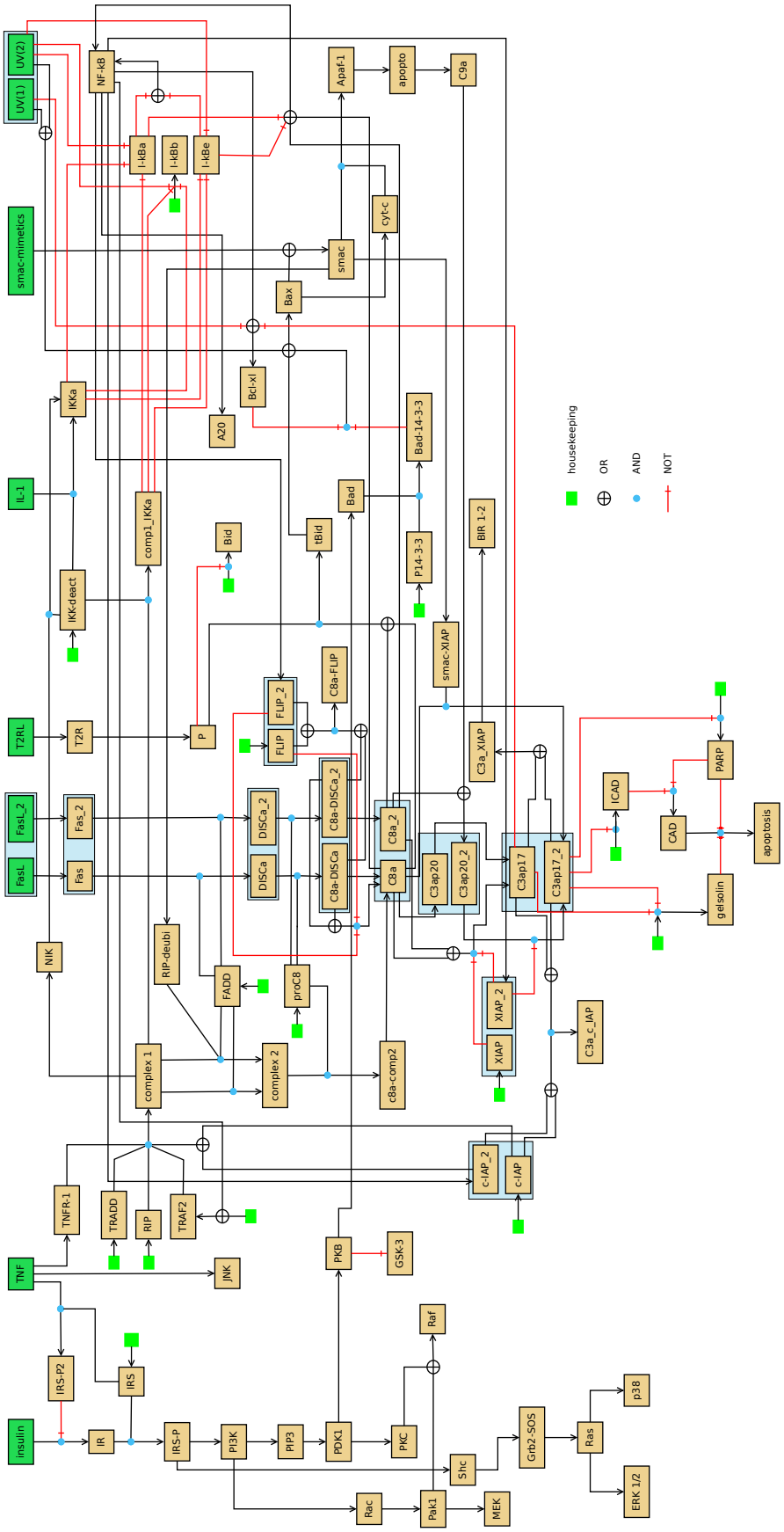


Fig. 8. The wiring of the probabilistic Boolean model of apoptosis in [TMP+14].

the GPU-accelerated parallelisation framework with the SCC-based network reduction technique applied. In this network, there are 36 redundant nodes for all the seven steady-state probabilities. We show in Table 4 the computed steady-state probabilities, the sample size (in millions), the time cost (in seconds), and the speedups we obtain for this computation. The confidence level s, precision r, and the steady-state convergence parameter ϵ of this computation are set to 0.95, 5×10^{-6}, and 10^{-10} respectively. The density of the network is approximately 1.78. The three approaches compute comparable steady-state probabilities with similar trajectory lengths; while our two GPU-accelerated parallelisation frameworks reduce the time cost by approximately 400 and 600 times, respectively. The total time cost for computing the seven probabilities is reduced from about 11 hours to approximately 1.5 min. for the parallel framework without the network reduction technique applied and to less than 1 min. for the parallel framework with the network reduction technique applied.

6. Conclusion and Future Work

Being able to compute the steady-state probabilities of a PBN is an important task for understanding the behaviours of biological systems modelled as PBNs. Estimating the probabilities with simulation-based methods is the only viable way for large PBNs. However, obtaining such probabilities for large PBNs is often time-consuming. In this study, we have proposed a GPU-based parallelisation framework to accelerate the computation speed of estimating the probabilities. We show with experiments that our GPU-based parallelisation gains a speedup of more than two orders of magnitudes. Evaluation on a real-life apoptosis network shows that our GPU-based parallelisation obtains a speedup of approximately 600 times. The significant improvement in the computation speed enables the in-depth steady-state analysis of large PBNs. For example, we can give answers in a reasonable amount of time to questions like "what is the influence of gene A to gene B in the long-run".

In the future, we will apply our work to analyse other large real-life biological models. Moreover, we plan to explore the computation power of GPUs in problems of attractor detection for large Boolean networks. The challenge for this type of problems is how to solve the large state-space exploration issue with the use of GPUs.

Acknowledgements. We would like to thank the anonymous referees who read carefully the previous versions of this paper and gave a lot of valuable comments. Those comments help us to greatly improve the quality of our paper both in content and presentation. Qixia Yuan was supported by the National Research Fund, Luxembourg (grant 7814267). Jun Pang was partially supported by the project SEC-PBN (funded by the University of Luxembourg) and the ANR-FNR project AlgoReCell (INTER/ANR/15/11191283). Andrzej Mizera contributed to this work while holding a postdoctoral researcher position at the Computer Science and Communications Research Unit, University of Luxembourg.

References

[GR92] A. Gelman and D.B. Rubin. Inference from iterative simulation using multiple sequences. *Statistical Science*, 7(4):457–472, 1992.

[HSIO06] Lähdesmäki Harri, Hautaniemi Sampsa, Shmulevich Ilya, and Yli-Harja Olli. Relationships between probabilistic Boolean networks and dynamic Bayesian networks as models of gene regulatory networks. *Signal Processing*, 86(4):814–834, 2006.

[Kau69] Stuart A. Kauffman. Homeostasis and differentiation in random genetic control networks. *Nature*, 224:177–178, 1969.

[MPY15a] A. Mizera, J. Pang, and Q. Yuan. ASSA-PBN: An approximate steady-state analyser of probabilistic Boolean networks. In *Proc. 13th International Symposium on Automated Technology for Verification and Analysis*, volume 9364 of *LNCS*, pages 214–220. Springer, 2015.

[MPY15b] A. Mizera, J. Pang, and Q. Yuan. Reviving the two-state markov chain approach (technical report). Available online at `http://arxiv.org/abs/1501.01779`, 2015.

[MPY16a] A. Mizera, J. Pang, and Q. Yuan. ASSA-PBN 2.0: A software tool for probabilistic Boolean networks. In *Proc. 14th International Conference on Computational Methods in Systems Biology*, volume 9859 of *LNCS*, pages 309–315. Springer, 2016.

[MPY16b] A. Mizera, J. Pang, and Q. Yuan. Fast simulation of probabilistic Boolean networks. In *Proc. 14th International Conference on Computational Methods in Systems Biology*, volume 9859 of *LNCS*, pages 216–231. Springer, 2016.

[MPY16c] A. Mizera, J. Pang, and Q. Yuan. GPU-accelerated steady-state computation of large probabilistic Boolean networks. In *Proc. 2nd International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, volume 9984 of *LNCS*, pages 50–66. Springer, 2016.

[MPY16d] A. Mizera, J. Pang, and Q. Yuan. Parallel approximate steady-state analysis of large probabilistic Boolean networks. In *Proc. 31st ACM Symposium on Applied Computing*, pages 1–8, 2016.

[MPY17] A. Mizera, J. Pang, and Q. Yuan. Reviving the two-state Markov chain approach. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2017. accepted.

[RL92] A. E. Raftery and S. Lewis. How many iterations in the Gibbs sampler? *Bayesian Statistics*, 4:763–773, 1992.

[SD10] Ilya Shmulevich and Edward R. Dougherty. *Probabilistic Boolean Networks: The Modeling and Control of Gene Regulatory Networks*. SIAM Press, 2010.

[SDKZ02] Ilya Shmulevich, Edward R. Dougherty, Seungchan Kim, and Wei Zhang. Probabilistic Boolean networks: a rule-based uncertainty model for gene regulatory networks. *Bioinformatics*, 18(2):261–274, 2002.

[SGH+03] I. Shmulevich, I. Gluhovsky, R.F. Hashimoto, E.R. Dougherty, and W. Zhang. Steady-state analysis of genetic regulatory networks modelled by probabilistic Boolean networks. *Comparative and Functional Genomics*, 4(6):601–608, 2003.

[SSV+09] Rebekka Schlatter, Kathrin Schmich, Ima Avalos Vizcarra, Peter Scheurich, Thomas Sauter, Christoph Borner, Michael Ederer, Irmgard Merfort, and Oliver Sawodny. ON/OFF and beyond - a Boolean model of apoptosis. *PLOS Computational Biology*, 5(12):e1000595, 2009.

[TMP+13] P. Trairatphisan, A. Mizera, J. Pang, A.-A. Tantar, J. Schneider, and T. Sauter. Recent development and biomedical applications of probabilistic Boolean networks. *Cell Communication and Signaling*, 11:46, 2013.

[TMP+14] P. Trairatphisan, A. Mizera, J. Pang, A.-A. Tantar, and T. Sauter. optPBN: An optimisation toolbox for probabilistic boolean networks. *PLOS ONE*, 9(7), 2014.