Error-Free Stable Computation with Polymer-Supplemented Chemical Reaction Networks

Allison Tai(✉) and Anne Condon

University of British Columbia, Vancouver, BC V6T 1Z4, Canada
{tyeuyang,condon}cs.ubc.ca

Abstract. When disallowing error, traditional chemical reaction networks (CRNs) are very limited in computational power: Angluin et al. and Chen et al. showed that only semilinear predicates and functions are stably computable by CRNs. Qian et al. and others have shown that polymer-supplemented CRNs (psCRNs) are capable of Turing-universal computation. However, their model requires that inputs are pre-loaded on the polymers, in contrast with the traditional convention that inputs are represented by counts of molecules in solution. Here, we show that psCRNs can stably simulate Turing-universal computations even with solution-based inputs. However, such simulations use a unique "leader" polymer per input type and thus involve many slow bottleneck reactions. We further refine the polymer-supplemented CRN model to allow for anonymous polymers, that is, multiple functionally-identical copies of a polymer, and provide an illustrative example of how bottleneck reactions can be avoided in this new model.

Keywords: Stable computation · Chemical reaction networks · DNA polymers

1 Introduction

The logical, cause-and-effect nature of chemical reactions has long been recognized for its potential to carry information and make decisions. Indeed, biological systems exploit interacting digital molecules to perform many important processes. Examples include inheritance with DNA replication, passing information from the nucleus to the cytoplasm using messenger RNA, or activating different cellular states through signal transduction cascades. Chemical reaction networks (CRNs) exploit these capabilities to perform molecular computations, using a finite set of molecular species (including designated input and output species) and reactions. Reactions among molecules in a well-mixed solution correspond to computation steps. CRN models may use mass-action kinetics, where the dynamics of the reactions are governed by ordinary differential equations, or stochastic kinetics, where the choice of reaction and length of time between reactions depends on counts of molecular species. We focus on stochastic CRNs in this work.

C. Thachuk and Y. Liu (Eds.): DNA 25, LNCS 11648, pp. 197–218, 2019.
https://doi.org/10.1007/978-3-030-26807-7_11

Stochastic CRNs using unbounded molecular counts are Turing-universal [1], but have a non-zero chance of failure. One challenge is that CRNs are unable to detect the absence of a molecule, and therefore when all molecules of a particular species have been processed. For example, when trying to simulate a register machine, if their count of a species corresponds to a register value, then a test-for-zero instruction needs to detect when all molecules have been depleted, i.e., their count is zero. Indeed, while the error of a CRN simulation of Turing-universal computation can be made arbitrarily small, it can never reach zero [2].

Error-free CRNs include those that exhibit *stable computation*: the output can change as long as it eventually converges to the correct answer; and *committing computation*: the presence of a designated "commit" species indicates that the output is correct and does not change subsequently. The class of predicates stably computable by CRNs is limited to semilinear predicates [3], and functions computable by committing CRNs are just the constant functions [2].

Cummings et al. [4] introduced the notion of *limit-stable computation*, which relaxes the stability requirement. In a computation of a limit-stable CRN, the output may change repeatedly, but the probability of changing the output from its correct value goes to zero in the limit. Cummings et al. show that any halting register machine can be simulated by a limit-stable CRN. Their construction involves repeated simulations of a register machine, resetting the register values each time, along with slowing down any error-prone reactions each time they occur. They show that the computational power then becomes equivalent to that of a Turing machine with the ability to change its output a finite number of times, capable of deciding predicates in the class Δ_2^0 of limit-computable function.

From these insights, we can see that CRN computations that produce the correct answer with probability 1 are still severely limited. We ask, "Is there any way to extend CRNs to work around the lack of ability to detect absence?" Qian et al. [5] gave a promising answer to this question by introducing a CRN model that is supplemented by polymers that behave as stacks, onto which monomers can be pushed and popped. Most importantly, this extended model allows for the stack base unit $\perp$ to be used as a reactant in implementing a "stack empty" operation. Indeed, Qian et al. use this operation in an error-free simulation of a stack machine. The resulting protocol, however, requires that the entire input is pre-loaded onto one of the stacks, a large change from traditional CRNs which assumes the inputs are well-mixed in a solution.

Motivated by the work of Qian et al., we wish to go one step further: Is Turing-universal stable computation by polymer-supplemented CRNs (psCRNs) possible when the input is represented by counts of monomers in a well-mixed solution? Intuitively, if input monomers can be loaded on to a polymer, absence of that species from the system could be detected by emptiness of the polymer, and thus circumvent a significant barrier to error-free, Turing-universal computation. The obvious obstacle is that it seems impossible to guarantee that all inputs are loaded on the polymer before computation can begin, if we can't reliably check for absence of inputs in the environment. At first glance, the logic appears circular, but we show that indeed stable Turing-universal computation is possible, and also present ideas for speeding up such computations.

In the rest of this section we describe our four main contributions and review related work. Section 2 introduces our polymer CRN models, Sects. 3, 4, and 5 describe our results, and Sect. 6 concludes with a summary and directions for future work.

1.1 Contributions and Highlights

Stable Register Machine Simulation Using CRNs with Leader Polymers. We design a polymer-supplemented CRN (psCRN) that simulates a register machine, assuming that all inputs are pre-loaded on polymers, with one polymer per species. We then augment the simulator CRN with CRNs that detect when an input has been loaded, and that restarts the simulator in this case. This scheme is similar to the error correction scheme of Cummings et al., but leverages polymers to ensure stable computation. Our polymer simulation of register machines, and thus Turing-universal computation, has a unique polymer per input species, as well as a "program counter" to ensure that execution of reactions follows the proper order. In the parlance of traditional CRNs, these molecules serve as "leaders". As a consequence, the simulation has a high number of so-called bottleneck reactions, which involve two leader reactants. Bottleneck reactions are undesirable because they are slow.

Anonymous Polymers Can Help Avoid Bottleneck Reactions. To avoid bottleneck reactions, we propose a CRN polymer model with no limit on the number of polymers of a given species, other than the limit posed by the volume of the system. In addition to type-specific increment, decrement and test-if-empty operations (which are applied to one polymer of a given species), polymer stubs can be created or destroyed. We call such polymers "anonymous". We illustrate the potential of psCRNs with anonymous polymers to reduce bottleneck reactions, by describing psCRN to compute $f(n) = n2^{\lfloor \lg n \rfloor}$ (which is the same as n^2 when n is a power of 2).

Abstractions for Expressing CRN Multi-threading and Synchronization. Our CRN for $f(n) = n2^{\lfloor \lg n \rfloor}$ uses threading to ensure that polymer reactions can happen in parallel, and uses a "leader" polymer for periodic synchronization. To describe our psCRN, we develop threading abstractions for psCRNs with anonymous polymers.

Time Complexity and a Simulator for CRNs with Anonymous Polymers. To test the correctness of our psCRNs and evaluate their running times, we developed a custom CRN simulator designed to support anonymous polymers and their associated reactions. Underlying our simulator is a stochastic model of psCRN kinetics that is a natural extension of traditional stochastic CRNs and population protocols. We also use this model to analyze the expected time complexities of our psCRNs examples in this paper, showing how speedups are possible with anonymous polymers.

1.2 Related Work

Soloveichik et al. [6] demonstrated how Turing-universal computation is possible with traditional stochastic CRNs, achieving arbitrarily small (but non-zero) error probability. For the CRN model without polymers Cummings et al. [4] showed how to reset computations so as to correct error and achieve limit-stable computation (which is weaker than stable computation).

In order to understand the inherent energetic cost of computation, Bennett [7,8] envisioned a polymer-based chemical computer, capable of simulating Turing machines in a logically reversible manner. Qian et al. [5] introduced a stack-supplemented CRN model in which inputs are pre-loaded on stacks, and showed how the model can stably simulate stack machines. Johnson et al. [9] introduce a quite general linear polymer reaction network (PRN) model for use with simulation and verification, as opposed to computation. Cardelli et al. [10] also demonstrated Turing-universal computation using polymers, using process algebra systems, but these systems are not stochastic. Jiang et al. [11] also worked on simulating computations with mass-action chemical reactions, using a chemical clock to synchronize reactions and minimize errors.

Lakin et al. [12] described polymerizing DNA strand displacement systems, and showed how to model and verify stack machines at the DSD level. They also simulated their stochastic systems using a "just-in-time" extension of Gillespie's algorithm. Their model has a single complex to represent a stack. Recognizing limitations of this, they noted that "it would be desirable to invent an alternative stack machine design in which there are many copies of each stack complex...", which is what we do in this paper. They propose that updates to stacks could perhaps be synchronized using a clock signal such as that proposed by Jiang et al. [11]. In contrast, our synchronization mechanism is based on detection of empty polymers.

The population protocol (PP) model introduced by Angluin et al. [13], which is closely related to the CRN model, focuses on pairwise-interacting agents that can change state. In Angluin et al.'s model, agents in a PP are finite-state. An input to a computation is encoded in the agents' initial states; the number of agents equals the input size. Any traditional CRN can be transformed into a PP and vice versa. Chatzigiannakis et al. [14] expand the n agents to be Turing machines, then examine what set of predicates such protocols can stably compute using $O(\log n)$ memory. Although the memory capacity of our polymers can surpass $O(\log n)$, polymer storage access is constrained to be that of a counter or stack, unlike the model of Chatzigiannakis et al.

2 Polymer-Supplemented Chemical Reaction Networks

A polymer-supplemented stochastic chemical reaction network (psCRN) models the evolution of interacting molecules in a well-mixed volume, when monomers can form polymers. We aim for simplicity in our definitions here, providing just enough capability to communicate the key ideas of this paper. Many aspects of our definitions can be generalized, for example by allowing multiple monomer

species in a polymer, or double-end polymer extensibility, as is done in the work of Johnson et al. [9], Lakin et al. [12], Qian et al. [5], and others.

Reactions. A traditional CRN describes reactions involving molecules whose species are given by a finite set Σ. A *reaction*

$$r + r' \longrightarrow p + p' \tag{1}$$

describes what happens when two so-called reactant molecules of species $r, r' \in \Sigma$ collide: they produce molecules of species $p \in \Sigma$ and $p' \in \Sigma$. We assume that all reactions have exactly two reactants and two products, that the multi-sets $\{r, r'\}$ and $\{p, p'\}$ are not equal, that for any r and r', there is at most one reaction with reactants of species r and r'. For now we do not ascribe a rate constant to a reaction; we will do that in Sect. 5.

Polymer-supplemented chemical reaction networks (psCRNs) also have reactions pertaining to polymers. A designated subset $\Sigma^{(m)}$ of Σ is a set of *monomers*. A *polymer* of type $\sigma \in \Sigma^{(m)}$, which we also call a σ-polymer, is a string $\bot_\sigma \sigma^i$, $i \geq 0$; its *length* is i and we say that the polymer is *empty* if its length is 0. We call $\bot_\sigma$ a *stub* and let $\bot = \{\bot_\sigma \mid \sigma \in \Sigma^{(m)}\} \subset \Sigma$. Reactions can produce stubs from molecules of other species in Σ; this is an important way in which our model differs from previous work [5,12]. Polymer reactions also involve molecules in a set $\mathcal{A} = \{A_\sigma \mid \sigma \in \Sigma^{(m)}\}$ of *active query molecules*, where $\mathcal{A} \subseteq \Sigma - \Sigma^{(m)}$. For each $\sigma \in \Sigma^{(m)}$ there is a reversible polymer reaction, with the forwards and backwards directions corresponding to σ-push and σ-pop, respectively:

$$[\bot_\sigma \ \ldots] + \sigma \rightleftharpoons [\bot_\sigma \ \ldots \sigma] + A_\sigma.$$

Later, we will introduce "inactive" variants of the A_σ molecules, both to help control when pushes and pops happen, and to help track whether a polymer is empty (has length 0) or not.

Configurations. A *configuration* specifies how many molecules of each species are in the system, keeping track also of the lengths of all σ-polymers. Formally, a configuration is a mapping $\mathbf{c} : \Sigma \cup \{\bot_\sigma \sigma^i \mid i \geq 1\} \to \mathbb{N}$, where $\mathbb{N}$ is the set of nonnegative integers. We let $\mathbf{c}([\bot_\sigma \ldots])$ denote total number of σ-polymers in the system (including stubs) and let $\mathbf{c}([\bot_\sigma \ldots \sigma])$ denote total number of σ-polymers in the system that have length at least 1. With respect to configuration $\mathbf{c}$, we say that a molecule of species $\sigma \in \Sigma$ is a *leader* if $\mathbf{c}(\sigma) = 1$, and we say that a σ-polymer is a leader if $\mathbf{c}([\bot_\sigma \ldots]) = 1$.

A reaction of type (1) is *applicable* to configuration $\mathbf{c}$ if, when $r \neq r'$, $\mathbf{c}(r) \geq 1$ and $\mathbf{c}(r') \geq 1$, and when $r = r'$, $\mathbf{c}(r) \geq 2$. If the reaction is applied to $\mathbf{c}$, a new configuration $\mathbf{c}'$ is reached, in which the counts of r and r' decrease by 1 (when $r = r'$ the count of r decreases by 2), the counts of p and p' increase by 1 (when $p = p'$ the count of p increases by 2), and all other counts remain unchanged.

A σ-push is applicable if $\mathbf{c}([\bot_\sigma \ldots]) > 0$ and $\mathbf{c}(\sigma) > 0$, and a σ-pop is applicable if $\mathbf{c}([\bot_\sigma \ldots \sigma]) > 0$ and $\mathbf{c}(A_\sigma) > 0$. The *result* of applying a σ-push reaction is that $\mathbf{c}(\sigma)$ decreases by 1, $\mathbf{c}(A_\sigma)$ increases by 1 and also for exactly one

$i \geq 0$ such that $\mathbf{c}(\perp_\sigma \sigma^i) > 0$, $\mathbf{c}(\perp_\sigma \sigma^i)$ decreases by 1 and $\mathbf{c}(\perp_\sigma \sigma^{i+1})$ increases by 1. Similarly, the *result* of applying a σ-pop reaction is that $\mathbf{c}(\sigma)$ increases by 1, $\mathbf{c}(A_\sigma)$ decreases by 1, and for exactly one $i \geq 1$ such that $\mathbf{c}(\perp_\sigma \sigma^i) > 0$, $\mathbf{c}(\perp_\sigma \sigma^i)$ decreases by 1 and $\mathbf{c}(\perp_\sigma \sigma^{i-1})$ increases by 1. Intuitively, the length of one σ-polymer in the system either grows or shrinks by 1 and correspondingly the count of A_σ either increases or decreases by 1. The affected polymer is chosen nondeterministically; exactly how the polymer is chosen is not important in the context of stable computation. For example, the polymer could be chosen uniformly at random, consistent with the model of Lakin and Phillips [12]. We defer further discussion of this to Sect. 5.

If $\mathbf{c}'$ results from the application of some reaction to $\mathbf{c}$, we write $\mathbf{c} \to \mathbf{c}'$ and say that $\mathbf{c}'$ is *directly reachable* from $\mathbf{c}$. We say that $\mathbf{c}'$ is reachable from $\mathbf{c}$ if for some $k \geq 0$ and configurations $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k$,

$$\mathbf{c} \to \mathbf{c}_1 \to \mathbf{c}_2 \ldots \to \mathbf{c}_k \to \mathbf{c}'.$$

Computations and Stable Computations. We're interested in CRNs that compute, starting from some initial configuration $\mathbf{c}_0$ that contains an input. For simplicity, we focus on CRNs that compute functions $f : \mathbb{N}^k \to \mathbb{N}$. For example, the function may be Square, namely $f(n) = n^2$.

In a function-computing psCRN, the input $\mathbf{n} = (n_1, \ldots, n_k) \in \mathbb{N}^k$ is represented by counts of species in a designated set $\mathcal{I} = \{X_1, X_2, \ldots, X_k\} \subseteq \Sigma^{(m)}$ and the output is represented by the count of a different designated species $Y \in \Sigma^{(m)}$. In the initial configuration $\mathbf{c}_0 = \mathbf{c}_0(n)$, the initial counts of the input species X_i is n_i, $1 \leq i \leq k$, and the counts of all species other than the input species, including polymers and active query molecules, is 0, with the following exceptions. First, there may be some leader molecules or polymers present. Second, the count of a designated "blank" species $B \in \Sigma$ may be positive. Blank molecules are useful in order to keep all reactions bimolecular, since a unimolecular reaction $r \to p$ can be replaced by $r + B \to p + B$ (if B's are guaranteed to always be present). Blanks can also be used to create new copies of a particular molecular species.

A *computation* of a psCRN is a sequence of configurations starting with an initial configuration $\mathbf{c}_0$, such that each configuration (other than the first) is directly reachable from its predecessor. Let $\mathcal{C}$ be a psCRN, and let $\mathbf{c}$ be a configuration of $\mathcal{C}$. We say that $\mathbf{c}$ is *stable* if for all configurations $\mathbf{c}'$ reachable from $\mathbf{c}$, $\mathbf{c}(Y) = \mathbf{c}'(Y)$, where Y is the output species. The psCRN *stably computes* a given function $f : \mathbb{N}^k \to \mathbb{N}$ if on any input $\mathbf{n} \in \mathbb{N}^k$, for any configuration $\mathbf{c}$ reachable from $\mathbf{c}_0(\mathbf{n})$, a stable configuration $\mathbf{c}'$ is reachable from $\mathbf{c}$ and moreover, $\mathbf{c}'(Y) = f(n)$. Finally if psCRN $\mathcal{C}$ stably computes a given predicate, we say that $\mathcal{C}$ is *committing* if $\mathcal{C}$ has a special "commit" species L_H such that for all $\mathbf{n} \in \mathbb{N}^k$, for any configuration $\mathbf{c}$ reachable from $\mathbf{c}_0(\mathbf{n})$ that contains species L_H, if $\mathbf{c}'$ is reachable from $\mathbf{c}$ then $\mathbf{c}'$ also contains L_H and $\mathbf{c}'(Y) = f(n)$.

Bottleneck Reactions. In our CRN algorithms of Sect. 3, many reactions involve a leader molecule, representing a program counter, that reacts with a leader

polymer. Such reactions, in which the count of both reactants is 1, is often described as a bottleneck reaction [15]. As explained in Sect. 5, in a stochastic CRN that executes in a well-mixed system with volume V, the expected time for such a reaction is $\Theta(V)$ [6]. Our motivation for the anonymous polymer model in Sect. 4 is to explore how to compute with polymers in a way that reduces bottleneck reactions.

3 Stable, Turing-Universal Computation by Sequential PsCRNs with Leader Polymers

Here we describe how psCRNs with leader polymers can stably simulate register machines, thereby achieving Turing-universal computation. Before doing so, we first introduce psCRN "pseudocode" which is convenient for describing psCRN algorithms. Then, as an illustration, we describe a psCRN to compute the Square function $f(n) = n^2$. We first do this for a slightly different input convention than that described in Sect. 2: we assume that all input molecules are "pre-loaded" on polymers. For this pre-loaded input model, building strongly on a construction of Cummings et al. [4], we show how committing psCRNs can simulate register machines, thereby achieving Turing-universal computation. Finally, we remove the requirement that the input is pre-loaded by adding mechanisms to detect when an input is loaded, and to restart the simulator in this case.

Sequential psCRN Pseudocode. Following earlier work [4–6,16], we describe a psCRN program as a sequence of instructions, ordered consecutively starting at 1. Because one instruction must finish before moving on to the next, we call these *sequential* psCRNs. Corresponding to each instruction number i is a molecular "program counter" species $L_i \in \Sigma$. One copy of L_1 is initially present, and no other $L_{i'}$ for $i' \neq i$ is initially present.

The instructions $\texttt{inc}(\sigma)$ and $\texttt{dec}(\sigma)$ of Table 1 increase and decrease the length of a σ-polymer by 1, respectively, making it possible to use the polymers as counters. We assume that always a sufficient number of blanks are in the system in order for the `inc`() instruction to proceed. In order to ensure that the push and pop reactions happen only within the `inc`() and `dec`() reactions, the $\texttt{inc}(\sigma)$ operation generates the active query A_σ, which is converted into an inactive variant $I_\sigma \in \Sigma - \Sigma^{(m)}$ before the instruction execution completes, and the $\texttt{dec}(\sigma)$ instruction reactivates A_σ in order to reduce the length of a σ-polymer by 1. If a psCRN executes only instructions of Table 1, starting from an initial configuration in which there is no polymer of length greater than 0, then we have the following invariant:

> **Invariant:** Upon completion of any instruction, the count of I_σ equals the sum of the lengths of σ-polymers.

The `jump-if-empty` instruction is useful when there is a leader σ-polymer. This σ-polymer is empty (has length 0) if and only if a stub $\perp_\sigma$ is in the system. Assuming that our invariant holds, the leader σ-polymer is not empty if

and only if at least one I_σ molecule is in the system. Either way, the instruction ensures that the program counter advances properly. When the σ-polymer is empty, the `dec`(σ) cannot proceed and causes an algorithm to stall. The `jump-if-empty`(σ, k) instruction provides a way to first check whether the σ-polymer is empty, and if not, `dec`(σ) can safely be used. The `create` and `destroy` instructions provide a way to create and destroy copies of a species. For clarity, we also include `create-polymer`(σ) and `destroy-polymer`(σ) instructions, which create and destroy the stub $\perp_\sigma$, respectively. While more than one reaction is needed to implement one instruction, all will have completed when the instruction has completed and the program counter is set to the number of the next instruction to be executed in the pseudocode.

Table 1. Instruction abstractions of psCRN reactions. The decrement dec(σ) instruction can complete only if some σ-polymer has length is at least 1.

i: `inc`(σ)	$L_i + B \longrightarrow L_i^* + \sigma$ $\sigma + [\perp_\sigma \ldots] \rightleftharpoons A_\sigma + [\perp_\sigma \ldots \sigma]$ $L_i^* + A_\sigma \longrightarrow L_{i+1} + I_\sigma$
i: `dec`(σ)	$L_i + I_\sigma \longrightarrow L_i^* + A_\sigma$ $A_\sigma + [\perp_\sigma \ldots \sigma] \rightleftharpoons \sigma + [\perp_\sigma \ldots]$ $L_i^* + \sigma \longrightarrow L_{i+1} + B$
i: `jump-if-empty`(σ, k)	$L_i + \perp_\sigma \longrightarrow L_k + \perp_\sigma$ $L_i + I_\sigma \longrightarrow L_{i+1} + I_\sigma$
i: `goto`(k)	$L_i + B \longrightarrow L_k + B$
i: `create`(σ)	$L_i + B \longrightarrow L_{i+1} + \sigma$
i: `destroy`(σ)	$L_i + \sigma \longrightarrow L_{i+1} + B$
i: `create-polymer`(σ)	$L_i + B \longrightarrow L_{i+1} + \perp_\sigma$
i: `destroy-polymer`(σ)	$L_i + \perp_\sigma \longrightarrow L_{i+1} + B$
i: `halt`	$L_i + B \longrightarrow L_H + B$

Pseudocode instructions may also be function calls, where a function is itself a sequence of instructions expressed as pseudocode. Suppose again that there is a leader σ-polymer and also a leader σ'-polymer in the system. Then the `copy`(σ, σ') function (using a temporary τ-polymer) extends the length of the σ'-polymer by the length of the σ-polymer. Another useful function is `flush`(σ) which decrements the (leader) σ-polymer until its length is 0. A third function, `release-output`(σ), is useful to "release" molecules on a (leader) σ-polymer as Y molecules into the solution. This function uses an additional special leader Y'-polymer which is empty in the initial configuration, and whose length at the end of the function equals the number of released Y molecules. The Y' molecule will be useful later, when we address how a psCRN can be restarted (and should not be used elsewhere in the code).

i: copy(σ, σ')	i : goto$(i.1)$
	$i.1$: create-polymer(τ)
	$i.2$: jump-if-empty$(\sigma, i.7)$
	$i.3$: dec(σ)
	$i.4$: inc(σ')
	$i.5$: inc(τ)
	$i.6$: goto$(i.2)$
	$i.7$: jump-if-empty$(\tau, i.11)$
	$i.8$: dec(τ)
	$i.9$: inc(σ)
	$i.10$: goto$(i.7)$
	$i.11$: destroy-polymer(τ)
	$i.12$: goto$(i+1)$

i: flush(σ)	i: goto$(i.1)$
	$i.1$: jump-if-empty$(\sigma, i+1)$
	$i.2$: dec(σ)
	$i.3$: goto$(i.1)$

i: release-output(σ)	i: goto$(i.1)$
	$i.1$ jump-if-empty$(\sigma, i+1)$
	$i.2$ dec(σ)
	$i.3$ inc(Y')
	$i.4$ create(Y)
	$i.5$ goto$(i.1)$

Numbering of Function Instructions. For clarity, we use $i.1, i.2$, and so on to label the lines of a function called from line i of the main program. Upon such a function call, the CRN's program counter first changes from L_i to $L_{i.1}$. The program counter is restored to L_{i+1} upon completion of the function's instructions, e.g., via a goto$(i+1)$ instruction or a jump-if-empty$(\sigma, i+1)$ instruction. If one function f_B is called from line $a.b$ of another function f_A, the program counter labels would be $a.b.1$, $a.b.2$ and so on, and so the label "i" in the function description should be interpreted as "$a.b$". In this case, when the function f_B completes, control is passed back to line $a.(b+1)$ of function f_A; that is, the "goto$(i+1)$" statement should be interpreted as "goto$(a.(b+1))$". Also for clarify, we use special labeling of instructions in a few special places, such as the restart function below, in which instructions are labeled s1, s2 and so on.

psCRNs with Pre-loaded Inputs. As noted in the introduction, a challenge in achieving stable computation with psCRNs is detecting the absence of inputs. To build up to our methods for addressing this challenge, we first work with a more convenient convention, that of pre-loaded inputs. By this we mean that if the input contains n_i molecules of a given species X_i, then in the initial configuration there is a unique X_i-polymer of length n_i (the "pre-loaded" polymer). Furthermore, there are n_i copies of the inactive query molecule I_{X_i} in the system. Intuitively, the pre-loaded initial configuration is one that would be reached

if n_i `inc`(X_i) operations were performed from an initial configuration with no inputs and an empty X_i-polymer, for every input species X_i.

A Committing, Sequential psCRN with Pre-loaded Inputs for Square. Our psCRN for the Square function $f(n) = n^2$ has one input species X and one output species Y. In the pre-loaded initial configuration, the input is represented as the length n of a leader X-polymer, and the count of I_X is n. The number of blanks in the initial configuration must be greater than n^2, since blanks are used to produce the n^2 output molecules. The only other molecule in the initial configuration is the leader program counter L_1. The psCRN has a loop (implemented using `jump-if-empty` and `goto`) that executes n times, adding n to an intermediate Y_{int}-polymer each time. When the loop completes, the output is released from the Y_{int}-polymer in the form of Y, so that the number of Y's in solution is n^2, and the psCRN halts. The halting state is in effect a committing state, since no transition is possible from L_H.

Algorithm 1. Sequential-n^2-psCRN, with input n pre-loaded on X-polymer.

```
1: create-polymer(X′)
2: create-polymer(Y_int)
3: copy(X, X′)
4: jump-if-empty(X′,8)
5:     dec(X′)
6:     copy(X, Y_int)
7:     goto(4)
8: release-output(Y_int)
9: halt
```

Committing Turing-Universal Computation by psCRNs with Pre-loaded Inputs. Turing-universal computation is possible with register machines (RMs). To simulate a halting register machine that computes function $f : \mathbb{N}^k \to \mathbb{N}$ with $r \geq k$ unary registers, a psCRN has r unary counters $R_1, R_2, \ldots, R_r$, the first k of which initially contain the input counts $n_1, n_2, \ldots, n_k$, while the others are initially 0. Throughout the simulation of the register machine, the psCRN has exactly one R_l-polymer for each register R_l, $1 \leq l \leq r$. In addition, there is one additional polymer, a Y'-polymer, which is initially empty and is used by the `release-output` function. A register machine program is a sequence of instructions, where instructions can increment a register; decrement a non-empty register; test if a register is empty (0) and jump to a new instruction if so; or halt. Table 1 already shows how all four of these instructions can be implemented using a psCRN. We assume in what follows that these are the only instructions used by the psCRN simulator; in particular, no additional registers (polymers) are ever created or destroyed. If register R_r is used to store the output, then the output is released into solution, using `release-output`(R_r), once the machine being simulated reaches its halt state. We assume that `release-output`(R_r) is the only function call of the RM simulator.

Stable, Turing-Universal Computation by psCRNs. We now handle the case that the input is represented as counts of molecules, rather than pre-loaded polymers. That is, in the initial configuration of the psCRN all polymers of types $R_1, R_2, \ldots, R_r$ are empty, and instead, for each input n_l, $1 \leq l \leq k$, there are n_l copies of molecule R_l in solution. Our scheme uses the R_l-push reaction to load inputs. We add CRNs to detect input-loading and to restart the simulator in this case. Once all inputs are loaded, the system is never subsequently restarted. Overall our simulation has four components:

- **Input loading:** This is done as R_l-push, which can happen at any time until all inputs are loaded. Recall that the R_l-push reactions are
$$[\perp_{\mathrm{R_1}} \ldots] + \mathrm{R}_l \longrightarrow [\perp_{\mathrm{R_1}} \ldots \mathrm{R}_l] + A_{R_l}, 1 \leq l \leq k.$$
Each such reaction generates an active query molecule A_{R_l} which, as explained below, triggers input detection.
- **Register machine (RM) simulation:** Algorithm 2 shows the simulator in the case of three input registers. This psCRN program has a "prelude" phase that starts by creating three new polymers R'_1, R'_2 and R'_3 (lines P1, P2, and P3), and then copies the input register polymers R_1, R_2, and R_3 to polymers R'_1, R'_2, and R'_3, respectively (lines P4, P5, and P6). Then starting from line numbered 1, the simulation uses register R'_l rather than R_l, $1 \leq l \leq 3$, as well as the remaining initially empty registers $R_4, \ldots, R_r$. Upon completion of the computation, the output is released from register R_r, and the simulator halts (produces the L_H species).

Algorithm 2. Sequential-RM-psCRN, $k = 3$ input registers, r registers in total.

```
P1: create-polymer(R'_1)
P2: create-polymer(R'_2)
P3: create-polymer(R'_3)
P4: copy(R_1, R'_1)
P5: copy(R_2, R'_2)
P6: copy(R_3, R'_3)
 1: // Rest of psCRN simulation pseudocode here, with
 2: // R_1, R_2 and R_3 replaced by R'_1, R'_2 and R'_3
  :      ...
  : // ending with release-output(R_r) function and halt instruction.
```

- **Input detection:** This is triggered by the presence of an active query molecule A_{R_l}. For each value i of the main program counter after the prelude phase (i.e., after lines P1 through P6) and for L_H we have the following reactions, where S1 is the first number of the `restart` pseudocode (see below). The reactions convert the active A_{R_l} molecule into its inactive counterpart, I_{R_l}, since the input molecule is now loaded, and also changes the program counter to L_{S1}, which triggers restart.
$$L_i + A_{R_l} \longrightarrow L_{\mathrm{S1}} + I_{R_l}, 1 \leq l \leq k,$$
$$L_H + A_{R_l} \longrightarrow L_{\mathrm{S1}} + I_{R_l}, 1 \leq l \leq k.$$

L_H is no longer a committing species, since it may change to L_{S1}.

No input detection is done in prelude lines P1, P2, and P3. Line P4 is a function call to `copy`(R_1, R'_1), which executes instructions numbered P4.1 through P4.11 of `copy`. Input detection is only done at line P4.2, the first `jump-if-empty` instruction:

$$L_{\text{P4.2}} + A_{R_1} \longrightarrow L_{\text{P4.2}} + I_{R_1}.$$

It does not trigger a restart, but simply converts the active query molecule A_{R_1} to I_{R_1}. Similarly, for lines P5 and P6, we add the reactions

$$L_{\text{P5.2}} + A_{R_2} \longrightarrow L_{\text{P5.2}} + I_{R_2} \text{ and}$$
$$L_{\text{P6.2}} + A_{R_3} \longrightarrow L_{\text{P6.2}} + I_{R_3}.$$

– **Restart:** Restart happens a number of times that is at most the total input length $n_1 + n_2 + \ldots n_k$, since each input molecule is loaded into a register exactly once, generating one active query molecule. For $k = 3$, the registers R'_1, R'_2 and R'_3, as well as the registers $R_4, \ldots, R_r$ are flushed, and any outputs that have been released in solution are destroyed, assuming that the number of outputs released into the solution was tracked by some Y' register, as before. Then the program counter is set to line P4 of the simulator (leader molecule L_{P4}). Algorithm 3 shows the restart pseudocode.

Algorithm 3. Restart

S1: `flush`(R'_1)
S2: `flush`(R'_2)
S3: `flush`(R'_3)
S4: `flush`(R_4)
...
Sr: `flush`(R_r)
S($r+1$): `destroy-output`()
S($r+2$): `goto(P4)`

i: `destroy-output()`	i: `goto`(i.1)
	i.1: `jump-if-empty`(Y',i.5)
	i.2: `dec`(Y')
	i.3: `destroy`(Y)
	i.4: `goto`(i.1)
	i.5: `goto`($i+1$)

Correctness of Algorithm 2: Sequential-RM-psCRN. We claim that our register machine simulator without pre-loaded inputs stably computes the same function as the register machine, assuming that sufficiently many blank molecules B are

present to ensure that reactions with B as a reactant can always proceed. (We note that no "fairness" assumption regarding the order in which reactions happen is necessary to show stability, since stability is a "reachability" requirement.)

The first three instructions, lines P1, P2, and P3), simply create polymers R'_1, R'_2, and R'_3. The delicate part of the simulation lies in the next three instructions on lines P4, P5, and P6, which copy input registers R_1, R_2, and R_3 to R'_1, R'_2 and R'_3, respectively. For concreteness, consider the instruction `copy`(R_1, R'_1) in line P4. (The argument is the same for line P5, with R_2, R'_2 substituted for R_1, R'_1, and is also the same for line P6, with R_3, R'_3 substituted for R_1, R'_1.)

The R_1-push reaction used in input loading can cause a violation of our earlier invariant that upon completion of each instruction, the count of I_{R_1} equals the length of the (leader) R_1-polymer. Instead, we have that I_{R_1} is less than or equal to the length of the R_1-polymer. This can cause the `jump-if-empty` instruction numbered P4.2 in the `copy` function to stall, when there is no I_{R_1} and also no $\perp_{R_1}$ (since the R_1-polymer is not empty due to input loading). In this case, the input detection reaction (introduced above)

$$L_{\text{P4.2}} + A_{R_1} \longrightarrow L_{\text{P4.2}} + I_{R_1},$$

will convert A_{R_1} to I_{R_1}. This averts stalling, since `jump-if-empty` can proceed using I_{R_1}. The subsequent lines of the `copy`(R_1, R'_1) code can then proceed.

Once lines P4, P5, and P6 have completed, the correctness of the psCRN simulation of the register machine, using the copies R'_1, R'_2 and R'_3, is not affected by input loading. Input loading and input detection can also proceed. These are the only viable reactions from the "halting" state L_H, and so eventually (since the RM machine being simulated is a halting machine), on any sufficiently long computation path, all inputs must be loaded and detected. Input detection after the prelude phase produces a "missing" I_σ molecule and triggers a restart. The restart flushes all registers used by the simulator, and also, using the Y-polymer, destroys any outputs in solution. (Since restart is triggered only when the program counter is at a line of the main program, restart does not interrupt execution of the `release-output` function.) A new simulation is then started at line P4, ensuring that any inputs that have been loaded since the last detect are copied to the simulator's input registers R'_1, R'_2 and R'_3.

Once all inputs have been detected, the invariant is restored and the simulator proceeds correctly, producing the correct output. This correct output is never subsequently changed, and so the computation is stable.

Bottleneck Reactions. In our sequential psCRNs, both `inc`(σ) and `dec`(σ) contain bottleneck reactions, and so `copy`(σ, σ') has $\Theta(|\sigma|)$ bottleneck reactions. Thus the psCRN for Square has $\Theta(n^2)$ bottleneck reactions. In the next section we show how to compute a close variant of the Square function with fewer bottleneck reactions, using anonymous polymers rather than leader polymers.

4 Faster Computation of Square by Threaded psCRNs with Anonymous Polymers

To avoid the bottleneck reactions of our sequential psCRN, we enable the `inc`(σ) and `dec`(σ) instructions to operate on many functionally-identical "anonymous" σ-polymers, rather than just a single leader polymer. Here we describe how this can work, using the function $f(n) = n2^{\lfloor \lg n \rfloor}$ as an example, and focusing only on the pre-loaded input model. Error detection and correction can be layered on, in a manner similar to Sect. 3.

We start with a single Y-polymer of length n, and we wish to create a total of $2^{\lfloor \lg n \rfloor}$ Y-polymers, whose lengths sum to $n2^{\lfloor \lg n \rfloor}$. Algorithm 4 proceeds in $\lfloor \lg n \rfloor$ rounds (lines 6–9), doubling the number of Y's on each round. To keep track of the Y molecules, we introduce a distributed σ-counter data structure, and use it with $\sigma = Y$. The data structure consists of σ-polymers that we call *σ-thread-polymers*, plus a thread-polymer counter T_σ, which is a leader polymer whose length, $|T_\sigma|$, is the number of σ-thread-polymers. The *value* of this distributed counter is the total length of all σ-thread-polymers. We explain below how operations on this distributed counter work.

Algorithm 4. Threaded-$n2^{\lfloor logn \rfloor}$-psCRN.

```
 1: create-polymer(H)
 2: copy(X, H)
 3: create-distributed-counter(Y)
 4: add-thread-polymer(Y,1)
 5: copy(X, Y)
 6: halve(H)
 7: jump-if-empty(H,10)
 8:     double(Y)
 9:     goto(6)
10: halt
```

Algorithm 4 counts the number of rounds using a leader H-polymer, whose length is halved on each round. The `halve` function is fairly straightforward to implement, using the instructions and functions already introduced in Sect. 3.

i: `halve`(H)	i: `goto`(i.1)
	i.1: `create-polymer`(H')
	i.2: `copy`(H, H')
	i.3: `jump-if-empty`(H', i.9)
	i.4: `dec`(H')
	i.5: `dec`(H)
	i.6: `jump-if-empty`(H', i.9)
	i.7: `dec`(H')
	i.8: `goto`(i.3)
	i.9: `destroy-polymer`(H')
	i.10: `goto`($i + 1$)

The `double`(σ) function of Algorithm 4 is where we leverage our distributed Y-counter (with $\sigma = Y$). Recall that a distributed σ-counter data structure consists of a set of anonymous σ-polymers, which we call σ-thread-polymers, plus a thread-polymer counter T_σ, which is a leader polymer whose length is the number of σ-thread-polymers. The `double` function first creates two other distributed counters τ and τ' (lines i.1 and i.2), and gives each the same number of thread-polymers as σ, namely $|T_\sigma|$ thread-polymers (lines i.3 and i.4), all of which are empty. The heart of `double` (line i.5) transfers the contents of the distributed σ-counter to τ and τ', emptying and destroying all σ-thread-polymers in the process. It then creates double the original number of (empty) σ-thread-polymers (lines i.6 and i.7; note that the number of threads of τ is the original value of $|T_\sigma|$). It finally transfers the τ and τ' polymers back to σ (lines i.8 and i.9), thereby doubling σ.

i: double(σ)	i.1 `create-distributed-counter`(τ) i.2 `create-distributed-counter`(τ') i.3 `add-thread-polymers`(τ, T_σ) i.4 `add-thread-polymers`(τ', T_σ) i.5 `transfer`(σ, τ, τ') i.6 `add-thread-polymers`(σ, T_τ) i.7 `add-thread-polymers`(σ, $T_{\tau'}$) i.8 `transfer`(τ, σ) i.9 `transfer`(τ', σ) i.10 `destroy-distributed-counter`(τ) i.11 `destroy-distributed-counter`(τ') i.12 `goto`($i+1$)

Next are details of instructions used to create an empty distributed counter, and to add empty threads to the counter. Again, these are all straightforward sequential implementations (no threads), using leader polymers to keep track of counts.

i: `create-distributed-counter`(σ)
// Creates an empty counter
// with zero polymers
	i: `goto`(i.1)
	i.1 `create-polymer`(T_σ)
	i.2 `goto`($i+1$)

i `add-thread-polymers`(σ, T)
// Adds $\|T\|$ empty
// thread-polymers to the
// distributed σ-counter,
// where T is a counter
	i `goto`(i.1)
	i.1: `create-polymer`(Temp)
	i.2: `copy`(T,Temp)
	i.3: `jump-if-empty`(Temp, i.8)
	i.4: `dec`(Temp)
	i.5: `create-polymer`(σ)
	i.6: `inc`(T_σ)
	i.7: `goto`(i.3)
	i.8: `destroy-polymer`(Temp)
	i.9: `goto`($i+1$)

i `add-thread-polymer`(σ, 1)
// Adds one empty
// thread-polymer to the
// distributed σ-counter
	i `goto`(i.1)
	i.1: `create-polymer`(σ)
	i.2: `inc`(T_σ)
	i.3: `goto`($i+1$)

The `transfer` function transfers the value of a distributed σ-counter to two other distributed counters called τ and τ'. In line $i.2$ of `transfer`, function `create-threads` creates T_σ identical "thread" program counters, L_t. Once again this is straightforward, using a leader polymer to keep track of counts. All of the thread program counters execute the `thread-transfer` function in line $i.4$ of `transfer`, thereby reducing bottleneck reactions (details below). The "main" program counter, now at line $i.4$ of the `transfer` function, can detect when all threads have completed, because each decrements Thread-Count exactly once, and so Thread-Count has length zero exactly when all threads have completed. At that point, the main program counter progresses to line $i.5$, destroying the thread program counters using the `destroy-threads` function (not shown, but uses the `destroy` function to destroy each single thread).

i: `transfer`(σ, τ, τ')	i: `goto`(i.1)
// transfer σ to	$i.1$: `create-polymer`(Thread-Count)
// both τ and τ'	$i.2$: `create-threads`(T_σ, L_t)
	$i.3$: `copy`(T_σ, Thread-Count)
	$i.4$: `loop-until-empty`(Thread-Count, $i.5$)
	`thread-transfer`$(\sigma, \tau, \tau'$,Thread-Count)
	$i.5$: `destroy-threads`(L_t)
	$i.6$: `destroy-polymer`(Thread-Count)
	$i.7$: `goto`$(i+1)$

The function `transfer`(σ, τ), not shown but used in `double`, is the same as `transfer`(σ, τ, τ'), except the call to `thread-transfer` does not include τ' and the "`inc`(τ')" line is removed in the implementation of `thread-transfer`.

i: `create-threads`(T_σ, L_t)	i: `goto`(i.1)
// create T_σ thread	$i.1$: `create-polymer`(Temp)
// program counters, L_t	$i.2$: `copy`(T_σ, Temp)
	$i.3$: `jump-if-empty`(Temp, $i.7$)
	$i.4$ `dec`(Temp)
	$i.5$ `create`(L_t)
	$i.6$ `goto`($i.3$)
	$i.7$: `destroy-polymer`(Temp)
	$i.8$: `goto`$(i+1)$
i : `loop-until-empty`(σ, k)	$L_i + \perp_\sigma \longrightarrow L_k + \perp_\sigma$

Finally, we describe how threads work in `thread-transfer`. The `threadon`() function executes $|T_\sigma|$ times, one per copy of L_t, thereby creating $|T_\sigma|$ L_{t_1} program counters that execute computation "threads". Using the function `dec-until-destroy-polymer`, each thread repeatedly (zero or more times) decrements one of the σ-thread-polymers and then increments both τ and τ'. This continues until the thread finds an empty σ-thread-polymer, i.e., the stub $\perp_\sigma$, in which case it destroys the stub and moves to line $t.5$. The `dec`(σ) and `inc`(σ) functions of Sect. 3 work exactly as specified, even when applied to distributed counters. A key point is that the threads work "anonymously" with

the thread-polymers; it is not the case that each thread "owns" a single thread-polymer. Accordingly, one thread may do more work than another, but in the end all thread-polymers are empty.

A thread exits the `dec-until-destroy-polymer` loop by destroying exactly one σ-polymer. Since at the start of `thread-transfer` the number of σ-thread-polymers equals the number of thread program counters, all thread program counters eventually reach line $t.5$, and there are no σ-thread-polymers once all threads have reached line $t.5$ of the code. At line $t.5$, each thread decrements Thread-Count, and then stalls at line $t.6$. Moreover, once all threads have reached line $t.6$, polymer ThreadCount is empty. At this point, the program counter for `transfer` changes from line $i.4$ to line $i.5$, and all thread program counters are destroyed.

i: `thread-transfer` $(\sigma,\tau,\tau'$,Thread-Count)	i: `threadon`() $t.1$: `dec-until-destroy-polymer`(σ, $t.5$) $t.2$: `inc`(τ) $t.3$: `inc`(τ') $t.4$: `goto`($t.1$) $t.5$: `dec`(Thread-Count) $t.6$:

i: `threadon`():	$L_i + L_t \longrightarrow L_i + L_{t.1}$
i: `dec-until-destroy-polymer`(σ, k)	$L_i + I_\sigma \longrightarrow L_i^* + A_\sigma$ $A_\sigma + [\bot_\sigma \ldots \sigma] \rightleftharpoons \sigma + [\bot_\sigma \ldots]$ $L_i^* + \sigma \longrightarrow L_{i+1} + \mathrm{B}$ $L_i + \bot_\sigma \longrightarrow L_i^{**} + B$ $L_i^{**} + I_{T_\sigma} \longrightarrow L_i^{***} + A_{T_\sigma}$ $A_{T_\sigma} + [\bot_{T_\sigma} \ldots T_\sigma] \rightleftharpoons T_\sigma + [\bot_{T_\sigma} \ldots]$ $L_i^{***} + T_\sigma \longrightarrow L_k + \mathrm{B}$

Correctness. We claim that on any input $n \geq 0$, pre-loaded on a leader X-polymer, Algorithm 4: Threaded-$n2^{\lfloor \lg n \rfloor}$-psCRN eventually halts with the value of the distributed-Y-counter being $f(n) = n2^{\lfloor \log n \rfloor}$.

The algorithm creates and initializes H to be a polymer of length n (lines 1–2), and the Y-distributed-counter to have a single polymer-thread of length n (lines 3–5). When $n = 0$, H is empty, so from line 6 the algorithm jumps to line 10 and halts, with the value of Y being $f(0) = 0$ as claimed.

Suppose that $n > 0$. Reasoning about the `halve` function is straightforward, since it is fully sequential. We claim that in each round of the algorithm (lines 6–9), lines 7 and 8 complete successfully, with $|H|$ halving (that is, $|H| \to \lfloor |H|/2 \rfloor$) in line 7, and with both the value of Y and $|T_Y|$, the number of Y-thread-polymers, doubling in line 8. As a result, $|H| = 0$ after $\lfloor \lg n \rfloor$ rounds and the algorithm halts with value$(Y) = f(n)$.

Correctness of the `double` function is also straightforward to show, if we show that the `transfer`(σ, τ, τ') (and the `transfer`(σ, τ) variant) works correctly.

Line $i.4$ is the core of `transfer`. We show that line $i.4$ does complete, that is, Thread-Count does become empty, that execution of line $i.4$ increases the values of distributed counters τ and τ' by the value of σ (while leaving the number of τ- and τ'-thread-polymers unchanged), and also changes value(σ) and the number of σ-thread-polymers to 0.

The `loop-if-empty` instruction ensures that the main program counter must stay at line $i.4$ of function `transfer` until Thread-Count is empty. Meanwhile, this main program counter can also activate threads using the `threadon`() function, that is, change the thread program counters from L_t to $L_{t.1}$. From line $i.2$ of `transfer`, the number of such thread program counters is $|T_\sigma|$.

Each of these program counters independently executes `thread-transfer`. At line $t.1$, either (i) a `dec`(σ) is performed (first three reactions of `dec-until-destroy-polymer`), or (ii) a σ-polymer-thread is destroyed and the polymer-thread-count T_σ is decremented (last four reactions). In case (i), both τ and τ' are incremented (lines $t.2$ and $t.3$), and the thread goes back to the `dec-until-destroy-polymer` instruction. In case (ii), the thread moves to line $t.4$, decrements Thread-Count exactly once, and moves to line $t.5$.

Because the number of threads equals the value of Thread-Count at the start of the loop-until-empty (line $i.4$), and because the main program counter can't proceed beyond line $i.4$ of the `transfer` function until Thread-Count is zero, all threads must eventually be turned on each of these threads must reach line $t.4$ and must decrement Thread-Count. Only then can the main program counter proceed to line $i.5$ of `transfer`. This in turn means that each thread must destroy a σ-polymer-thread. Since the number of σ-polymer-threads, $|T_\sigma|$, equals Thread-Count, all threads are destroyed (and the T_σ-polymer is empty) upon completion of `thread-transfer`.

Bottleneck Reactions. In each round, the `halve`(σ) function decreases the length of the H-polymer by a factor of 2, starting from n initially. Each decrement or increment of the H-polymer includes a bottleneck reaction, so there are $\Theta(n)$ bottleneck reactions in total, over all rounds. The `double` function creates 2^l thread-polymers in round l, for a total of $\Theta(n)$ thread-polymers over all rounds. The `transfer` function creates 2^l threads in round l and similarly destroys 2^l threads, and copies a polymer of length 2^l, so again has $\Theta(n)$ bottleneck reactions over all rounds. The reactions in `thread-transfer` are not bottleneck reactions (except in round 1); we analyze these in the next section.

5 psCRN Time Complexity Analysis and Simulation

We follow the stochastic model of Soloveichik et al. [6] for well-mixed, closed systems with fixed volume V. We assume that all reactions have rate constant 1. When in configuration $\mathbf{c}$, the *propensity* of reaction $R : r + r' \to p + p'$ is $\mathbf{c}(r)\mathbf{c}(r')/V$ if $r \neq r'$, and is $\binom{\mathbf{c}(r)}{2}/V$ if $r = r'$. Let $\Delta(\mathbf{c})$ be the sum of all reaction

propensities, when in configuration $\mathbf{c}$. When a reaction occurs in configuration $\mathbf{c}$, the probability that it is reaction R is the propensity of R divided by $\Delta(\mathbf{c})$, and the expected time for a reaction is $1/\Delta(\mathbf{c})$. When the only applicable reaction is a bottleneck reaction, and the volume V is $\Theta(n^2)$, the expected time for this bottleneck reaction is $\Theta(n^2)$. Soloveichik et al. [6] consider CRNs without polymers, but the same stochastic model is used by Lakin et al. [12] and Qian et al. [5], where the reactants r or r' (as well as the products) may be polymers.

Expected time complexity of Algorithm 1: Sequential-n^2-psCRN. This psCRN has n rounds, with $\Theta(n)$ instructions per round; for example, the copy of length n in each round has n `inc` instructions. So the total number of instructions executed, over all rounds is $\Theta(n^2)$; moreover, there are $\Theta(n^2)$ `inc`(σ) instruction overall. The program's instructions execute sequentially, that is, the ith instruction completes before the $(i+1)$st instruction starts, so the total expected time is the sum of the expected times of the individual instructions. Each instruction involves a constant number of reactions. Some instructions involve bottleneck reactions; for example, the push reaction of the `inc` instruction is a bottleneck reaction. So an execution of the program involves $\Theta(n^2)$ bottleneck reactions. Each of these takes $\Theta(n^2)$ time, so the overall expected time is $\Theta(n^4)$.

Expected time complexity of Algorithm 4: Threaded-$n2^{\lfloor log n \rfloor}$-psCRN. We noted earlier that Algorithm 4 has $\Theta(n)$ non-threaded instructions, and in fact $\Theta(n)$ bottleneck instructions. These take expected time $\Theta(n^3)$ overall, since the time for each is $\Theta(V) = \Theta(n^2)$.

Now, consider the threaded function, `thread-transfer`. In round $l, 1 \leq l \leq \lfloor \lg n \rfloor$, `thread-transfer` has 2^l threads, and pushes $n2^l$ Y monomers on to 2^l anonymous Y-polymers. Since each Y-push reaction is independent and is equally likely to increment each of the 2^l Y-polymers, the expected number of molecules per polymer is n. Using a Chernoff tail bound, we can show that all polymers have length in the range $[n/2, 2n]$ with all but negligibly small probability. In what follows, we assume that this is the case.

During the first $\geq n/2$ of the `thread-transfer` decrements in round l, the count of each of the reactants is 2^l: one program counter per thread and 2^l polymers in total. So the expected time for these decrements is $\Theta(V/2^{2l})$. Pessimistically, if all of the decrements happen to the same polymer, whose length could be as little as $n/2$ by our assumption above, there are $2^l - 1$ polymers and threads available for the next decrements, $2^l - 2$ polymers and threads available for the next n decrements after that once a second polymer is depleted, and so on. So the total expected time is $O(Vn \sum_{j=1}^{2^l}(1/j^2)) = O(nV)$. Multiplying by $\lfloor \lg n \rfloor$, the number of rounds, and noting that $V = O(n^2)$, we have that the total expected time for the `thread-transfer` over all rounds is $O(n^3 \lg n)$.

Simulator. To test the correctness of our protocols, we developed a custom CRN simulator designed to support anonymous polymers, though we only show the results of our sequential protocol here in this paper. The simulator uses a slightly modified version of Gibson and Bruck's next reaction method [17], which itself is an extension of Gillespie's algorithm [18]. We redefine what a single "species"

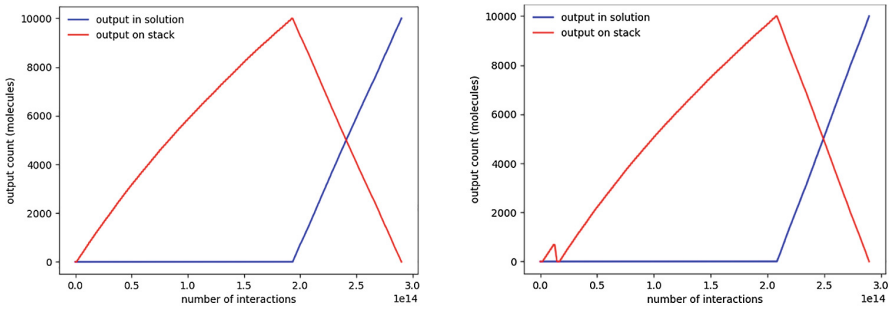

Fig. 1. (a) Simulation of $f(n) = n^2$ starting with pre-loaded input, $n = 100$. (b) Simulation of $f(n) = n^2$ with input detection and restart, $n = 100$. Each coloured line in the plots shows the count of the outputs as a function of the number of interactions, with the blue line being the count of output species Y finally released into solution, while the red line shows the size of the Y_{int} polymer. By interaction, we mean a collision of two molecular species in the system, which may or may not result in a reaction. (Color figure online)

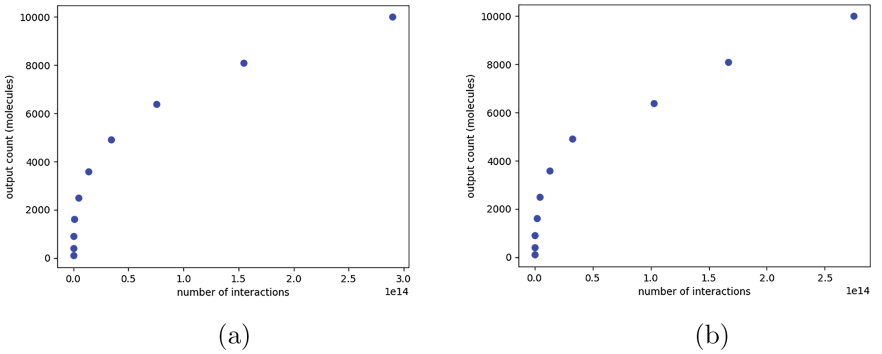

Fig. 2. (a) Simulations of $f(n) = n^2$ starting with pre-loaded input, covering a range of n. (b) Simulations of $f(n) = n^2$ with input detection and restart, covering a range of n. Each blue circle plots a single simulation, showing the final count of the output species Y against the number of interactions it took to complete. (Color figure online)

is from the algorithm's point of view, classifying all σ-polymers as one species, and track polymer lengths separately.

Interestingly, simulation of our stable, error-corrected sequential psCRN for Square usually takes little extra time compared to the committing, pre-loaded sequential psCRN (see both Figs. 1 and 2). This is because each of the n error detection steps, and subsequent restart, is expected to happen in $O(n^2)$ time, which is negligible compared to the time for the $\Theta(n^4)$ expected running time of the psCRN with fully loaded input.

6 Conclusions and Future Work

In this work, we've expanded the computing model of stochastic chemical reaction networks with polymers, by considering inputs that are represented as monomers in solution, as well as anonymous polymers that facilitate distributed data structures and threaded computation. We've shown that stable, error-free Turing-universal computation is possible in the monomer input model, by introducing an error-correction scheme that takes advantage of the ability to check for empty polymers. We've illustrated how programming with anonymous polymers can provide speed-ups, compared with using leader polymers only, and how leader polymers can be used for synchronization purposes by CRNs with anonymous polymers.

There are many interesting directions for future work. First, we have shown how to use anonymous polymers to get a speed-up for the Square problem, but we have not shown that such a speed-up is not possible without the use of anonymous polymers. Is it possible to show lower bounds on the time complexity of problems when only leader polymers are available? Or, could bottleneck reactions be reduced or avoided by a psCRN computing Square? Second, our faster psCRN for Square with anonymous polymers still uses leader polymers for synchronization. Is the speed-up possible even without the use of leader polymers? More generally, how can synchronization be achieved in leaderless psCRNs? Are there faster psCRNs, with or without leader polymers? It would be very interesting to know what problems have stable psCRNS that use no leaders, but can use anonymous polymers. Finally, it would be valuable to have more realistic models of reaction propensities for psCRN models.

References

1. Soloveichik, D., Cook, M., Winfree, E., Bruck, J.: Computation with finite stochastic chemical reaction networks. Nat. Comput. **7**(4), 615–633 (2008)
2. Chen, H.-L., Doty, D., Soloveichik, D.: Deterministic function computation with chemical reaction networks. Nat. Comput. **13**, 517–534 (2014)
3. Angluin, D., Aspnes, J., Eisenstat, D.: Stably computable predicates are semilinear. In: Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, New York, pp. 292–299. ACM Press (2006)
4. Cummings, R., Doty, D., Soloveichik, D.: Probability 1 computation with chemical reaction networks. Nat. Comput. **15**(2), 245–261 (2014)
5. Qian, L., Soloveichik, D., Winfree, E.: Efficient turing-universal computation with DNA polymers. In: Sakakibara, Y., Mi, Y. (eds.) DNA 2010. LNCS, vol. 6518, pp. 123–140. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18305-8_12
6. Soloveichik, D., Cook, M., Winfree, E., Bruck, J.: Computation with finite stochastic chemical reaction networks. Nat. Comput. **7**, 615–633 (2008)
7. Bennett, C.: Logical reversibility of computation. IBM J. Res. Dev. **17**(6), 525–532 (1973)
8. Bennett, C.: The thermodynamics of computation - a review. Int. J. Theor. Phys. **21**(12), 905–940 (1981)

9. Johnson, R., Winfree, E.: Verifying polymer reaction networks using bisimulation (2014)
10. Cardelli, L., Zavattaro, G.: Turing universality of the biochemical ground form. Math. Struct. Comput. Sci. **20**, 45–73 (2010)
11. Jiang, H., Riedel, M., Parhi, K.: Synchronous sequential computation with molecular reactions. In: Proceedings of the 48th Design Automation Conference, DAC 2011, New York, pp. 836–841. ACM (2011)
12. Lakin, M.R., Phillips, A.: Modelling, simulating and verifying turing-powerful strand displacement systems. In: Cardelli, L., Shih, W. (eds.) DNA 2011. LNCS, vol. 6937, pp. 130–144. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23638-9_12
13. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distrib. Comput. **18**, 235–253 (2006)
14. Chatzigiannakis, I., Michail, O., Nikolaou, S., Pavlogiannis, A., Spirakis, P.G.: Passively mobile communicating machines that use restricted space. In: Proceedings of the 7th ACM ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing, FOMC 2011, New York, pp. 6–15. ACM (2011)
15. Chen, H.-L., Cummings, R., Doty, D., Soloveichik, D.: Speed faults in computation by chemical reaction networks. In: Kuhn, F. (ed.) DISC 2014. LNCS, vol. 8784, pp. 16–30. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45174-8_2
16. Angluin, D., Aspnes, J., Eisenstat, D.: Fast computation by population protocols with a leader. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 61–75. Springer, Heidelberg (2006). https://doi.org/10.1007/11864219_5
17. Gibson, M.A., Bruck, J.: Efficient exact stochastic simulation of chemical systems with many species and many channels. J. Phys. Chem. A **104**(9), 1876–1889 (2000)
18. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. J. Phys. Chem. **81**(25), 2340–2361 (1977)