On Probabilistic Parallel Programs with Process Creation and Synchronisation *

Stefan Kiefer and Dominik Wojtczak

Oxford University Computing Laboratory, UK
{stefan.kiefer,dominik.wojtczak}@comlab.ox.ac.uk

Abstract. We initiate the study of probabilistic parallel programs with dynamic process creation and synchronisation. To this end, we introduce *probabilistic split-join systems (pSJSs)*, a model for parallel programs, generalising both probabilistic pushdown systems (a model for sequential probabilistic procedural programs which is equivalent to recursive Markov chains) and stochastic branching processes (a classical mathematical model with applications in various areas such as biology, physics, and language processing). Our pSJS model allows for a possibly recursive spawning of parallel processes; the spawned processes can synchronise and return values. We study the basic performance measures of pSJSs, especially the distribution and expectation of space, work and time. Our results extend and improve previously known results on the subsumed models. We also show how to do performance analysis in practice, and present two case studies illustrating the modelling power of pSJSs.

1 Introduction

The verification of probabilistic programs with possibly recursive procedures has been intensely studied in the last years. The Markov chains or Markov Decision Processes underlying these systems may have infinitely many states. Despite this fact, which prevents the direct application of the rich theory of finite Markov chains, many positive results have been obtained. Model-checking algorithms have been proposed for both linear and branching temporal logics [11, 15, 23], algorithms deciding properties of several kinds of games have been described (see e.g. [14]), and distributions and expectations of performance measures such as run-time and memory consumption have been investigated [12, 4, 5].

In all these papers programs are modelled as *probabilistic pushdown systems (pPDSs)* or, equivalently [9], as recursive Markov chains. Loosely speaking, a pPDS is a pushdown automaton whose transitions carry probabilities. The *configurations* of a pPDS are pairs containing the current control state and the current stack content. In each *step*, a new configuration is obtained from its predecessor by applying a transition rule, which may modify the control state and the top of the stack.

The programs modelled by pPDSs are necessarily sequential: at each point in time, only the procedure represented by the topmost stack symbol is active. Recursion, however, is a useful language feature also for multithreaded and other parallel programming

* The first author is supported by a postdoctoral fellowship of the German Academic Exchange Service (DAAD). The second author is supported by EPSRC grant EP/G050112/1.

languages, such as Cilk and JCilk, which allow, e.g., for a natural parallelisation of divide-and-conquer algorithms [7, 8]. To model parallel programs in probabilistic scenarios, one may be tempted to use *stochastic multitype branching processes*, a classical mathematical model with applications in numerous fields including biology, physics and natural language processing [17, 2]. In this model, each process has a type, and each type is associated with a probability distribution on transition rules. For instance, a branching process with the transition rules $X \xhookrightarrow{2/3} \{\}$, $X \xhookrightarrow{1/3} \{X, Y\}$, $Y \xhookrightarrow{1} \{X\}$ can be thought of describing a parallel program with two types of processes, X and Y. A process of type X terminates with probability $2/3$, and with probability $1/3$ stays active and spawns a new process of type Y. A process of type Y changes its type to X. A *configuration* of a branching process consists of a pool of currently active processes. In each *step*, all active processes develop in parallel, each one according to a rule which is chosen probabilistically. For instance, a step transforms the configuration $\langle XY \rangle$ into $\langle XYX \rangle$ with probability $\frac{1}{3} \cdot 1$, by applying the second X-rule to the X-process and, in parallel, the Y-rule to the Y-process.

Branching processes do not satisfactorily model parallel programs, because they lack two key features: synchronisation and returning values. In this paper we introduce *probabilistic split-join systems (pSJSs)*, a model which offers these features. Parallel spawns are modelled by rules of the form $X \hookrightarrow \langle YZ \rangle$. The spawned processes Y and Z develop independently; e.g., a rule $Y \hookrightarrow Y'$ may be applied to the Y-process, replacing Y by Y'. When terminating, a process enters a *synchronisation state*, e.g. with rules $Y' \hookrightarrow q$ and $Z \hookrightarrow r$ (where q and r are synchronisation states). Once a process terminates in a synchronisation state, it waits for its *sibling* to terminate in a synchronisation state as well. In the above example, the spawned processes wait for each other, until they terminate in q and r. At that point, they may *join* to form a single process, e.g. with a rule $\langle qr \rangle \hookrightarrow W$. So, synchronisation is achieved by the siblings waiting for each other to terminate. All rules could be probabilistic. Notice that synchronisation states can be used to return values; e.g., if the Y-process returns q' instead of q, this can be recorded by the existence of a rule $\langle q'r \rangle \hookrightarrow W'$, so that the resulting process (i.e., W or W') depends on the values computed by the joined processes. For the notion of siblings to make sense, a *configuration* of a pSJS is not a set, but a binary tree whose leaves are *process symbols* (such as X, Y, Z) or synchronisation states (such as q, r). A *step* transforms the leaves of the binary tree in parallel by applying rules; if a leaf is not a process symbol but a synchronisation state, it remains unchanged unless its sibling is also a synchronisation state and a joining rule (such as $\langle qr \rangle \hookrightarrow W$) exists, which removes the siblings and replaces their parent node with the right hand side.

Related work. The probabilistic models closest to ours are pPDSs, recursive Markov chains, and stochastic branching processes, as described above. The non-probabilistic (i.e., nondeterministic) version of pSJSs (SJSs, say) can be regarded as a special case of *ground tree rewriting systems*, see [19] and the references therein. A configuration of a ground tree rewriting system is a node-labelled tree, and a rewrite rule replaces a subtree. The *process rewrite system* (PRS) hierarchy of [21] features sequential and parallel process composition. Due to its syntactic differences, it is not obvious whether SJSs are in that hierarchy. They would be above pushdown systems (which is the sequential fragment of PRSs), because SJSs subsume pushdown systems, as we show in

Section 3.1 for the probabilistic models. *Dynamic pushdown networks* (DPNs) [3] are a parallel extension of pushdown systems. A configuration of a DPN is a list of configurations of pushdown systems running in parallel. DPNs feature the spawning of parallel threads, and an extension of DPNs, called *constrained DPNs*, can also model joins via regular expressions on spawned children. The DPN model is more powerful and more complicated than SJSs. All those models are non-probabilistic.

Organisation of the paper. In Section 2 we formally define our model and provide further preliminaries. Section 3 contains our main results: we study the relationship between pSJSs and pPDSs (Section 3.1), we show how to compute the probabilities for termination and finite space, respectively (Sections 3.2 and 3.3), and investigate the distribution and expectation of work and time (Section 3.4). In Section 4 we present two case studies illustrating the modelling power of pSJSs. We conclude in Section 5. All proofs are provided in a technical report [18].

2 Preliminaries

For a finite or infinite word w, we write $w(0), w(1), \ldots$ to refer to its individual letters. We assume throughout the paper that $\mathcal{B}$ is a fixed infinite set of *basic process symbols*. We use the symbols '$\langle$' and '$\rangle$' as special letters not contained in $\mathcal{B}$. For an alphabet Σ, we write $\langle\Sigma\Sigma\rangle$ to denote the language $\{\langle\sigma_1\sigma_2\rangle \mid \sigma_1, \sigma_2 \in \Sigma\}$ and $\Sigma^{1,2}$ to denote $\Sigma \cup \langle\Sigma\Sigma\rangle$. To a set Σ we associate a set $T(\Sigma)$ of binary trees whose leaves are labelled with elements of Σ. Formally, $T(\Sigma)$ is the smallest language that contains Σ and $\langle T(\Sigma)T(\Sigma)\rangle$. For instance, $\langle\langle\sigma\sigma\rangle\sigma\rangle \in T(\{\sigma\})$.

Definition 1 (pSJS). *Let Q be a finite set of* synchronisation states *disjoint from $\mathcal{B}$ and not containing '$\langle$' or '$\rangle$'. Let Γ be a finite set of* process symbols*, such that $\Gamma \subset \mathcal{B} \cup \langle QQ\rangle$. Define the* alphabet *$\Sigma := \Gamma \cup Q$. Let $\delta \subseteq \Gamma \times \Sigma^{1,2}$ be a transition relation. Let $Prob : \delta \to (0, 1]$ be a function so that for all $a \in \Gamma$ we have $\sum_{a \hookrightarrow \alpha \in \delta} Prob(a \hookrightarrow \alpha) = 1$. Then the tuple $S = (\Gamma, Q, \delta, Prob)$ is a* probabilistic split-join system *(pSJS). A pSJS with $\Gamma \cap \langle QQ\rangle = \emptyset$ is called* branching process*.*

We usually write $a \overset{p}{\hookrightarrow} \alpha$ instead of $Prob(a \hookrightarrow \alpha) = p$. For technical reasons we allow branching processes of "degree 3", i.e., branching processes where $\Sigma^{1,2}$ may be extended to $\Sigma^{1,2,3} := \Sigma^{1,2} \cup \{\langle\sigma_1\sigma_2\sigma_3\rangle \mid \sigma_1, \sigma_2, \sigma_3 \in \Sigma\}$. In branching processes, it is usually sufficient to have $|Q| = 1$.

A *Markov chain* is a stochastic process that can be described by a triple $M = (D, \to, Prob)$ where D is a finite or countably infinite set of *states*, $\to \subseteq D \times D$ is a *transition relation*, and $Prob$ is a function which to each transition $s \to t$ of M assigns its probability $Prob(s \to t) > 0$ so that for every $s \in D$ we have $\sum_{s \to t} Prob(s \to t) = 1$ (as usual, we write $s \overset{x}{\to} t$ instead of $Prob(s \to t) = x$). A *path* (or *run*) in M is a finite (or infinite, resp.) word $u \in D^+ \cup D^\omega$, such that $u(i-1) \to u(i)$ for every $1 \leq i < |u|$. The set of all runs that start with a given path u is denoted by $Run[M](u)$ (or $Run(u)$, if M is understood). To every $s \in D$ we associate the probability space $(Run(s), \mathcal{F}, \mathcal{P})$ where $\mathcal{F}$ is the σ-field generated by all *basic cylinders* $Run(u)$ where u is a path starting with s, and $\mathcal{P} : \mathcal{F} \to [0, 1]$ is the

unique probability measure such that $\mathcal{P}(Run(u)) = \Pi_{i=1}^{|u|-1} x_i$ where $u(i-1) \xrightarrow{x_i} u(i)$ for every $1 \leq i < |u|$. Only certain subsets of $Run(s)$ are $\mathcal{P}$-measurable, but in this paper we only deal with "safe" subsets that are guaranteed to be in $\mathcal{F}$. If $\mathbf{X}_s$ is a random variable over $Run(s)$, we write $\mathbb{E}[\mathbf{X}_s]$ for its expectation. For $s, t \in D$, we define $Run(s{\downarrow}t) := \{w \in Run(s) \mid \exists i \geq 0 : w(i) = t\}$ and $[s{\downarrow}t] := \mathcal{P}(Run(s{\downarrow}t))$.

To a pSJS $S = (\Gamma, Q, \delta, Prob)$ with alphabet $\Sigma = \Gamma \cup Q$ we associate a Markov chain M_S with $T(\Sigma)$ as set of states. For $t \in T(\Sigma)$, we define $Front(t) = a_1, \ldots, a_k$ as the unique finite sequence of subwords of t (read from left to right) with $a_i \in \Gamma$ for all $1 \leq i \leq k$. We write $|Front(t)| = k$. If $k = 0$, then t is called *terminal*. The Markov chain M_S has a transition $t \xrightarrow{p} t'$, if: $Front(t) = a_1, \ldots, a_k$; $a_i \xhookrightarrow{p_i} \alpha_i$ are transitions in S for all i; t' is obtained from t by replacing a_i with α_i for all i; and $p = \prod_{i=1}^{k} p_i$. Note that $t \xrightarrow{1} t$, if t is terminal. For branching processes of degree 3, the set $T(\Sigma)$ is extended in the obvious way to trees whose nodes may have two or three children.

Denote by $\mathbf{T}_\sigma$ a random variable over $Run(\sigma)$ where $\mathbf{T}_\sigma(w)$ is either the least $i \in \mathbb{N}$ such that $w(i)$ is terminal, or ∞, if no such i exists. Intuitively, $\mathbf{T}_\sigma(w)$ is the number of steps in which w terminates, i.e., the *termination time*. Denote by $\mathbf{W}_\sigma$ a random variable over $Run(\sigma)$ where $\mathbf{W}_\sigma(w) := \sum_{i=0}^{\infty} |Front(w(i))|$. Intuitively, $\mathbf{W}_\sigma(w)$ is the total *work* in w. Denote by $\mathbf{S}_\sigma$ a random variable over $Run(\sigma)$ where $\mathbf{S}_\sigma(w) := \sup_{i=0}^{\infty} |w(i)|$, and $|w(i)|$ is the length of $w(i)$ not counting the symbols '$\langle$' and '$\rangle$'. Intuitively, $\mathbf{S}_\sigma(w)$ is the maximal number of processes during the computation, or, short, the *space* of w.

Example 2. Consider the pSJS with $\Gamma = \{X, \langle qr\rangle\}$ and $Q = \{q, r\}$ and the transitions $X \xhookrightarrow{0.5} \langle XX\rangle$, $X \xhookrightarrow{0.3} q$, $X \xhookrightarrow{0.2} r$, $\langle qr\rangle \xhookrightarrow{1} X$. Let $u = X\ \langle XX\rangle\ \langle qr\rangle\ X\ q\ q$. Then u is a path, because we have $X \xrightarrow{0.5} \langle XX\rangle \xrightarrow{0.06} \langle qr\rangle \xrightarrow{1} X \xrightarrow{0.3} q \xrightarrow{1} q$. Note that q is terminal. The set $Run(u)$ contains only one run, namely $w := u(0)u(1)u(2)u(3)u(4)u(4)\cdots$. We have $\mathcal{P}(Run(u)) = 0.5 \cdot 0.06 \cdot 0.3$, and $\mathbf{T}_X(w) = 4$, $\mathbf{W}_X(w) = 5$, and $\mathbf{S}_X(w) = 2$. The dags in Figure 1 graphically represent this run (on the left), and another example run (on the right) with $\mathbf{T}_X = 3$, $\mathbf{W}_X = 5$, and $\mathbf{S}_X = 3$.

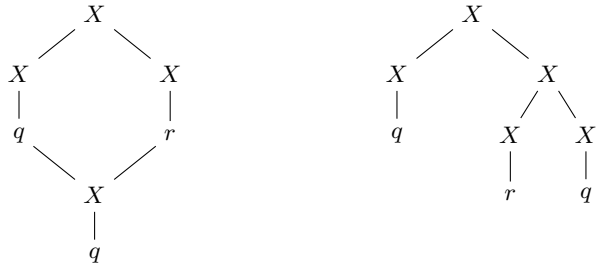

Fig. 1. Two terminating runs

Remark 3. Our definition of pSJSs may be more general than needed from a modelling perspective: e.g., our rules allow for both synchronisation and splitting in a single step. We choose this definition for technical convenience and to allow for easy comparisons with pPDSs (Section 3.1).

The complexity-theoretic statements in this paper are with respect to the *size* of the given pSJS $S = (\Gamma, Q, \delta, Prob)$, which is defined as $|\Gamma| + |Q| + |\delta| + |Prob|$, where $|Prob|$ equals the sum of the sizes of the binary representations of the values of $Prob$. A formula of $ExTh(\mathbb{R})$, the existential fragment of the first-order theory of the reals, is of the form $\exists x_1 \ldots \exists x_m R(x_1, \ldots, x_n)$, where $R(x_1, \ldots, x_n)$ is a boolean combination of comparisons of the form $p(x_1, \ldots, x_n) \sim 0$, where $p(x_1, \ldots, x_n)$ is a multivariate polynomial and $\sim \in \{<, >, \leq, \geq, =, \neq\}$. The validity of closed formulas ($m = n$) is decidable in PSPACE [6, 22]. We say that one can *efficiently express* a value $c \in \mathbb{R}$ associated with a pSJS, if one can, in polynomial space, construct a formula $\phi(x)$ in $ExTh(\mathbb{R})$ of polynomial length such that x is the only free variable in $\phi(x)$, and $\phi(x)$ is true if and only if $x = c$. Notice that if c is efficiently expressible, then $c \sim \tau$ for $\tau \in \mathbb{Q}$ is decidable in PSPACE for $\sim \in \{<, >, \leq, \geq, =, \neq\}$.

For some lower bounds, we prove hardness (with respect to P-time many-one reductions) in terms of the PosSLP decision problem. The PosSLP (Positive Straight-Line Program) problem asks whether a given straight-line program or, equivalently, arithmetic circuit with operations $+$, $-$, $\cdot$, and inputs 0 and 1, and a designated output gate, outputs a positive integer or not. PosSLP is in PSPACE. More precisely, it is known to be on the 4th level of the Counting Hierarchy [1]; it is not known to be in NP. The PosSLP problem is a fundamental problem for numerical computation; it is complete for the class of decision problems that can be solved in polynomial time on models with unit-cost exact rational arithmetic, see [1, 15] for more details.

3 Results

3.1 Relationship with probabilistic pushdown systems (pPDSs)

We show that pSJSs subsume pPDSs. A *probabilistic pushdown system (pPDS)* [11, 12, 4, 5] is a tuple $S = (\Gamma, Q, \delta, Prob)$, where Γ is a finite *stack alphabet*, Q is a finite set of *control states*, $\delta \subseteq Q \times \Gamma \times Q \times \Gamma^{\leq 2}$ (where $\Gamma^{\leq 2} = \{\alpha \in \Gamma^*, |\alpha| \leq 2\}$) is a *transition relation*, and $Prob : \delta \to (0, 1]$ is a function so that for all $q \in Q$ and $a \in \Gamma$ we have $\sum_{qa \hookrightarrow r\alpha} Prob(qa \hookrightarrow r\alpha) = 1$. One usually writes $qa \xrightarrow{p} r\alpha$ instead of $Prob(qa \to r\alpha) = p$. To a pPDS $S = (\Gamma, Q, \delta, Prob)$ one associates a Markov chain M_S with $Q \times \Gamma^*$ as set of states, and transitions $q \xrightarrow{1} q$ for all $q \in Q$, and $qa\beta \xrightarrow{p} r\alpha\beta$ for all $qa \overset{p}{\hookrightarrow} r\alpha$ and all $\beta \in \Gamma^*$.

A pPDS S_P with Γ_P as stack alphabet, Q_P as set of control states, and transitions $\overset{p}{\hookrightarrow}_P$ can be transformed to an equivalent pSJS S: Take $Q := Q_P \cup \Gamma_P$ as synchronisation states; $\Gamma := \{\langle qa \rangle \mid q \in Q_P,\ a \in \Gamma_P\}$ as process symbols; and transitions $\langle qa \rangle \overset{p}{\hookrightarrow} \langle \langle rb \rangle c \rangle$ for all $qa \overset{p}{\hookrightarrow}_P rbc$, $\langle qa \rangle \overset{p}{\hookrightarrow} \langle rb \rangle$ for all $qa \overset{p}{\hookrightarrow}_P rb$, and $\langle qa \rangle \overset{p}{\hookrightarrow} r$ for all $qa \overset{p}{\hookrightarrow}_P r$. The Markov chains M_{S_P} and M_S are isomorphic. Therefore, we occasionally say that a pSJS *is* a pPDS, if it can be obtained from a pPDS by this transformation. Observe that in pPDSs, we have $\mathbf{T} = \mathbf{W}$, because there is no parallelism.

Conversely, a pSJS S with alphabet $\Sigma = \Gamma \cup Q$ can be transformed into a pPDS S_P by "serialising" S: Take $Q_P := \{\Box\} \cup \{\overline{q} \mid q \in Q\}$ as control states; $\Gamma_P := \Gamma \cup Q \cup \{\widetilde{q} \mid q \in Q\}$ as stack alphabet; and transitions $\Box a \overset{p}{\hookrightarrow}_P \Box\sigma_1\sigma_2$ for all $a \overset{p}{\hookrightarrow} \langle \sigma_1\sigma_2 \rangle$,

$\Box a \overset{p}{\hookrightarrow}_P \Box\sigma$ for all $a \overset{p}{\hookrightarrow} \sigma$ with $\sigma \in \Sigma \setminus \langle QQ\rangle$, and $\Box q \overset{1}{\hookrightarrow}_P \overline{q}$ for all $q \in Q$, and $\overline{q}\sigma \overset{1}{\hookrightarrow}_P \Box\sigma\widetilde{q}$ for all $q \in Q$ and $\sigma \in \Sigma$, and $\overline{r}\widetilde{q} \overset{1}{\hookrightarrow}_P \Box\langle qr\rangle$ for all $q, r \in Q$. The Markov chains M_S and M_{S_P} are *not* isomorphic. However, we have:

Proposition 4. *There is a probability-preserving bijection between the runs $Run(\sigma{\downarrow}q)$ in M_S and the runs $Run(\Box\sigma{\downarrow}\overline{q})$ in M_{S_P}. In particular, we have $[\sigma{\downarrow}q] = [\Box\sigma{\downarrow}\overline{q}]$.*

For example, the pSJS run on the left side of Figure 1 corresponds to the pPDS run $\Box X \xrightarrow{0.5} \Box XX \xrightarrow{0.3} \Box qX \xrightarrow{1} \overline{q}X \xrightarrow{1} \Box X\widetilde{q} \xrightarrow{0.2} \Box r\widetilde{q} \xrightarrow{1} \overline{r}\widetilde{q} \xrightarrow{1} \Box\langle qr\rangle \xrightarrow{1} \Box X \xrightarrow{0.3} \Box q \xrightarrow{1} \overline{q} \xrightarrow{1} \overline{q} \xrightarrow{1} \dots$

3.2 Probability of Termination

We call a run *terminating*, if it reaches a terminal tree. Such a tree can be a single synchronisation state (e.g., q on the left of Figure 1), or another terminal tree (e.g., $\langle q\langle rq\rangle\rangle$ on the right of Figure 1). For any $\sigma \in \Sigma$, we denote by $[\sigma{\downarrow}]$ the termination probability when starting in σ; i.e., $[\sigma{\downarrow}] = \sum_{t \text{ is terminal}} [\sigma{\downarrow}t]$. One can transform any pSJS S into a pSJS S' such that whenever a run in S terminates, then a corresponding run in S' terminates in a synchronisation state. This transformation is by adding a fresh state $\check{q}$, and transitions $\langle rs\rangle \overset{1}{\hookrightarrow} \check{q}$ for all $r, s \in Q$ with $\langle rs\rangle \notin \Gamma$, and $\langle \check{q}r\rangle \overset{1}{\hookrightarrow} \check{q}$ and $\langle r\check{q}\rangle \overset{1}{\hookrightarrow} \check{q}$ for all $r \in Q$. It is easy to see that this keeps the probability of termination unchanged, and modifies the random variables $\mathbf{T}_\sigma$ and $\mathbf{W}_\sigma$ by at most a factor 2. Notice that the transformation can be performed in polynomial time. After the transformation we have $[\sigma{\downarrow}] = \sum_{q\in Q}[\sigma{\downarrow}q]$. A pSJS which satisfies this equality will be called *normalised* in the following. From a modelling point of view, pSJSs may be expected to be normalised in the first place: a terminating program should terminate all its processes.

We set up an equation system for the probabilities $[\sigma{\downarrow}q]$. For each $\sigma \in \Sigma$ and $q \in Q$, the equation system has a variable of the form $[\![\sigma{\downarrow}q]\!]$ and an equation of the form $[\![\sigma{\downarrow}q]\!] = f_{[\![\sigma\downarrow q]\!]}$, where $f_{[\![\sigma\downarrow q]\!]}$ is a multivariate polynomial with nonnegative coefficients. More concretely: If $q \in Q$, then we set $[\![q{\downarrow}q]\!] = 1$; if $r \in Q \setminus \{q\}$, then we set $[\![r{\downarrow}q]\!] = 0$; if $a \in \Gamma$, then we set

$$[\![a{\downarrow}q]\!] = \sum_{\substack{a \overset{p}{\hookrightarrow}\langle\sigma_1\sigma_2\rangle \\ \langle q_1q_2\rangle \in \Gamma\cap\langle QQ\rangle}} p \cdot [\![\sigma_1{\downarrow}q_1]\!] \cdot [\![\sigma_2{\downarrow}q_2]\!] \cdot [\![\langle q_1q_2\rangle{\downarrow}q]\!] \;+\; \sum_{\substack{a \overset{p}{\hookrightarrow}\sigma' \\ \sigma' \in \Sigma\setminus\langle QQ\rangle}} p \cdot [\![\sigma'{\downarrow}q]\!] \,.$$

Proposition 5. *Let $\sigma \in \Sigma$ and $q \in Q$. Then $[\sigma{\downarrow}q]$ is the value for $[\![\sigma{\downarrow}q]\!]$ in the least (w.r.t. componentwise ordering) nonnegative solution of the above equation system.*

One can efficiently approximate $[\sigma{\downarrow}q]$ by applying Newton's method to the fixed-point equation system from Proposition 5, cf. [15]. The convergence speed of Newton's method for such equation systems was recently studied in detail [10]. The simpler "Kleene" method (sometimes called "fixed-point iteration") often suffices, but can be much slower. In the case studies of Section 4, using Kleene for computing the termination probabilities up to machine accuracy was not a bottleneck. The following theorem essentially follows from similar results for pPDSs:

Theorem 6 (cf. [13, 15]). *Consider a pSJS with alphabet $\Sigma = \Gamma \cup Q$. Let $\sigma \in \Sigma$ and $q \in Q$. Then (1) one can efficiently express (in the sense defined in Section 2) the value of $[\sigma{\downarrow}q]$, (2) deciding whether $[\sigma{\downarrow}q] = 0$ is in* P*, and (3) deciding whether $[\sigma{\downarrow}q] < 1$ is* PosSLP*-hard even for pPDSs.*

3.3 Probability of Finite Space

A run $w \in Run(\sigma)$ is either (i) terminating, or (ii) nonterminating with $\mathbf{S}_\sigma < \infty$, or (iii) nonterminating with $\mathbf{S}_\sigma = \infty$. From a modelling point of view, some programs may be considered incorrect, if they do not terminate with probability 1. As is well-known, this does not apply to programs like operating systems, network servers, system daemons, etc., where nontermination may be tolerated or desirable. Such programs may be expected not to need an infinite amount of space; i.e., $\mathbf{S}_\sigma$ should be finite.

Given a pSJS S with alphabet $\Sigma = \Gamma \cup Q$, we show how to construct, in polynomial time, a normalised pSJS $\overline{S}$ with alphabet $\overline{\Sigma} = \overline{\Gamma} \cup \overline{Q} \supseteq \Sigma$ where $\overline{Q} = Q \cup \{\overline{q}\}$ for a fresh synchronisation state $\overline{q}$, and $\mathcal{P}\left(\mathbf{S}_a < \infty = \mathbf{T}_a \mid Run(a)\right) = [a{\downarrow}\overline{q}]$ for all $a \in \Gamma$. Having done that, we can compute this probability according to Section 3.2.

For the construction, we can assume w.l.o.g. that S has been normalised using the procedure of Section 3.2. Let $U := \{a \in \Gamma \mid \forall n \in \mathbb{N} : \mathcal{P}\left(\mathbf{S}_a > n\right) > 0\}$.

Lemma 7. *The set U can be computed in polynomial time.*

Let $B := \{a \in \Gamma \setminus U \mid \forall q \in Q : [a{\downarrow}q] = 0\}$, so B is the set of process symbols a that are both "bounded above" (because $a \notin U$) and "bounded below" (because a cannot terminate). By Theorem 6 (2) and Lemma 7 we can compute B in polynomial time. Now we construct $\overline{S}$ by modifying S as follows: we set $\overline{Q} := Q \cup \{\overline{q}\}$ for a fresh synchronisation state $\overline{q}$; we remove all transitions with symbols $b \in B$ on the left hand side and replace them with a new transition $b \overset{1}{\hookrightarrow} \overline{q}$; we add transitions $\langle q_1 q_2 \rangle \overset{1}{\hookrightarrow} \overline{q}$ for all $q_1, q_2 \in \overline{Q}$ with $\overline{q} \in \{q_1, q_2\}$. We have the following proposition.

Proposition 8. *(1) The pSJS $\overline{S}$ is normalised; (2) the value $[a{\downarrow}q]$ for $a \in \Gamma$ and $q \in Q$ is the same in S and $\overline{S}$; (3) we have $\mathcal{P}\left(\mathbf{S}_a < \infty = \mathbf{T}_a \mid Run(a)\right) = [a{\downarrow}\overline{q}]$ for all $a \in \Gamma$.*

Proposition 8 allows for the following theorem.

Theorem 9. *Consider a pSJS with alphabet $\Sigma = \Gamma \cup Q$ and $a \in \Gamma$. Let $s := \mathcal{P}\left(\mathbf{S}_a < \infty\right)$. Then (1) one can efficiently express s, (2) deciding whether $s = 0$ is in* P*, and (3) deciding whether $s < 1$ is* PosSLP*-hard even for pPDSs.*

Theorem 9, applied to pPDSs, improves Corollary 6.3 of [12]. There it is shown for pPDSs that comparing $\mathcal{P}\left(\mathbf{S}_a < \infty\right)$ with $\tau \in \mathbb{Q}$ is in EXPTIME, and in PSPACE if $\tau \in \{0, 1\}$. With Theorem 9 we get PSPACE for $\tau \in \mathbb{Q}$, and P for $\tau = 0$.

3.4 Work and Time

We show how to compute the distribution and expectation of work and time of a given pSJS S with alphabet $\Sigma = \Gamma \cup Q$.

Distribution. For $\sigma \in \Sigma$ and $q \in Q$, let $T_{\sigma\downarrow q}(k) := \mathcal{P}\left(Run(\sigma\downarrow q),\ \mathbf{T}_\sigma = k \mid Run(\sigma)\right)$. It is easy to see that, for $k \geq 1$ and $a \in \Gamma$ and $q \in Q$, we have

$$T_{a\downarrow q}(k) = \sum_{\substack{a \xhookrightarrow{p} \langle \sigma_1 \sigma_2 \rangle \\ \langle q_1 q_2 \rangle \in \Gamma \cap \langle QQ \rangle}} p \cdot \sum_{\substack{\ell_1,\ell_2,\ell_3 \geq 0 \\ \max\{\ell_1,\ell_2\}+\ell_3 = k-1}} T_{\sigma_1\downarrow q_1}(\ell_1) \cdot T_{\sigma_2\downarrow q_2}(\ell_2) \cdot T_{\langle q_1 q_2\rangle\downarrow q}(\ell_3) + \sum_{\substack{a \xhookrightarrow{p} \sigma' \\ \sigma' \in \Sigma \setminus \langle QQ \rangle}} p \cdot T_{\sigma'\downarrow q}(k-1)\,.$$

This allows to compute the distribution of time (and, similarly, work) using dynamic programming. In particular, for any k, one can compute $\overrightarrow{T}_{\sigma\downarrow q}(k) := \mathcal{P}\left(\mathbf{T}_\sigma > k \mid Run(\sigma\downarrow q)\right) = 1 - \frac{1}{[\sigma\downarrow q]} \sum_{i=0}^{k} T_{\sigma\downarrow q}(k)$.

Expectation. For any random variable Z taking positive integers as value, it holds $\mathbb{E}Z = \sum_{k=0}^{\infty} \mathcal{P}\left(Z > k\right)$. Hence, one can approximate $\mathbb{E}\left[\mathbf{T}_\sigma \mid Run(\sigma\downarrow q)\right] = \sum_{k=0}^{\infty} \overrightarrow{T}_{\sigma\downarrow q}(k)$ by computing $\sum_{k=0}^{\ell} \overrightarrow{T}_{\sigma\downarrow q}(k)$ for large ℓ. In the rest of the section we show how to decide on the finiteness of expected work and time. It follows from Propositions 10 and 11 below that the expected work $\mathbb{E}\left[\mathbf{W}_\sigma \mid Run(\sigma\downarrow q)\right]$ is easier to compute: it is the solution of a linear equation system.

We construct a branching process $\overline{S}$ with process symbols $\overline{\Gamma} = \{(\!|aq|\!) \mid a \in \Gamma, q \in Q, [a\downarrow q] > 0\}$, synchronisation states $\overline{Q} = \{\bot\}$ and transitions as follows. For notational convenience, we identify $\bot$ and $(\!|qq|\!)$ for all $q \in Q$. For $(\!|aq|\!) \in \overline{\Gamma}$, we set

- $(\!|aq|\!) \xrightarrow{y/[a\downarrow q]} \langle (\!|\sigma_1 q_1|\!)(\!|\sigma_2 q_2|\!)(\!|\langle q_1 q_2\rangle q|\!)\rangle$ for all $a \xhookrightarrow{p} \langle \sigma_1\sigma_2\rangle$ and $\langle q_1 q_2\rangle \in \Gamma \cap \langle QQ\rangle$, where $y := p \cdot [\sigma_1\downarrow q_1] \cdot [\sigma_2\downarrow q_2] \cdot [\langle q_1 q_2\rangle\downarrow q] > 0$;
- $(\!|aq|\!) \xrightarrow{y/[a\downarrow q]} (\!|\sigma' q|\!)$ for all $a \xhookrightarrow{p} \sigma'$ with $\sigma' \in \Sigma \setminus \langle QQ\rangle$, where $y := p \cdot [\sigma'\downarrow q] > 0$.

The following proposition (inspired by a statement on pPDSs [5]) links the distributions of $\mathbf{W}_\sigma$ and $\mathbf{T}_\sigma$ conditioned under termination in q with the distributions of $\mathbf{W}_{(\!|\sigma q|\!)}$ and $\mathbf{T}_{(\!|\sigma q|\!)}$.

Proposition 10. *Let $\sigma \in \Sigma$ and $q \in Q$ with $[\sigma\downarrow q] > 0$. Then*

$$\mathcal{P}\left(\mathbf{W}_\sigma = n \mid Run(\sigma\downarrow q)\right) = \mathcal{P}\left(\mathbf{W}_{(\!|\sigma q|\!)} = n \mid Run((\!|\sigma q|\!))\right) \quad \textit{for all } n \geq 0 \quad \textit{and}$$
$$\mathcal{P}\left(\mathbf{T}_\sigma \leq n \mid Run(\sigma\downarrow q)\right) \leq \mathcal{P}\left(\mathbf{T}_{(\!|\sigma q|\!)} \leq n \mid Run((\!|\sigma q|\!))\right) \quad \textit{for all } n \geq 0.$$

In particular, we have $[(\!|\sigma q|\!)\downarrow] = 1$.

Proposition 10 allows us to focus on branching processes. For $X \in \Gamma$ and a finite sequence $\sigma_1, \ldots, \sigma_k$ with $\sigma_i \in \Sigma$, define $|\sigma_1, \ldots, \sigma_k|_X := |\{i \mid 1 \leq i \leq k,\ \sigma_i = X\}|$, i.e., the number of X-symbols in the sequence. We define the *characteristic matrix* $A \in \mathbb{R}^{\Gamma\times\Gamma}$ of a branching process by setting

$$A_{X,Y} := \sum_{X \xhookrightarrow{p} \langle \sigma_1\sigma_2\sigma_3\rangle} p \cdot |\sigma_1,\sigma_2,\sigma_3|_Y + \sum_{X \xhookrightarrow{p} \langle \sigma_1\sigma_2\rangle} p \cdot |\sigma_1,\sigma_2|_Y + \sum_{X \xhookrightarrow{p} \sigma_1} p \cdot |\sigma_1|_Y\,.$$

It is easy to see that the (X, Y)-entry of A is the expected number of Y-processes after the first step, if starting in a single X-process. If S is a branching process and $X_0 \in \Gamma$, we call the pair (S, X_0) a *reduced branching process*, if for all $X \in \Gamma$ there is $i \in \mathbb{N}$ such that $(A^i)_{X_0,X} > 0$. Intuitively, (S, X_0) is reduced, if, starting in X_0, all process symbols can be reached with positive probability. If (S, X_0) is not reduced, it is easy to reduce it in polynomial time by eliminating all non-reachable process symbols.

The following proposition characterises the finiteness of both expected work and expected time in terms of the spectral radius $\rho(A)$ of A. (Recall that $\rho(A)$ is the largest absolute value of the eigenvalues of A.)

Proposition 11. *Let (S, X_0) be a reduced branching process. Let A be the associated characteristic matrix. Then the following statements are equivalent:*

(1) $\mathbb{E}\mathbf{W}_{X_0}$ is finite; *(2) $\mathbb{E}\mathbf{T}_{X_0}$ is finite;* *(3) $\rho(A) < 1$.*

Further, if $\mathbb{E}\mathbf{W}_{X_0}$ is finite, then it equals the X_0-component of $(I - A)^{-1} \cdot \mathbf{1}$, where I is the identity matrix, and $\mathbf{1}$ is the column vector with all ones.

Statements similar to Proposition 11 do appear in the standard branching process literature [17, 2], however, not explicitly enough to cite directly or with stronger assumptions[1]. Our proof adapts a technique which was developed in [4] for a different purpose. It uses only basic tools and Perron-Frobenius theory, the spectral theory of nonnegative matrices. Proposition 11 has the following consequence:

Corollary 12. *Consider a branching process with process symbols Γ and $X_0 \in \Gamma$. Then $\mathbb{E}\mathbf{W}_{X_0}$ and $\mathbb{E}\mathbf{T}_{X_0}$ are both finite or both infinite. Distinguishing between those cases is in* P.

By combining the previous results we obtain the following theorem.

Theorem 13. *Consider a pSJS S with alphabet $\Sigma = \Gamma \cup Q$. Let $a \in \Gamma$. Then $\mathbb{E}\mathbf{W}_a$ and $\mathbb{E}\mathbf{T}_a$ are both finite or both infinite. Distinguishing between those cases is in* PSPACE, *and* PosSLP*-hard even for pPDSs. Further, if S is normalised and $\mathbb{E}\mathbf{W}_a$ is finite, one can efficiently express $\mathbb{E}\mathbf{W}_a$.*

Theorem 13 can be interpreted as saying that, although the pSJS model does not impose a bound on the number of active processes at a time, its parallelism *cannot* be used to do an infinite expected amount of work in a finite expected time. However, the "speedup" $\mathbb{E}[\mathbf{W}] / \mathbb{E}[\mathbf{T}]$ may be unbounded:

Proposition 14. *Consider the family of branching processes with transitions $X \xhookrightarrow{p} \langle XX \rangle$ and $X \xhookrightarrow{1-p} \bot$, where $0 < p < 1/2$. Then the ratio $\mathbb{E}[\mathbf{W}_X] / \mathbb{E}[\mathbf{T}_X]$ is unbounded for $p \to 1/2$.*

[1] For example, [2] assumes that there is $n \in \mathbb{N}$ such that A^n is positive in all entries, a restriction which is not natural for our setting.

4 Case Studies

We have implemented a prototype tool in the form of a Maple worksheet, which allows to compute some of the quantities from the previous section: the termination probabilities, and distributions and expectations of work and time. In this section, we use our tool for two case studies[2], which also illustrate how probabilistic parallel programs can be modelled with pSJSs. We only deal with normalised pSJSs in this section.

4.1 Divide and Conquer

The pSJS model lends itself to analyse parallel divide-and-conquer programs. For simplicity, we assume that the problem is already given as a binary tree, and solving it means traversing the tree and combining the results of the children. Figure 2 shows generic parallel code for such a problem.

```
function divCon(node)
  if node.leaf() then return node.val()
  else parallel ⟨ val1 := divCon(node.c1), val2 := divCon(node.c2) ⟩
      return combine(val1, val2)
```

Fig. 2. A generic parallel divide-and-conquer program.

For an example, think of a routine for numerically approximating an integral $\int_0^1 f(x)\,dx$. Given the integrand f and a subinterval $I \subseteq [0,1]$, we assume that there is a function which computes $osc_f(I) \in \mathbb{N}$, the "oscillation" of f in the interval I, a measure for the need for further refinement. If $osc_f(I) = 0$, then the integration routine returns the approximation $1 \cdot f(1/2)$, otherwise it returns $I_1 + I_2$, where I_1 and I_2 are recursive approximations of $\int_0^{1/2} f(x)\,dx$ and $\int_{1/2}^1 f(x)\,dx$, respectively.[3]

We analyse such a routine using probabilistic assumptions on the integrand: Let n, n_1, n_2 be nonnegative integers such that $0 \leq n_1{+}n_2 \leq n$. If $osc_f([a,b]) = n$, then $osc_f([a,(a{+}b)/2]) = n_1$ and $osc_f([(a{+}b)/2,b]) = n_2$ with probability $x(n,n_1,n_2) := \binom{n}{n_1} \cdot \binom{n-n_1}{n_2} \cdot \left(\frac{p}{2}\right)^{n_1} \cdot \left(\frac{p}{2}\right)^{n_2} \cdot (1{-}p)^{n-n_1-n_2}$, where $0 < p < 1$ is some parameter.[4] Of course, other distributions could be used as well. The integration routine can then be modelled by the pSJS with $Q = \{q\}$ and $\Gamma = \{\langle qq\rangle, 0, \ldots, n_{\max}\}$ and the following rules:

$$0 \overset{1}{\hookrightarrow} q \quad \text{and} \quad \langle q\,q\rangle \overset{1}{\hookrightarrow} q \quad \text{and} \quad n \xrightarrow{x(n,n_1,n_2)} \langle n_1\,n_2\rangle \text{ for all } 1 \leq n \leq n_{\max}\,,$$

where $0 \leq n_1{+}n_2 \leq n$. (Since we are merely interested in the performance of the algorithm, we can identify all return values with a single synchronisation state q.)

Using our prototype, we computed $\mathbb{E}[\mathbf{W}_n]$ and $\mathbb{E}[\mathbf{T}_n]$ for $p = 0.8$ and $n = 0, 1, \ldots, 10$. Figure 3 shows that $\mathbb{E}[\mathbf{W}_n]$ increases faster with n than $\mathbb{E}[\mathbf{T}_n]$; i.e., the parallelism increases.

[2] Available at `http://www.comlab.ox.ac.uk/people/Stefan.Kiefer/case-studies.mws`.

[3] Such an adaptive approximation scheme is called "local" in [20].

[4] That means, the oscillation n in the interval $[a,b]$ can be thought of as distributed between $[a,(a+b)/2]$ and $[(a+b)/2,b]$ according to a ball-and-urn experiment, where each of the n balls is placed in the $[a,(a{+}b)/2]$-urn and the $[(a{+}b)/2,b]$-urn with probability $p/2$, respectively, and in a trash urn with probability $1{-}p$.

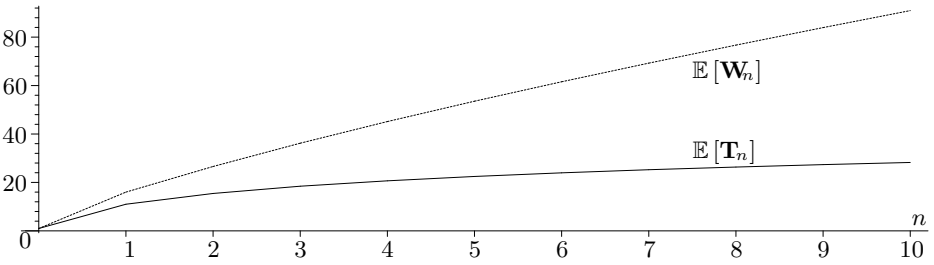

Fig. 3. Expectations of time and work.

4.2 Evaluation of Game Trees

The evaluation of game trees is a central task of programs that are equipped with "artificial intelligence" to play games such as chess. These game trees are min-max trees (see Figure 4): each node corresponds to a position of the game, and each edge from a

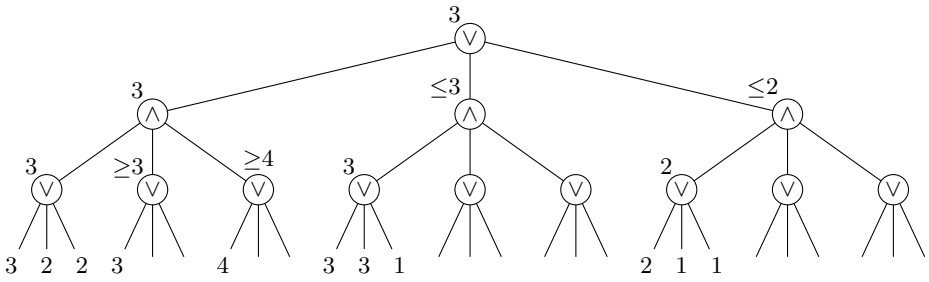

Fig. 4. A game tree with value 3.

parent to a child corresponds to a move that transforms the position represented by the parent to a child position. Since the players have opposing objectives, the nodes alternate between max-nodes and min-nodes (denoted $\vee$ and $\wedge$, respectively). A leaf of a game tree corresponds either to a final game position or to a position which is evaluated heuristically by the game-playing program; in both cases, the leaf is assigned a number. Given such a leaf labelling, a number can be assigned to each node in the tree in the straightforward way; in particular, *evaluating a tree* means computing the root value.

In the following, we assume for simplicity that each node is either a leaf or has exactly three children. Figure 5 shows a straightforward recursive parallel procedure for evaluating a max-node of a game tree. (Of course, there is a symmetrical procedure for min-nodes.)

Notice that in Figure 4 the value of the root is 3, independent of some missing leaf values. Game-playing programs aim at evaluating a tree as fast as possible, possibly by not evaluating nodes which are irrelevant for the root value. The classic technique is called *alpha-beta pruning*: it maintains an interval $[\alpha, \beta]$ in which the value of the current node is to be determined exactly. If the value turns out to be below α or above β, it is safe to return α or β, respectively. This may be the case even before all children have

```
function parMax(node)
  if node.leaf() then return node.val()
  else parallel ⟨ val1 := parMin(node.c1), val2 := parMin(node.c2), val3 := parMin(node.c3) ⟩
       return max{val1, val2, val3}
```

Fig. 5. A simple parallel program for evaluating a game tree.

been evaluated (a so-called *cut-off*). Figure 6 shows a sequential program for alpha-beta pruning, initially to be called "seqMax(root, $-\infty$, $+\infty$)". Applying seqMax to the tree from Figure 4 results in several cut-offs: all non-labelled leaves are pruned.

```
function seqMax(node, α, β)
  if node.leaf() then if node.val() ≤ α then return α
                      elsif node.val() ≥ β then return β
                      else return node.val()
  else val1 := seqMin(node.c1, α, β)
       if val1 = β then return β
       else val2 := seqMin(node.c2, val1, β)
            if val2 = β then return β
            else return seqMin(node.c3, val2, β)
```

Fig. 6. A sequential program for evaluating a game tree using alpha-beta pruning.

Although alpha-beta pruning may seem inherently sequential, parallel versions have been developed, often involving the *Young Brothers Wait (YBW)* strategy [16]. It relies on a good ordering heuristic, i.e., a method that sorts the children of a max-node (resp. min-node) in increasing (resp. decreasing) order, without actually evaluating the children. Such an ordering heuristic is often available, but usually not perfect. The tree in Figure 4 is ordered in this way. If alpha-beta pruning is performed on such an ordered tree, then either all children of a node are evaluated or only the first one. The YBW method first evaluates the first child only and hopes that this creates a cut-off or, at least, decreases the interval $[\alpha, \beta]$. If the first child fails to cause a cut-off, YBW *speculates* that both "younger brothers" need to be evaluated, which can be done in parallel without wasting work. A wrong speculation may affect the performance, but not the correctness. Figure 7 shows a YBW-based program. Similar code is given in [7] using *Cilk*, a C-based parallel programming language.

We evaluate the performance of these three (deterministic) programs using probabilistic assumptions about the game trees. More precisely, we assume the following: Each node has exactly three children with probability p, and is a leaf with probability $1-p$. A leaf (and hence any node) takes as value a number from $\mathbb{N}_4 := \{0, 1, 2, 3, 4\}$, according to a distribution described below. In order to model an ordering heuristic on the children, each node carries a parameter $e \in \mathbb{N}_4$ which intuitively corresponds to its expected value. If a max-node with parameter e has children, then they are min-nodes with parameters e, $e\ominus 1$, $e\ominus 2$, respectively, where $a\ominus b := \max\{a-b, 0\}$; similarly, the children of a min-node with parameter e are max-nodes with parame-

```
function YBWMax(node, α, β)
  if node.leaf() then if node.val() ≤ α then return α
                      elsif node.val() ≥ β then return β
                      else return node.val()
  else val1 := YBWMin(node.c1, α, β)
       if val1 = β then return β
       else parallel ⟨ val2 := YBWMin(node.c2, val1, β), val3 := YBWMin(node.c3, val1, β) ⟩
            return max{val2, val3}
```

Fig. 7. A parallel program based on YBW for evaluating a game tree.

ters e, $e{\oplus}1$, $e{\oplus}2$, where $a{\oplus}b := \min\{a{+}b, 4\}$. A leaf-node with parameter e takes value k with probability $\binom{4}{k} \cdot (e/4)^k \cdot (1{-}e/4)^{4-k}$; i.e., a leaf value is binomially distributed with expectation e. One could think of a game tree as the terminal tree of a branching process with $\Gamma = \{Max(e), Min(e) \mid e \in \{0,\ldots,4\}\}$ and $Q = \mathbb{N}_4$ and the rules $Max(e) \overset{p}{\hookrightarrow} \langle Min(e)\ Min(e{\ominus}1)\ Min(e{\ominus}2)\rangle$ and $Max(e) \overset{x(k)}{\longrightarrow} k$, with $x(k) := (1{-}p){\cdot}\binom{4}{k}{\cdot}(e/4)^k{\cdot}(1{-}e/4)^{4-k}$ for all $e, k \in \mathbb{N}_4$, and similar rules for $Min(e)$.

We model the YBW-program from Figure 7 running on such random game trees by the pSJS with $Q = \{0,1,2,3,4,q(\vee),q(\wedge)\} \cup \{q(\alpha,\beta,\vee,e), q(\alpha,\beta,\wedge,e) \mid 0 \le \alpha < \beta \le 4,\ 0 \le e \le 4\} \cup \{q(a,b) \mid 0 \le a,b \le 4\}$ and the following rules:

$$Max(\alpha,\beta,e) \xrightarrow{x(0)+\cdots+x(\alpha)} \alpha, \quad Max(\alpha,\beta,e) \xrightarrow{x(\beta)+\cdots+x(4)} \beta, \quad Max(\alpha,\beta,e) \xrightarrow{x(k)} k$$

$$Max(\alpha,\beta,e) \overset{p}{\hookrightarrow} \langle Min(\alpha,\beta,e)\ \ q(\alpha,\beta,\vee,e{\ominus}1)\rangle$$

$$\langle\beta\ \ q(\alpha,\beta,\vee,e)\rangle \overset{1}{\hookrightarrow} \beta, \qquad \langle\gamma\ \ q(\alpha,\beta,\vee,e)\rangle \overset{1}{\hookrightarrow} \langle Max2(\gamma,\beta,e)\ \ q(\vee)\rangle$$

$$Max2(\alpha,\beta,e) \overset{1}{\hookrightarrow} \langle Min(\alpha,\beta,e)\ \ Min(\alpha,\beta,e{\ominus}1)\rangle$$

$$\langle a\ \ b\rangle \overset{1}{\hookrightarrow} q(a,b), \qquad \langle q(a,b)\ \ q(\vee)\rangle \overset{1}{\hookrightarrow} \max\{a,b\}\,,$$

where $0 \le \alpha \le \gamma < \beta \le 4$ and $\alpha < k < \beta$ and $0 \le e \le 4$ and $0 \le a,b \le 4$. There are analogous rules with Min and Max exchanged. Notice that the rules closely follow the program from Figure 7. The programs parMax and seqMax from Figures 5 and 6 can be modelled similarly.

Let $T(\text{YBW}, p) := \mathbb{E}\left[\mathbf{T}_{Max(0,4,2)} \mid Run(Max(0,4,2){\downarrow}2)\right]$; i.e., $T(\text{YBW}, p)$ is the expected time of the YBW-program called with a tree with value 2 and whose root is a max-node with parameter 2. (Recall that p is the probability that a node has children.) Let $W(\text{YBW}, p)$ defined similarly for the expected work, and define these numbers also for par and seq instead of YBW, i.e., for the programs from Figures 5 and 6. Using our prototype we computed $W(\text{seq}, p) = 1.00, 1.43, 1.96, 2.63, 3.50, 4.68, 6.33$ for $p = 0.00, 0.05, 0.10, 0.15, 0.20, 0.25, 0.30$. Since the program seq is sequential, we have the same sequence for $T(\text{seq}, p)$. To assess the speed of the parallel programs par and YBW, we also computed the percentaged increase of their runtime relative to seq, i.e., $100 \cdot (T(\text{par}, p)/T(\text{seq}, p) - 1)$, and similarly for YBW. Figure 8 shows the results. One can observe that for small values of p (i.e., small trees), the program par is slightly faster than seq because of its parallelism. For larger values of p, par still evaluates all nodes in

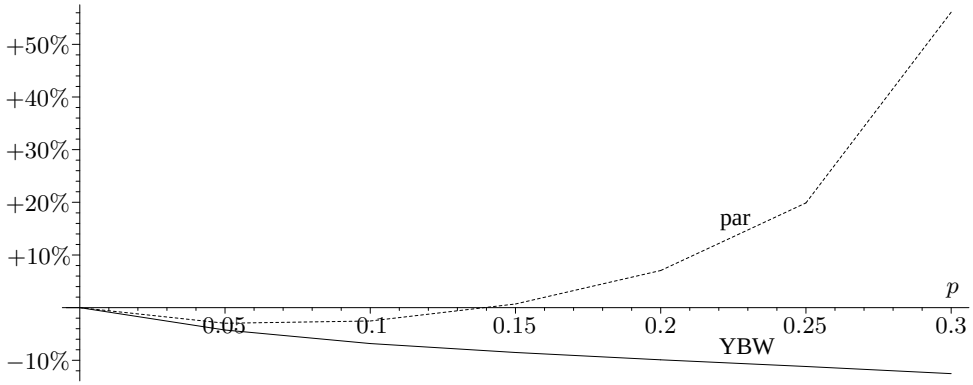

Fig. 8. Percentaged runtime increase of par and YBW relative to seq.

the tree, whereas seq increasingly benefits from cut-offs of potentially deep branches. Using Proposition 11, one can prove $W(\text{par}, \frac{1}{3}) = T(\text{par}, \frac{1}{3}) = \infty > W(\text{seq}, \frac{1}{3})$.[5] The figure also shows that the YBW-program is faster than seq: the advantage of YBW increases with p up to about 10%.

We also compared the *work* of YBW with seq, and found that the percentaged increase ranges from 0 to about $+0.4\%$ for p between 0 and 0.3. This means that YBW wastes almost no work; in other words, a sequential version of YBW would be almost as fast as seq. An interpretation is that the second child rarely causes large cut-offs. Of course, all of these findings could depend on the exact probabilistic assumptions on the game trees.

5 Conclusions and Future Work

We have introduced pSJSs, a model for probabilistic parallel programs with process spawning and synchronisation. We have studied the basic performance measures of termination probability, space, work, and time. In our results the upper complexity bounds coincide with the best ones known for pPDSs, and the lower bounds also hold for pPDSs. This suggests that analysing pSJSs is no more expensive than analysing pPDSs. The pSJS model is amenable to a practical performance analysis. Our two case studies have demonstrated the modelling power of pSJSs: one can use pSJSs to model, analyse, and compare the performance of parallel programs under probabilistic assumptions.

We intend to develop model-checking algorithms for pSJSs. It seems to us that a meaningful functional analysis should not only model-check the Markov chain induced by the pSJS, but rather take the individual process "histories" into account.

Acknowledgements. We thank Javier Esparza, Alastair Donaldson, Markus Müller-Olm, Luke Ong and Thomas Wahl for helpful discussions on the non-probabilistic version of pSJSs. We also thank the anonymous referees for valuable comments.

[5] In fact, $W(\text{seq}, p)$ is finite even for values of p which are slightly larger than $\frac{1}{3}$; in other words, seq cuts off infinite branches.

References

1. E. Allender, P. Bürgisser, J. Kjeldgaard-Pedersen, and P. B. Miltersen. On the complexity of numerical analysis. In *IEEE Conf. on Computational Complexity*, pages 331–339, 2006.
2. K.B. Athreya and P.E. Ney. *Branching Processes*. Springer, 1972.
3. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proceedings of CONCUR'05*, pages 473–487. Springer, 2005.
4. T. Brázdil, J. Esparza, and S. Kiefer. On the memory consumption of probabilistic pushdown automata. In *Proceedings of FSTTCS*, pages 49–60, 2009.
5. T. Brázdil, S. Kiefer, A. Kučera, and I.H. Vařeková. Runtime analysis of probabilistic programs with unbounded recursion. 2010. Submitted for publication. Available at `http://arxiv.org/abs/1007.1710`.
6. J. Canny. Some algebraic and geometric computations in PSPACE. In *STOC'88*, pages 460–467, 1988.
7. D. Dailey and C.E. Leiserson. Using Cilk to write multiprocessor chess programs. *The Journal of the International Computer Chess Association*, 2002.
8. J.S. Danaher, I.A. Lee, and C.E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming (SCP)*, 63(2):147–171, 2006.
9. J. Esparza and K. Etessami. Verifying probabilistic procedural programs. In *FSTTCS 2004*, pages 16–31, 2004.
10. J. Esparza, S. Kiefer, and M. Luttenberger. Computing the least fixed point of positive polynomial systems. *SIAM Journal on Computing*, 39(6):2282–2335, 2010.
11. J. Esparza, A. Kučera, and R. Mayr. Model checking probabilistic pushdown automata. In *LICS'04*, pages 12–21. IEEE, 2004.
12. J. Esparza, A. Kučera, and R. Mayr. Quantitative analysis of probabilistic pushdown automata: Expectations and variances. In *LICS'05*, pages 117–126. IEEE, 2005.
13. K. Etessami and M. Yannakakis. Algorithmic verification of recursive probabilistic state machines. In *TACAS 2005*, pages 253–270, 2005.
14. K. Etessami and M. Yannakakis. Recursive concurrent stochastic games. *Logical Methods in Computer Science*, 4(4), 2008.
15. K. Etessami and M. Yannakakis. Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. *Journal of the ACM*, 56(1):1–66, 2009.
16. R. Feldmann, B. Monien, P. Mysliwietz, and O. Vornberger. Distributed game-tree search. *ICCA Journal*, 12(2):65–73, 1989.
17. T.E. Harris. *The Theory of Branching Processes*. Springer, 1963.
18. S. Kiefer and D. Wojtczak. On probabilistic parallel programs with process creation and synchronisation. Technical report, arxiv.org, 2010. Available at `http://arxiv.org/abs/1012.2998`.
19. C. Löding. Reachability problems on regular ground tree rewriting graphs. *Theory of Computing Systems*, 39:347–383, 2006.
20. M.A. Malcolm and R.B. Simpson. Local versus global strategies for adaptive quadrature. *ACM Transactions on Mathematical Software*, 1(2):129–146, 1975.
21. R. Mayr. Process rewrite systems. *Information and Computation*, 156(1-2):264–286, 2000.
22. J. Renegar. On the computational complexity and geometry of the first-order theory of the reals. Parts I–III. *Journal of Symbolic Computation*, 13(3):255–352, 1992.
23. M. Yannakakis and K. Etessami. Checking LTL properties of recursive Markov chains. In *QEST 2005*, pages 155–165, 2005.