Fast Computation by Population Protocols With a Leader

Dana Angluin[1], James Aspnes[1⋆], and David Eisenstat[2]

[1] Yale University, Department of Computer Science
[2] Princeton University, Department of Computer Science

Abstract. Fast algorithms are presented for performing computations in a probabilistic population model. This is a variant of the standard population protocol model—in which finite-state agents interact in pairs under the control of an adversary scheduler—where all pairs are equally likely to be chosen for each interaction. It is shown that when a unique leader agent is provided in the initial population, the population can simulate a virtual register machine in which standard arithmetic operations like comparison, addition, subtraction, and multiplication and division by constants can be simulated in $O(n \log^4 n)$ interactions with high probability. Applications include a reduction of the cost of computing a semilinear predicate to $O(n \log^4 n)$ interactions from the previously best-known bound of $O(n^2 \log n)$ interactions and simulation of a LOGSPACE Turing machine using the same $O(n \log^4 n)$ interactions per step. These bounds on interactions translate into $O(\log^4 n)$ time per step in a natural parallel model in which each agent participates in an expected $\Theta(1)$ interactions per time unit. The central method is the extensive use of epidemics to propagate information from and to the leader, combined with an epidemic-based phase clock used to detect when these epidemics are likely to be complete.

1 Introduction

The **population protocol** model of Angluin *et al.* [3] consists of a population of finite-state agents that interact in pairs, where each interaction updates the state of both participants according to a transition function based on the participants' previous states and the goal is to have all agents eventually converge to a common output value that represents the result of the computation, typically a predicate on the initial state of the population. A population protocol that always converges to the correct output is said to perform **stable computation** and a predicate that can be so computed is called **stably computable**.

In the simplest version of the model, any pair of agents may interact, but which interaction occurs at each step is under the control of an adversary, subject to a fairness condition that essentially says that any continuously reachable global configuration is eventually reached. The class of stably computable predicates in this model is now very well understood: it consists precisely of the **semilinear predicates** (those predicates on counts of input agents definable in first-order **Presburger arithmetic** [23]), where

⋆ Supported in part by NSF grants CNS-0305258 and CNS-0435201.

semilinearity was shown to be sufficient in [3] and necessary in [5]. However, the fact that a protocol will eventually converge to the correct value of a semilinear predicate says little about how long such convergence will take.

Our fundamental measure of convergence is the total number of pairwise interactions until all agents have the correct output value, considered as a function of n, the number of agents in the population. We may also consider models in which reactions occur in parallel according to a Poisson process (as assumed in e.g. [17, 18]); this gives an equivalent distribution over sequences of reactions but suggests a measure of **time** based on assuming each each agent participates in an expected $\Theta(1)$ interactions per time unit. It is not hard to see that this time measure is asymptotically equal to the number of interactions divided by n.

To bound these measures, it is necessary to place further restrictions on the adversary: a merely fair adversary may wait an arbitrary number of interactions before it allows a particular important interaction to occur. In the present work, we consider the natural probabilistic model, proposed in [3], in which each interaction occurs between a pair of agents chosen uniformly at random. In this model, it was shown in [3] that any semilinear predicate can be computed in $\Theta(n^2 \log n)$ expected interactions using a protocol based on leader election in which the leader communicates the outcome by interacting with every other agent. Protocols were also given to simulate randomized LOGSPACE computations with polynomial slowdown, allowing an inverse polynomial probability of failure.

We give a new method for the design of probabilistic population protocols, based on controlled use of self-timed epidemics to disseminate control information rapidly through the population. This method organizes a population as an array of registers that can hold values linear in the size of the population. The simulated registers support the usual arithmetic operations, including addition, subtraction, multiplication and division by constants, and comparison, with implementations that complete with high probability in $O(n \log^4 n)$ interactions and polylogarithmic time per operation. As a consequence, any semilinear predicate can be computed without error by a probabilistic population protocol that converges in $O(n \log^4 n)$ interactions with high probability, and randomized LOGSPACE computation can be simulated with inverse polynomial error with only polylogarithmic slowdown. These bounds are optimal up to polylogarithmic factors, because $\Omega(n \log n)$ interactions are necessary to ensure that every agent has participated in at least one interaction with high probability.

However, in order to achieve these low running times, it is necessary to assume a leader in the form of some unique input agent. This is a reasonable assumption in sensor network models as a typical sensor network will have some small number of sensors that perform the specialized task of communicating with the user and we can appoint one of these as leader. Assuming the existence of a leader does not trivialize the problem; for example, any protocol that requires that the leader personally visit every agent in the population runs in expected number of interactions at least $\Omega(n^2 \log n)$.

If a leader is not provided, it is in principle possible to elect one; however, the best known expected bounds for leader election in a population protocol is still the $\Theta(n^2)$ interactions or $\Theta(n)$ time of a naive protocol in which candidate leaders drop out only on encountering other leaders. It is an open problem whether a leader can be elected

significantly faster. There must also be a way to reinitialize the simulation protocol once all but one of the candidates drops out. We discuss these issues further in Section 7.

In building a register machine from agents in a population protocol, we must solve many of the same problems as hardware designers building register machines from electrons. Thus the structure of the paper roughly follows the design of increasing layers of abstraction in a CPU. We present the underlying physics of the world—the population protocol model—in Section 2. Section 3 gives concentration bounds on the number of interactions to propagate the epidemics that take the place of electrical signals and describes the phase clock used to coordinate the virtual machine's instruction cycle. Section 4 describes the microcode level of our machine, showing how to implement operations that are convenient to implement but hard to program with. More traditional register machine operations are then built on top of these microcode operations in Section 5, culminating in a summary of our main construction in Theorem 2. Applications to simulating LOGSPACE Turing machines and computing semilinear predicates are described in Section 6. Some directions for future work are described in Section 7. Due to space limitations, most proofs are omitted from this extended abstract.

Many of our results are probabilistic, and our algorithms include tuning parameters that can be used to adjust the probability of success. For example, the algorithm that implements a given register machine program is designed to run for n^k instructions for some k, and the probability of failure for each instruction must be bounded by a suitable inverse polynomial in n. We say that a statement holds **with high probability** if for any constant c there is a setting of the tuning parameters that cause the statement to hold with probability at least $1 - n^{-c}$. The cost of achieving a larger value of c is a constant factor slowdown in the number of interactions (or time) used by the algorithms.

1.1 Related work

The population protocol model has been the subject of several recent papers. Diamadi and Fischer introduced a version of the probabilistic model to study the propagation of trust in a social network [15], and a related model of urn automata was explored in [2]. One motivation for the basic model studied in [3] was to understand the computational capabilities of populations of passively mobile sensors with very limited computational power. In the simplest form of the model, any agent may interact with any other, but variations of the model include limits on which pairs of agents may interact [1, 3, 4], various forms of one-way and delayed communication [6], and failures of agents [14]. The properties computed by population protocols have also been extended from predicates on the initial population to predicates on the underlying interaction graph [1], self-stabilizing behaviors [7], and stabilizing consensus [8].

Similar systems of pairwise interaction have previously been used to model the interaction of small molecules in solution [18, 19] and the propagation in a human population of rumors [12] or epidemics of infectious disease [10]. Epidemic algorithms have also been used previously to perform multicast operations, e.g. by Birman *et al.* [11].

The notion of a "phase clock" as used in our protocol is common in the self-stabilizing literature, e.g. [20]. There is a substantial stream of research on building self-stabilizing synchronized clocks dating back to to the work of Arora *et al.* [9]. Recent

work such as [16] shows that it is possible to perform self-stabilizing clock synchronization in traditional distributed systems even with a constant fraction of Byzantine faults; however, the resulting algorithms require more network structure and computational capacity at each agent that is available in a population protocol. An intriguing protocol of Daliot *et al.* [13] constructs a protocol for the closely-related problem of pulse synchronization inspired directly by biological models. Though this protocol also exceeds the finite-state limits of population protocols, it may be possible to construct a useful phase clock for our model by adapting similar techniques.

2 Model

In this paper we consider only the complete all-pairs interaction graph, so we can simplify the general definition of a probabilistic population protocol as follows. A **population protocol** consists of a finite set Q of states, of which a nonempty subset X are the initial states (thought of as inputs), a deterministic transition function $(a, b) \mapsto (a', b')$ that maps ordered pairs of states to ordered pairs of states, and an output function that maps states to an output alphabet Y. The **population** consists of agents numbered 1 through n; agent identities are not visible to the agents themselves, but facilitate the description of the model. A **configuration** C is a map from the population to states, giving the current state of every agent. An **input configuration** is a map from the population to X, representing an input consisting of a multiset of elements of X. C can reach C' in one interaction, denoted $C \to C'$, if there exist distinct agents i and j such that $C(i) = a$, $C(j) = b$, the transition function specifies $(a, b) \mapsto (a', b')$ and $C'(i) = a'$, $C'(j) = b'$ and $C'(k) = C(k)$ for all k other than i and j. In this interaction, i is the **initiator** and j is the **responder** – this asymmetry of roles is an assumption of the model [4].

An **execution** is a sequence $C_1, C_2, \ldots$ of configurations such that for each i, $C_i \to C_{i+1}$. An execution **converges** to an output $y \in Y$, if there exists an i such that for every $j \geq i$, the output function applied to every state occurring in C_j is y. In general, individual agents may not know when convergence to a common output has been reached, and protocols are generally designed not to halt. An execution is **fair** if for any C_i and C_j such that $C_i \to C_j$ and C_i occurs infinitely often in the execution, C_j also occurs infinitely often in the execution. A protocol **stably computes** a predicate P on multisets of elements of X if for any input configuration C, every fair execution of the protocol starting with C converges to 1 if P is true on the multiset of inputs represented by C, and converges to 0 otherwise. Note that a fixed protocol must be able to handle populations of arbitrary finite size – there is no dependence of the number of states on n, the population size.

For a **probabilistic population protocol**, we stipulate a particular probability distribution over executions from a given configuration C_1 as follows. We generate C_{k+1} from C_k by drawing an ordered pair (i, j) of agents independently and uniformly, applying the transition function to $(C_k(i), C_k(j))$, and updating the states of i and j accordingly to obtain C_{k+1}. (Note that an execution generated this way will be fair with probability 1.) In the probabilistic model we consider both the random variable of the

number of interactions until convergence and the probabilities of various error conditions in our algorithms.

3 Tools

Here we give the basic tools used to construct our virtual machine. These consist of concentration bounds on the number of interactions needed to spread epidemics through the population (Section 3.1), which are then used to construct a phase clock that controls the machine's instruction cycle (Section 3.2). Basic protocols for duplication (Section 3.3), cancellation (Section 3.4), and probing (Section 3.5) are then defined and analyzed.

3.1 Epidemics

By a **one-way epidemic** we denote the population protocol with state space $\{0, 1\}$ and transition rule $(x, y) \mapsto (x, \max(x, y))$. Interpreting 0 as "susceptible" and 1 as "infected," this protocol corresponds to a simple epidemic in which transmission of the infection occurs if and only if the initiator is infected and the responder is susceptible. In the full paper, we show, using a reduction to coupon collector and sharp concentration results of [21], that the number of interactions for the epidemic to finish (that is, infect every agent) is $\Theta(n \log n)$ with high probability.

It will be useful to have a slightly more general lemma that bounds the time to infect the first k susceptible agents. Because of the high variance associated with filling the last few bins in the coupon collection problem, we consider only $k \geq n^\epsilon$ for $\epsilon > 0$.

Lemma 1. *Let $T(k)$ be number of interactions before a one-way epidemic starting with a single infected agent infects k agents. For any fixed $\epsilon > 0$ and $c > 0$, there exist positive constants c_1 and c_2 such that for sufficiently large n and any $k > n^\epsilon$, $c_1 n \ln k \leq T(k) \leq c_2 n \ln k$ with probability at least $1 - n^{-c}$.*

3.2 The phase clock

The core of our construction is a **phase clock** that allows a leader to determine when an epidemic or sequence of triggered epidemics is likely to have finished. In essence, the phase clock allows a finite-state leader to count off $\Theta(n \log n)$ total interactions with high probability; by adjusting the constants in the clock, the resulting count is enough to outlast the $c_2 n \ln n$ interactions needed to complete an epidemic by Lemma 1. Like physical clocks, the phase clock is based on a readily-available natural phenomenon with the right duration constant. A good choice for this natural phenomenon, in a probabilistic population protocol, turns out to be itself the spread of an epidemic. Like the one-way epidemic of Section 3.1, the phase clock requires only one-way communication.

Here is the protocol: each agent has a state in the range $0 \ldots m-1$ for some constant m that indicates which phase of the clock it is infected with. (The value of m will be chosen independent of n, but depending on c, where $1 - n^{-c}$ is the desired success probability.) Up to a point, later phases overwrite earlier phases: a responder in phase i

will adopt the phase of any initiator in phases $i+1 \bmod m$ through $i+m/2 \bmod m$, but will ignore initiators in other phases. This behavior completely describes the transition function for non-leader responders.

New phases are triggered by a unique leader agent. When the leader encounters an initiator with its own phase, it spontaneously moves to the next phase. The leader ignores interactions with initiators in other phases. The initial configuration of the phase clock has the leader in phase 0 and all other agents in phase $m-1$. A **round** consists of m phases. A new round starts when the leader enters phase 0.

The normal operation of the phase clock has all the agents in a very few adjacent states, with the leader in the foremost one. When that state becomes populated enough for the leader to encounter another agent in that state, the leader moves on to the next state (modulo m) and the followers are pulled along. Successive rounds should be $\Theta(n \log n)$ interactions apart with high probability; the lower bound allows messages sent epidemically to reach the whole population, and the upper bound is essential for the overall efficiency of our algorithms.

Analysis We wish to show that for appropriate constants c and m, any epidemic (running in parallel with the phase clock) that starts in phase i completes by the next occurrence of phase $(i+c) \mod m$ with high probability. To simplify the argument, we first consider an infinite-state version of the phase clock with state space $\mathbb{Z} \times \{\text{leader}, \text{follower}\}$ and transition rules

$$\begin{aligned}(x, b), (y, \text{follower}) &\mapsto (x, b), (\max(x, y), \text{follower}) \\ (x, b), (x, \text{leader}) &\mapsto (x, b), (x+1, \text{leader}) \\ (x, b), (y, \text{leader}) &\mapsto (x, b), (y, \text{leader}) \quad [y \neq x]\end{aligned}$$

We assume the initial configuration (at interaction 0) has the leader in state 0 and each follower in state -1. This infinite-state protocol has the useful invariant that every agent has a phase less than or equal to that of the leader. We define phase i as starting when the leader agent first adopts phase i. This result bounds the probability that a phase "ends too early" by $n^{-1/2}$.

Lemma 2. *Let phase i start at interaction t. Then there is a constant a such that for sufficiently large n, phase $i+1$ starts before interaction $t + an \ln n$ with probability at most $n^{-1/2}$.*

Observing that several phases must "end too early" in order for a round to "end too early" allows us to go from a failure probability of $n^{-1/2}$ for a phase to n^{-c} for a round.

Corollary 1. *Let phase i start at interaction t. Then for any $c > 0$ and $d > 0$, there is a constant k such that for sufficiently large n, phase $i+k$ starts before $t + dn \ln n$ interactions with probability at most n^{-c}.*

The following theorem gives probabilistic guarantees for a polynomial number of rounds of the phase clock. In the proof the probability of failure due to a "straggler" (agent so far behind that it appears to be ahead modulo m) must be also be appropriately bounded, to ensure that m may be a constant independent of n.

Theorem 1. *For any fixed $c, d > 0$, there exists a constant m such that, for all sufficiently large n, the finite-state phase clock with parameter m, starting from an initial state consisting of one leader in phase 0 and $n-1$ followers in phase $m-1$, completes n^c rounds of m phases each, where the minimum number of interactions in any of the n^c rounds is at least $dn \ln n$ with probability at least $1 - n^{-c}$.*

Proof. The essential idea is to apply Corollary 1 twice: once to show that with high probability the number of interactions between phase $i+1$ and phase $i + m/2$ is long enough for any old phase-i agents to be eaten up (thus avoiding any problems with wrap-around), and once to show the lower bound on the length of a round.

To show that no agent is left behind, consider, in the infinite-state protocol, the fate of agents in phase i or lower once at least one agent in phase $i+1$ or higher exists. If we map all phases i or lower to 0 and all phases $i+1$ or higher to 1, then encounters between agents have the same effect after the mapping as in a one-way epidemic. By Lemma 1, there is a constant c_2 such that all n agents are infected by interaction $c_2 n \ln n$ with probability at least $1 - n^{-3c}$. By Corollary 1, there is a constant k_1 such that phase $i + k_1 + 1$ starts at least $c_2 n \ln n$ interactions after phase $i+1$ with probability at least $1 - n^{-3c}$. Setting $m > 2(k_1 + 1)$ then ensures that all phase i (or lower) agents have updated their phase before phase $i + m/2$ with probability at least $1 - 2n^{-3c}$. If we sum the probability of failure over all mn^c phases in the first n^c rounds, we get a probability of at most $2mn^{-2c}$ that some phase i agent survives long enough to cause trouble.

Assuming that no such trouble occurs, we can simulate the finite-state phase clock by mapping the phases of the infinite-state phase clock mod m. Now by Corollary 1 there is a constant k_2 such that the number of interactions to complete k_2 consecutive phases is at least $dn \ln n$ with probability at least $1 - n^{-3c}$. Setting $m \geq k_2$ thus gives that all n^c rounds take at least $dn \ln n$ interactions with probability at least $1 - n^c n^{-3c} = 1 - n^{-2c}$. Thus the total probability of failure is bounded by $2mn^{-2c} + n^{-2c} < n^{-c}$ for sufficiently large n as claimed.

3.3 Duplication

A **duplication** protocol has state space $\{(1,1), (0,1), (0,0)\}$ and transition rules:

$$(1,1), (0,0) \mapsto (0,1), (0,1)$$
$$(0,0), (1,1) \mapsto (0,1), (0,1)$$

with all other encounters having no effect.

When run to convergence, a duplication protocol starting with a "active" agents in state $(1,1)$ and the rest in the null state $(0,0)$ converges to $2a$ "inactive" agents in state $(0,1)$, provided $2a$ is less than n; otherwise it converges to a population of mixed active and inactive agents with no unrecruited agents left in the null state. The invariant is that the total number of 1 tokens is preserved while eliminating as many double-token agents as possible. We do not consider agents in a $(1,0)$ state as they can be converted to $(0,1)$ immediately at the start of the protocol.

When the initial number of active agents a is close to $n/2$, duplication may take as much as $\Theta(n^2)$ interactions to converge, as the last few active agents wait to encounter the last few null agents. But for smaller values of a the protocol converges more quickly.

Lemma 3. *Let $2a + b \leq n/2$. The probability that a duplication protocol starting with a active agents and b inactive agents, has not converged after $(2c+1)n \ln n$ interactions is at most n^{-c}.*

3.4 Cancellation

A **cancellation** protocol has states $\{(0,0),(1,0),(0,1)\}$ and transition rules:

$$(1,0),(0,1) \mapsto (0,0),(0,0)$$
$$(0,1),(1,0) \mapsto (0,0),(0,0)$$

It maintains the invariant that the number of 1 tokens in the left-hand position minus the number of 1 tokens in the right-hand position is fixed. It converges when only $(1,0)$ and $(0,0)$ or only $(0,1)$ and $(0,0)$ agents remain. We assume that there are no $(1,1)$ agents as these can be converted to $(0,0)$ agents at the start of the protocol. We refer to agents in state $(1,0)$ or $(0,1)$ as nonzero agents.

As with duplication, the number of interactions to converge when $(1,0)$ and $(0,1)$ are nearly equally balanced can be as many as $\Theta(n^2)$, since we must wait in the end for the last few survivors to find each other. This is too slow to use cancellation to implement subtraction directly. Instead, we will use cancellation for inequality testing, using duplication to ensure that there is a large enough majority of one value or the other to ensure fast convergence. We will use the following fact.

Lemma 4. *Starting from any initial configuration, with probability at least $1 - n^{-c}$, after $4(c+1)n \ln n$ interactions a cancellation protocol has either converged or has at most $n/8$ of each type of nonzero agent.*

3.5 Probing

A **probing** protocol is used to detect if any agents satisfying a given predicate exist. It uses three states (in addition to any state tested by the predicate) and has transition rules

$$(x,y) \mapsto (x, \max(x,y))$$

when the responder does not satisfy the predicate and

$$(0,y) \mapsto (0,y)$$
$$(x,y) \mapsto (x,2) \quad [x > 0]$$

when the responder does. Note that this is a one-way protocol.

To initiate a probe, a leader starts in state 1; this state spreads through an initial population of state 0 agents as in a one-way epidemic and triggers the epidemic spread of state 2 if it reaches an agent that satisfies the predicate.

Lemma 5. *For any $c > 0$, there is a constant d such that for sufficiently large n, with probability at least $1 - n^{-c}$ it is the case that after $dn \ln n$ interactions in the probing protocol either (a) no agent satisfies the predicate and every agent is in state 1, or (b) some agent satisfies the predicate and every agent is in state 2.*

4 Computation by epidemic: the microcode level

In this section, we describe how to construct an abstract register machine on top of a population protocol. This machine has a constant number of registers each capable of holding integer values in the range 0 to n, and supports the usual arithmetic operations on these registers, including addition, subtraction, multiplication and division by constants, inequality tests, and so forth. Each of these operations takes at most a polylogarithmic number of basic instruction cycles, where an instruction cycle takes $\Theta(n \log n)$ interactions or $\Theta(\log n)$ time.

The simulation is probabilistic; there is an inverse polynomial probability of error for each operation, on which the exponent can be made arbitrarily large at the cost of increasing the constant factor in the running time.

The value of each register is distributed across the population in unary. For each register A, every member i of the population maintains one bit A_i and the current value of A is simply $\sum_i A_i$. Thus the finite state of each agent can be thought of as a finite set of finite-valued control variables, and one boolean variable for each of a finite set of registers. Recall that the identities of agents are invisible to the agents themselves, and are used to facilitate description of the model.

We assume there is a leader agent that organizes the computation; part of the leader's state stores the finite-state control for the register machine. We make a distinction between the "microcode layer" of the machine, which uses the basic mechanisms of Section 3, and the "machine code" layer, which provides familiar arithmetic operations.

At the microcode layer, we implement a basic instruction cycle in which the leader broadcasts an instruction to all agents using an epidemic. The agents then carry out this instruction until stopped by a second broadcast from the leader. This process repeats until the computation terminates.

To track the current instruction, each agent (including the leader) has a **current instruction register** in addition to its other state. These instructions are tagged with a **round number** in the range $0, 1, 2$, where round i instructions are overwritten by round $i + 1 \pmod 3$ instructions.

The instructions and their effects are given in Table 1. Most take registers as arguments. We also allow any occurrence of a register to be replaced by its negation, in which case the operation applies to those agents in which the appropriate bit is not set. For example, SET($\neg A$) resets A_i, PROBE($\neg A$) tests for agents in which A_i is not set, COPY($\neg A, B$) sets B_i to the negation of A_i, and so forth.

To interpret the table entries: when an agent changes its current instruction register to SET(A), it sets its boolean variable for register A to 1 and waits for the next instruction. Similarly, when it changes its current instruction register to COPY(A, B), then the agent sets its boolean variable for register B to the value of its boolean variable for

Instruction	Effect on state of agent i
NOOP	No effect.
SET(A)	Set $A_i = 1$.
COPY(A, B)	Copy A_i to B_i
DUP(A, B)	Run duplication protocol on state (A_i, B_i).
CANCEL(A, B)	Run cancellation protocol on state (A_i, B_i).
PROBE(A)	Run probe protocol with predicate $A_i = 1$.

Table 1. Instructions at the microcode level.

register A. When its current instruction becomes DUP(A, B), then the agent begins running the duplication protocol (Section 3.3) on the ordered pair of its boolean variables for registers A and B. (In the case of $(1, 0)$, it immediately exchanges them to $(0, 1)$, and in the cases of $(1, 1)$ and $(0, 0)$, it participates in the duplication protocol when it interacts with other agents with current instruction DUP(A, B), until either its pair becomes inactive or a new instruction supersedes the current one.) CANCEL(A, B) and PROBE(A) are handled analogously, where the predicate probed is whether the agent's boolean variable for register A is 1. We omit describing the underlying transitions as the details are tedious.

When the leader updates its own current instruction register, the new value spreads to all other agents in $\Theta(n \log n)$ interactions with high probability (Lemma 1). The NOOP, SET, and COPY operations take effect immediately, so no additional interactions are required. The PROBE operation may require waiting for a second triggered epidemic, but the total interactions are still bounded by $O(n \log n)$ with high probability (by Lemma 5). Only the DUP and CANCEL operations may take longer to converge. Because subsequent operations overwrite each agent's current instruction register, issuing a new operation has the effect of cutting these operations off early. But if this new operation is issued $\Omega(n \log n)$ interactions later, the DUP operation converges with high probability unless it must recruit more than half the agents (Lemma 3), and the CANCEL operation either converges or leaves at most $n/4$ uncanceled values (Lemma 4). Note that for either operation, which outcome occurred can be detected with COPY and PROBE operations.

Thus, the leader waits for $\Omega(n \log n)$ interactions between issuing successive instructions, where the constant is chosen based on the desired error bound. But this can be done using a phase clock with appropriate parameter (Theorem 1): if it is large enough that both the probability that an operation completes too late and the probability that some phase clock triggers to early is $o(n^{-2c})$ per operation, then the total probability that any of n^c operations fails is $o(n^{-c})$.

5 Computation by epidemic: higher-level operations

The operations of the previous section are not very convenient for programming. In this section, we describe how to implement more traditional register operations.

These can be divided into two groups: those that require a constant number of microcode instructions, and those that are implemented using loops. The first group,

Operation	Effect	Implementation	Notes
Constant 0	$A \leftarrow 0$	SET($\neg A$)	
Constant 1	$A \leftarrow 1$	SET($\neg A$) $A_{\text{leader}} \leftarrow 1$	
Assignment	$A \leftarrow B$	COPY(B, A)	
Addition	$A \leftarrow A + B$	COPY(B, X) DUP(X, A) PROBE(X)	May fail with $X \neq 0$ if $A + B > n/2$.
Multiplication	$A \leftarrow kB$	Use repeated addition.	$k = O(1)$
Zero test	$A \neq 0$?	PROBE(A)	

Table 2. Simple high-level operations and their implementations. Register X is an auxiliary register.

shown in Table 2, includes assignment, addition, multiplication by a constant, and zero tests. The second group includes comparison (testing for $A < B$, $A = B$, or $A > B$), subtraction, and division by a constant (including obtaining the remainder). These operations are described in more detail below.

Comparison For comparison, it is tempting just to apply CANCEL and see what tokens survive. But if the two registers A and B being compared are close in value, then CANCEL may take $\Theta(n^2)$ interactions to converge. Instead, we apply up to $2 \lg n$ rounds of cancellation, alternating with duplication steps that double the discrepancy between A and B. If $A > B$ or $B > A$, the difference soon becomes large enough that all of the minority tokens are eliminated. The case where $A = B$ is detected by failure to converge, using a counter variable C that doubles every other round.

The algorithm is given in Figure 1. It uses registers A', B', and C plus a bit r to detect even-numbered rounds.

Lemma 6. *Algorithm 1 returns the correct answer with high probability after executing at most $O(\log n)$ microcode operations.*

Subtraction Subtraction is the inverse of addition, and addition is a monotone operation. It follows that we can implement subtraction using binary search. Our rather rococo algorithm for computing $C \leftarrow A - B$, given in Figure 2 repeatedly looks for the largest power of two that can be added to the candidate difference C without making the sum of the difference C and the subtrahend B greater than the minuend A. It obtains one more 1 bit of the difference for each iteration.

The algorithm assumes $A \geq B$. An initial cancellation step is used to handle particularly large inputs. This allows the algorithm to work even when A lies outside the safe range of the addition operation.

The algorithm uses several auxiliary registers to keep track of the power of two to add to C (this is the D register) and to perform various implicit sums and tests (as in computing $B' + C + D + D$).

Lemma 7. *When $A \geq B$, Algorithm 2 computes $C \leftarrow A - B$ with high probability in $O(\log^3 n)$ microcode operations.*

```
 1: A' ← A.
 2: B' ← B.
 3: C ← 1.
 4: r ← 0.
 5: while true do
 6:    CANCEL(A', B').
 7:    if A' = 0 and B' = 0 then
 8:       return A = B.
 9:    else if A' = 0 then
10:       return A < B.
11:    else if B' = 0 then
12:       return A > B.
13:    end if
14:    r ← 1 − r.
15:    if r = 0 then
16:       C ← C + C.
17:       if addition failed then
18:          return A = B.
19:       end if
20:    end if
21:    A' ← A' + A'.
22:    B' ← B' + B'.
23: end while
```

Fig. 1. Comparison algorithm.

```
 1: A' ← A.
 2: B' ← B.
 3: CANCEL(A', B').
 4: if B' = 0 then
 5:    C ← A.
 6:    return.
 7: end if
 8: C ← 0.
 9: while A' ≠ B' + C do
10:    D ← 1.
11:    while A' ≥ B' + C + D + D do
12:       D ← D + D.
13:    end while
14:    C ← C + D.
15: end while
```

Fig. 2. Subtraction algorithm.

Division Division of A by a constant k is analogous to subtraction; we set $A' \leftarrow A$ and $B \leftarrow 0$ and repeatedly seek the largest power of two D such that kD can be successfully computed (i.e., does not cause addition to overflow) and $kD \leq A'$. We then subtract kD from A' and add D to B.

The protocol terminates when $A' < k$, i.e. when no value of D works. At this point B holds the quotient $\lfloor A/k \rfloor$ and A' the remainder $A \bmod k$. Since each iteration adds one bit to the quotient, there are at most $O(\lg n)$ iterations of the outer loop, for a total cost of $O(\lg^4 n)$ microcode operations (since each outer loop iteration requires one subtraction operation).

One curious property of this protocol is that the leader does not learn the value of the remainder, even though it is small enough to fit in its limited memory. If it is important for the leader to learn the remainder, it can do so using k addition and comparison operations, by successively testing the remainder A' for equality with the values $0, 1, 1+1, 1+1+1, \ldots, k$. The cost of this test is dominated by the cost of the division algorithm.

Other operations Multiplication and division by constants give us the ability to extract individual bits of a register value A. This is sufficient to implement basic operations like $A \leftarrow B \cdot C$, $A \leftarrow \lfloor B/C \rfloor$ in polylogarithmic time using standard bitwise algorithms.

Summary Combining preceding results gives:

Theorem 2. *A probabilistic population can simulate steps of a virtual machine with a constant number of registers holding integer values in the range* 0 *to* n*, where each step consists of (a) assigning a constant* 0 *or* 1 *value to a register; (b) assigning the value of one register to another; (c) adding the value of one register to another, provided the total does not exceed* $n/2$*; (d) multiplying a register by a constant, provided the result does not exceed* $n/2$*; (e) testing if a register is equal to zero; (f) comparing the values of two registers; (g) subtracting the values of two registers; or (h) dividing the value of a register by a constant and computing the remainder. The probability that for any single operation the simulation fails or takes more than* $O(n \log^4 n)$ *interactions can be made* $O(n^{-c})$ *for any fixed* c*.*

6 Applications

Simulating RL In [3], it was shown that a probabilistic population protocol with a leader could simulate a randomized LOGSPACE Turing machine with a constant number of read-only unary input tapes with polynomial slowdown. The basic technique was to use the standard reduction of Minsky [22] of a Turing machine to a counter machine, in which a Turing machine tape is first split into two stacks and then each stack is represented as a base-b number stored in unary. Because the construction in [3] could only increment or decrement counters, each movement of the Turing machine head required decrementing a counter to zero in order to implement division or multiplication. Using Theorem 2, we can perform division and multiplication in $O(n \log^4 n)$ interactions, which thus gives the number of interactions for a single Turing machine step. If we treat this quantity as $O(\log^4 n)$ time, we get a simulation with polylogarithmic slowdown.

Theorem 3. *For any fixed* $c > 0$*, there is a constant* d *such that a probabilistic population protocol on a complete graph with a leader that can simulate* n^c *steps of a randomized LOGSPACE Turing machine with a constant number of read-only unary input tapes using* $d \log^4 n$ *time per step with a probability of failure bounded by* n^{-c}*.*

Protocols for semilinear predicates From [3] we have that it is sufficient to be able to compute congruence modulo k, $+$, and $<$ to compute any semilinear predicate. From Theorem 2 we have that all of these operations can be computed with a leader in $O(n \log^4 n)$ interactions with high probability. The final stage of broadcasting the result to all agents can also be performed in $O(n \log n)$ interactions with high probability using an epidemic.

However, there is some chance of never converging to the correct answer if the protocol fails. To eliminate this possibility, we construct an optimistic hybrid protocol in which the fast but potentially inaccurate $O(n \log^4 n)$-interaction protocol is supplemented by an $O(n^2)$ leaderless protocol, with the leader choosing (in case of disagreement) to switch its output from that of the fast protocol to that of the slow protocol when it is likely the slow protocol has finished. The resulting hybrid protocol converges to the correct answer in all executions while still converging in $O(n \log^4 n)$ interactions in expectation and with high probability.

Theorem 4. *For any semilinear predicate P, and for any $c > 0$, there is a probabilistic population protocol on a complete graph with a leader to compute P without error that converges in $O(n \log^4 n)$ interactions with probability at least $1 - n^{-c}$ and in expectation.*

7 Open problems

For most of the paper, we have assumed that a unique leader agent is provided in the initial input. The most pressing open problem is whether this assumption can be eliminated without drastically raising the cost of our protocols.

One problem is the question of whether we can efficiently restart the phase clock after completing an initial leader election phase. A proof of possibility can be obtained by observing that the leader can shut off all other agents one at a time in $O(n^2 \log n)$ interactions, and then restart them in the same number of interactions; however, the leader may have to wait an additional large polynomial time to be confident that it has in fact reached all agents. We believe, based on preliminary simulation results, that a modified version of our phase clock can be restarted much more efficiently by a newly-elected leader. This would allow us to use our LOGSPACE simulator after an initial $O(n^2)$-interaction leader election stage. But more work is still needed.

Even better would be a phase clock that required no leader at all. This would allow every agent to independently simulate the single leader, eliminating both any initial leader election stage and the need to disseminate instructions. Whether such a leaderless phase clock is possible is not clear.

It would be interesting to explore refinements of the underlying assumption that pairs are drawn uniformly at random to interact, for example, to reflect the physical effects of spatial dispersion of the agents.

References

1. Dana Angluin, James Aspnes, Melody Chan, Michael J. Fischer, Hong Jiang, and René Peralta. Stably computable properties of network graphs. In Viktor K. Prasanna, Sitharama Iyengar, Paul Spirakis, and Matt Welsh, editors, *Distributed Computing in Sensor Systems: First IEEE International Conference, DCOSS 2005, Marina del Rey, CA, USA, June/July, 2005, Proceedings*, volume 3560 of *Lecture Notes in Computer Science*, pages 63–74. Springer-Verlag, June 2005.
2. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Urn automata. Technical Report YALEU/DCS/TR-1280, Yale University Department of Computer Science, November 2003.
3. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *PODC '04: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, pages 290–299. ACM Press, 2004.
4. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, March 2006.

5. Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. To appear, PODC 2006, July 2006.
6. Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. On the power of anonymous one-way communication. In *Ninth International Conference on Principles of Distributed Systems*, pages 307–318, December 2005.
7. Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing population protocols. In *Ninth International Conference on Principles of Distributed Systems*, pages 79–90, December 2005.
8. Dana Angluin, Michael J. Fischer, and Hong Jiang. Stabilizing consensus in mobile networks. To appear in Proc. International Conference on Distributed Computing in Sensor Systems (DCOSS06), June 2006.
9. Anish Arora, Shlomi Dolev, and Mohamed G. Gouda. Maintaining digital clocks in step. In Sam Toueg, Paul G. Spirakis, and Lefteris M. Kirousis, editors, *Distributed Algorithms, 5th International Workshop*, volume 579 of *Lecture Notes in Computer Science*, pages 71–79, Delphi, Greece, 1991. Springer-Verlag.
10. Norman T. J. Bailey. *The Mathematical Theory of Infectious Diseases, Second Edition*. Charles Griffin & Co., London and High Wycombe, 1975.
11. Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.
12. D. J. Daley and D. G. Kendall. Stochastic rumours. *Journal of the Institute of Mathematics and its Applications*, 1:42–55, 1965.
13. Ariel Daliot, Danny Dolev, and Hanna Parnas. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems*, volume 2704 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2003.
14. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Eric Ruppert. When birds die: Making population protocols fault-tolerant. To appear in Proc. International Conference on Distributed Computing in Sensor Systems (DCOSS06), June 2006.
15. Zoë Diamadi and Michael J. Fischer. A simple game for the study of trust in distributed systems. *Wuhan University Journal of Natural Sciences*, 6(1–2):72–82, March 2001. Also appears as Yale Technical Report TR–1207, January 2001.
16. Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004.
17. Michael A. Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *Journal of Physical Chemistry A*, 104:1876–1880, 2000.
18. Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.
19. Daniel T. Gillespie. A rigorous derivation of the chemical master equation. *Physica A*, 188:404–425, 1992.
20. Ted Herman. Phase clocks for transient fault repair. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1048–1057, 2000.
21. A. P. Kamath, R. Motwani, K. Palem, and P. Spirakis. Tail bounds for occupancy and the satisfiability threshold conjecture. *Random Structures and Algorithms*, 7:59–80, 1995.
22. M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1967.
23. Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes-Rendus du I Congrès de Mathématiciens des Pays Slaves*, pages 92–101, Warszawa, 1929.