Improved Leader Election for Self-organizing Programmable Matter

Joshua J. Daymude[1(✉)], Robert Gmyr[2], Andréa W. Richa[1], Christian Scheideler[2], and Thim Strothmann[2]

[1] Computer Science, CIDSE, Arizona State University, Tempe, AZ, USA
{jdaymude,aricha}@asu.edu

[2] Department of Computer Science, Paderborn University, Paderborn, Germany
{gmyr,scheidel,thim}@mail.upb.de

Abstract. We consider programmable matter that consists of computationally limited devices (called *particles*) that are able to self-organize in order to achieve some collective goal without the need for central control or external intervention. We use the geometric amoebot model to describe such self-organizing particle systems, which defines how particles can actively move and communicate with one another. In this paper, we present an efficient local-control algorithm which solves the leader election problem in $\mathcal{O}(n)$ asynchronous rounds with high probability, where n is the number of particles in the system. Our algorithm relies only on local information — particles do not have unique identifiers, any knowledge of n, or any sort of global coordinate system — and requires only constant memory per particle.

1 Introduction

The vision for *programmable matter* is to create some material or substance that can change its physical properties like shape, density, conductivity, or color in a programmable fashion based on either user input or autonomous sensing of its environment. Many realizations of programmable matter have been proposed — including DNA tiles, shape-changing molecules, synthetic cells, and reconfiguring modular robots — each of which is pursuing solutions applicable to its own situation, subject to domain-specific capabilities and constraints. We envision programmable matter as a more abstract system of computationally limited devices (which we refer to as *particles*) which can move, bond, and exchange information in order to collectively reach a given goal without any outside intervention. *Leader election* is a central and classical problem in distributed computing that is very interesting for programmable matter; e.g., most known shape formation techniques for programmable matter suppose the existence of

J. J. Daymude and A. W. Richa—Supported in part by NSF awards CCF-1422603 and CCF-1637393.

R. Gmyr, C. Scheideler and T. Strothmann—Supported in part by DFG grant SCHE 1592/3-1.

a leader/seed particle (examples can be found in [23] for the nubot model, [20] for the abstract tile self assembly model and [12,13] for the amoebot model).

In this paper, we present a fully asynchronous local-control protocol for the leader election problem, improving our previous algorithm for leader election in [14] which was only described at a high level, lacking specific rules for each particle's execution. Moreover, while the analysis in [14] used a simplified, synchronous setting and only achieved its linear runtime bound in expectation, here we prove with high probability[1] correctness and runtime guarantees for the full local-control protocol[2]. Finally, as this algorithm is both conceptually simpler than that of [14] and presented directly from the point-of-view of an individual particle, it is more easily understood and implemented.

1.1 Amoebot Model

We represent any structure the particle system can form as a subgraph of the infinite graph $G = (V, E)$, where V represents all possible positions the particles can occupy relative to their structure, and E represents all possible atomic movements a particle can perform as well as all places where neighboring particles can bond to each other. In the *geometric amoebot model*, we assume that $G = G_{\text{eqt}}$, where G_{eqt} is the infinite regular triangular grid graph. We recall the properties of the geometric amoebot model necessary for this algorithm; a full description can be found in [14].

Each particle occupies either a single node (i.e., it is *contracted*) or a pair of adjacent nodes in G_{eqt} (i.e., it is *expanded*), and every node can be occupied by at most one particle. Particles move through *expansions* and *contractions*; however, as our leader election algorithm does not require particles to move, we omit a detailed description of these movement mechanisms.

Particles are *anonymous*; they have no unique identifiers. Instead, each particle has a collection of *ports* — one for each edge incident to the node(s) the particle occupies — that have unique labels from the particle's local perspective. We assume that the particles have a common *chirality* (i.e., a shared notion of clockwise direction), which allows each particle to label its ports in clockwise order. However, particles do not have a common sense of global orientation and may have different offsets for their port labels.

Two particles occupying adjacent nodes are connected by a *bond*, and we refer to such particles as *neighbors*. Neighboring particles establish bonds via the ports facing each other. The bonds not only ensure that the particle system forms a connected structure, but also are used for exchanging information. Each particle has a constant-size local memory that can be read and written to by any neighboring particle. Particles exchange information with their neighbors by

[1] An event occurs *with high probability* (*w.h.p.*), if the probability of success is at least $1 - n^{-c}$, where $c > 1$ is a constant; in our context, n is the number of particles.

[2] An astute reader may note that a w.h.p. guarantee on correctness is weaker than the absolute guarantee given for the algorithm in [14], but the latter was given without considering the necessary particle-level execution details.

simply writing into their memory. Due to the constant-size memory constraint, particles know neither the total number of particles in the system nor any estimate of this number.

We assume the standard asynchronous model, wherein particles execute an algorithm concurrently and no assumptions are made about individual particles' activation rates or computation speeds. A classical result under this model is that for any asynchronous concurrent execution of atomic particle activations, there exists a sequential ordering of the activations which produces the same end configuration, provided conflicts which arise from the concurrent execution are resolved (namely, only conflicts of shared memory writes can happen in our algorithm; we simply assume that an arbitrary particle wins). Thus, it suffices to view particle system progress as a sequence of *particle activations*; i.e., only one particle is active at a time. Whenever a particle is activated, it can perform an arbitrary, bounded amount of computation involving its local memory and the memories of its neighbors and can perform at most one movement. We define an *asynchronous round* to be complete once each particle has been activated at least once.

1.2 Related Work

A variety of work related to programmable matter has recently been proposed and investigated. One can distinguish between active and passive systems. In passive systems, the computational units either have no intelligence (moving and bonding is based only on their structural properties or interactions with their environment), or have limited computational capabilities but cannot control their movements. Examples of research on *passive systems* are DNA computing [1,3,7,21], tile self-assembly systems (e.g., the surveys in [15,19,22]), and population protocols [2]. We will not describe these models in detail as they are of little relevance to our approach. *Active systems*, on the other hand, are composed of computational units which can control the way they act and move in order to solve a specific task. Prominent examples of active systems are *swarm robotics* (see, e.g., [17,18]), *modular self-reconfigurable robotic systems* (e.g., [16,24]) — especially *metamorphic robots* [8] — and the *nubot* model [5,6,23] by Woods et al. For an in depth discussion of these models and how they relate to our amoebot model, we refer the reader to the full version of this paper [9].

The *amoebot* model [10] is a model for self-organizing programmable matter that aims to provide a framework for rigorous algorithmic research for nano-scale systems. In [14], the authors describe a leader election algorithm for an abstract (synchronous) version of the amoebot model that decides the problem in expected linear time. Recently, a universal shape formation algorithm [13], a universal coating algorithm [11] and a Markov chain algorithm for the compression problem [4] were introduced, showing that there is potential to investigate a wide variety of problems under this model.

1.3 Problem Description

We consider the classical problem of *leader election*. An algorithm is said to solve the leader election problem if for any connected particle system of initially contracted particles with empty memories, eventually a single particle *irreversibly* declares itself the *leader* (e.g., by setting a dedicated bit in its memory) and no other particle ever declares itself to be the leader. We define the running time of a leader election algorithm to be the number of asynchronous rounds until a leader is declared. Note that we do not require the algorithm to terminate for particles other than the leader.

2 Algorithm

Before we describe the leader election algorithm in detail, we give a short high-level overview. The algorithm consists of six *phases*. These phases are not strictly synchronized among each other, i.e., at any point in time, different parts of the particle system may execute different phases. Furthermore, a particle can be involved in the execution of multiple phases at the same time. The first phase is *boundary setup* (Sect. 2.1). In this phase, each particle locally checks whether it is part of a *boundary* of the particle system. Only the particles on a boundary participate in the leader election. Particles occupying a common boundary organize themselves into a directed cycle. The remaining phases operate on each boundary independently. In the *segment setup* phase (Sect. 2.2), the boundaries are subdivided into *segments*: each particle flips a fair coin. Particles that flip heads become *candidates* and compete for leadership whereas particles that flip tails become *non-candidates* and assist the candidates in their competition. A segment consists of a candidate and all subsequent non-candidates along the boundary up to the next candidate. The *identifier setup* phase (Sect. 2.3) assigns a random identifier to each candidate. The identifier of a candidate is stored distributively among the particles of its segment. In the *identifier comparison* phase (Sect. 2.4), the candidates compete for leadership by comparing their identifiers using a token passing scheme. Whenever a candidate sees an identifier that is higher than its own, it revokes its candidacy. Whenever a candidate sees its own identifier, the *solitude verification* phase (Sect. 2.5) is triggered. In this phase, the candidate checks whether it is the last remaining candidate on the boundary. If so, it initiates the *boundary identification* phase (Sect. 2.6) to determine whether it occupies the unique *outer boundary* of the system. In that case, it becomes the leader; otherwise, it revokes its candidacy.

2.1 Boundary Setup

The boundary setup phase organizes the particle system into a set of *boundaries*. This approach is directly adopted from [14], but we give a full description here to introduce important notation. Let $A \subset V$ be the set of nodes in $G_{\text{eqt}} = (V, E)$ that are occupied by particles. According to the problem definition, the subgraph

$G_{\text{eqt}}|_A$ of G_{eqt} induced by A is connected. Consider the graph $G_{\text{eqt}}|_{V \setminus A}$ induced by the unoccupied nodes in G_{eqt}. We call a connected component R of $G_{\text{eqt}}|_{V \setminus A}$ an *empty region.* Let $N(R)$ be the neighborhood of an empty region R in G_{eqt}; that is, $N(R) = \{u \in V \setminus R : \exists v \in R \text{ such that } (u, v) \in E\}$. Note that by definition, all nodes in $N(R)$ are occupied by particles. We refer to $N(R)$ as the *boundary* of the particle system corresponding to R. Since $G_{\text{eqt}}|_A$ is a finite graph, exactly one empty region has infinite size while the remaining empty regions have finite size. We define the boundary corresponding to the infinite empty region to be the unique *outer boundary* and refer to a boundary that corresponds to a finite empty region as an *inner boundary.*

For each boundary of the particle system, we organize the particles occupying that boundary into a directed cycle. Upon first activation, each particle p instantly determines its place in these cycles using only local information as follows. First, p checks for two special cases. If p has no neighbors, it must be the only particle in the particle system since the particle system is connected. Thus, it immediately declares itself the leader and terminates. If all neighboring nodes of p are occupied, p is not part of any boundary and terminates without participating in the leader election process any further.

If these special cases do not apply, then p has at least one occupied node and one unoccupied node in its neighborhood. Interpret the neighborhood of p as a directed ring of six nodes that is oriented clockwise around p. Consider all maximal sequences of unoccupied nodes $(v_1, \ldots v_k)$ in this ring; call such a sequence an *empty sequence.* Such a sequence is part of some empty region and hence corresponds to a boundary that includes p. Let v_0 be the node before v_1 and let v_{k+1} be the node after v_k in the ring. Note that we might have $v_0 = v_{k+1}$. By definition, v_0 and v_{k+1} are occupied. Particle p implicitly arranges itself as part of a directed cycle spanning the aforementioned boundary by considering the particle occupying v_0 to be its *predecessor* and the particle occupying v_{k+1} to be its *successor* on that boundary. It repeats this process for each empty sequence in its neighborhood.

A particle can have up to three empty sequences in its neighborhood, and consequently can be part of up to three distinct boundaries. However, a particle cannot locally decide whether two distinct empty sequences belong to two distinct empty regions or to the same empty region. To guarantee that the executions on distinct boundaries are isolated, we let the particles treat each empty sequence as a distinct empty region. For each such sequence, a particle acts as a distinct *agent* which executes an independent instance of the algorithm encompassing the remaining five phases of the leader election algorithm. Whenever a particle is activated, it sequentially executes the independent instances of the algorithm for each of its agents in an arbitrary order, i.e., whenever a particle is activated also its agents are activated. Each agent a is assigned the predecessor and successor — denoted a.pred and a.succ, respectively — that was determined by the particle for its corresponding empty sequence. This organizes the set of all agents into disjoint cycles spanning the boundaries of the particle system (see Fig. 1). As consequence of this approach, a particle can occur up to three times on the same boundary as different agents. While we can ignore this property for most of the remaining phases, it will remain a cause for special consideration in the solitude verification phase (Sect. 2.5).

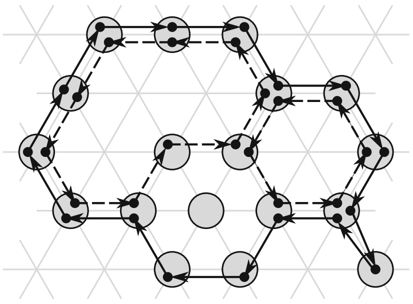

Fig. 1. Boundaries and agents. Particles are depicted as gray circles and the agents of a particle are depicted as black dots inside of the corresponding circle. After the boundary setup phase, the agents form disjoint cycles that span the boundaries of the particle system. The solid arrows represent the unique outer boundary and the dashed arrows represent the two inner boundaries.

2.2 Segment Setup

All remaining phases (including this one) operate exclusively on boundaries, and furthermore execute on each boundary independently. Therefore, we only consider a single boundary for the remainder of the algorithm description. The goal of the segment setup phase is to divide the boundary into disjoint "segments". Each agent flips a fair coin. The agents which flip heads become *candidates* and the agents which flip tails become *non-candidates*. In the following phases, candidates compete for leadership while non-candidates assist the candidates in their competition. A *segment* is a maximal sequence of agents $(a_1, a_2, \ldots, a_k)$ such that a_1 is a candidate, a_i is a non-candidate for $i > 1$, and $a_i = a_{i-1}.\text{succ}$ for $i > 1$. Note that the maximality condition implies that the successor of a_k is a candidate. We refer to the segment starting at a candidate c as $c.\text{seg}$ and call it the segment of c. In the following phases, each candidate uses its segment as a distributed memory.

2.3 Identifier Setup

After the segments have been set up, each candidate generates a random *identifier* by assigning a random digit to each agent in its segment. The candidates use these identifiers in the next phase to engage in a competition in which all but one candidate on the boundary are eliminated. Note that the term identifier is slightly misleading in that two distinct candidates can have the same identifier.

Nevertheless, we hope that the reader agrees that the way these values are used makes this term an appropriate choice.

To generate a random identifier, a candidate c sends a *token* along its segment in the direction of the boundary. A token is simply a constant-size piece of information that is passed from one agent to the next by writing it to the memory of a neighboring particle. While the token traverses the segment, it assigns a value chosen uniformly at random from $[0, r-1]$ to each visited agent where r is a constant that is fixed in the analysis. The identifier generated in this way is a number with radix r consisting of $|c.\text{seg}|$ digits where c holds the most significant digit and the last agent of c.seg holds the least significant digit. We refer to the identifier of a candidate c as c.id. The competition in the next phase of the algorithm is based on comparing identifiers. When comparing identifiers of different lengths, we define the shorter identifer to be lower than the longer identifier.

After generating its random identifier, each candidate creates a copy of its identifier that is stored in *reversed digit order* in its segment. This step is required as a preparation for the next phase. To achieve this, we use a single token that moves back and forth along the segment and copies one digit at a time. More specifically, we reuse the token described above that generated the random identifier. Once this token reaches the end of the segment, it starts copying the identifier by reading the digit of the last agent of the segment and moving to the beginning of the segment. There, it stores a copy of that digit in the candidate c. It then reads the digit of c and moves back to the end of the segment where it stores a copy of that digit in the last agent of the segment. It proceeds in a similar way with the second and the second to last agent and so on until the identifier is completely copied. Afterwards, the token moves back to c to inform the candidate that the identifier setup is complete.

Note that for ease of presentation we deliberately opted for simplicity over speed when creating a reversed copy of the identifier. As we will show in Sect. 3.2, the running time of this simple algorithm is dominated by the running time of the next phase so that the overall asymptotic running time of the leader election algorithm does not suffer.

2.4 Identifier Comparison

During the identifier comparison phase the agents use their identifiers to compete with each other. Each candidate compares its own identifier with the identifier of every other candidate on the boundary. A candidate with the highest identifier eventually progresses to the solitude verification phase, described in the next section, while any candidate with a lower identifier withdraws its candidacy. To achieve the comparison, the non-reversed copies of the identifiers remain stored in their respective segments while the reversed copies move backwards along the boundary as a sequence of tokens. More specifically, a *digit token* is created for each digit of a reversed identifier. A digit token created by the last agent of a segment is marked as a *delimiter token*. Once created, the digit tokens traverse the boundary against the direction of the cycle spanning it. Each agent is allowed to hold at most two tokens at a time, which gives the tokens some space to move along the boundary. The tokens are not allowed to overtake each other, so whenever an agent stores two tokens, it keeps track of the order they were received in and forwards them accordingly. An agent forwards at most one token per activation. Furthermore, an agent can only receive a token after it creates its own digit token. We define the *token sequence* of a candidate c as the sequence of digit tokens created by the agents in c.seg. Note that according to the rules for forwarding tokens, the token sequences of distinct candidates remain separated and the tokens within a token sequence maintain their relative order along the boundary.

Whenever a token sequence traverses a segment c.seg of a candidate c, the agents in c.seg cooperate with the tokens of the token sequence to compare the identifier c.id with the identifier stored in the token sequence. This comparison has three possible outcomes: (*i*) the token sequence is longer than c.seg or the lengths are equal and the token sequence stores an identifier that is strictly greater than c.id, (*ii*) the token sequence is shorter than c.seg or the lengths are equal and the token sequence stores an identifier that is strictly smaller than c.id, or (*iii*) the lengths are equal and the identifiers are equal. In the first case, c does not have the highest identifier and withdraws its candidacy. In the second case, c might be a candidate with the highest identifier and therefore remains a candidate. Finally, in the third case, c initiates the solitude verification phase, which is then executed in parallel to the identifier comparison phase. Solitude verification might be triggered quite frequently, especially for candidates with short segments; we describe how this is handled in the next section. Due to space constraints, we omit the exact token passing scheme for identifier comparison and refer to the full version of this paper [9].

2.5 Solitude Verification

The goal of the solitude verification phase is for a candidate c to check whether it is the last remaining candidate on its boundary. Solitude verification is triggered during the identifier comparison phase whenever a candidate detects equality between its own identifier and the identifier of a token sequence that traversed its segment. Note that such a token sequence can either be the token sequence created by c itself or the token sequence created by some other candidate that generated the same identifier. Once the solitude verification phase is started, it runs in parallel to the identifier comparison phase and does not interfere with it. This phase is based on the idea of solitude verification given in [14], but greatly simplifies many of the original ideas to obtain a more easily understood protocol.

A candidate c can check whether it is the last remaining candidate on its boundary by determining whether or not the next candidate in direction of the cycle is c itself. To achieve this, the solitude verification phase has to span not only c.seg but also all subsequent segments of former candidates that already withdrew their candidacy during the identifier comparison phase. We refer to the union of these segments as the *extended segment* of c. The basic idea of the algorithm is the following. We treat the edges that connect the agents on the boundary as vectors in the two-dimensional Euclidean plane. If c is the last remaining candidate on its boundary, the vectors corresponding to the directed edges of the boundary cycle in the extended segment of c and the next edge (connecting the extended segment of c to the next candidate) sum to the zero vector, implying that the next candidate and c occupy the same node. To perform this summation in a local manner, c locally defines a two-dimensional coordinate system (e.g., by choosing two consecutive ports as the x and y axes, respectively) and uses two token passing schemes to generate and sum the x and y coordinates of these vectors in parallel. Again, due to space constraints, the details of this token passing scheme for summing x or y vector coordinates is detailed in the full version of this paper [9].

Using the token passing scheme, a candidate c can decide whether the next candidate along the boundary is itself. However, this is not sufficient to decide whether c is the last remaining candidate on the boundary. As described in Sect. 2.1, a particle can occur up to three times as different agents on the same boundary. Therefore, there can be distinct agents on the same boundary that occupy the same node of G_{eqt}. If an extended segment reaches from one of these agents to another, the vectors induced by the extended segment sum up to the zero vector even though there are at least two agents left on the boundary. To handle this case, each particle assigns a locally unique agent identifier from $\{1, 2, 3\}$ to each of its agents in an arbitrary way. The token passing scheme then additionally checks that the agent identifier of the last agent in the extended segment matches that of c, ensuring that c is the last remaining candidate on its boundary.

Finally, we must address the interaction between the solitude verification phase and the identifier comparison phase. As noted in the previous section, solitude verification may be triggered quite frequently. Therefore, it may occur that solitude verification is triggered for a candidate c while c is still performing a previously triggered execution of solitude verification. In this case, c simply continues with the already ongoing execution and ignores the request for another execution. Furthermore, c might be eliminated by the identifier comparison phase while it is performing solitude verification. In this case, c waits for the ongoing solitude verification to finish and only then withdraws its candidacy.

2.6 Boundary Identification

Once a candidate c determines that it is the only remaining candidate on its boundary, it initiates the boundary identification phase to check whether or not it lies on the unique outer boundary of the particle system. If it lies on the outer boundary, the particle acting as candidate agent c declares itself the leader. Otherwise, c revokes its candidacy. To achieve this, we make use of the observation that the outer boundary is oriented clockwise while an inner boundary is oriented counter-clockwise (see Fig. 1), a property resulting directly from the way the an agent's predecessor and successor are defined in Sect. 2.1.

A candidate c can distinguish between clockwise and counter-clockwise oriented boundaries using a simple token passing scheme introduced in [14]. It sends a token along the boundary that sums up the angles of the turns it takes according to Fig. 2, storing the results in a counter α. When the token returns to c, the absolute value $|\alpha|$ represents the external angle of the polygon induced by the boundary. It is well known that the external angle of a polygon in the Euclidean plane is $|\alpha| = 360°$. Since the outer boundary is oriented clockwise and an inner boundary is oriented counter-clockwise, we have $\alpha = 360°$ for the outer boundary and $\alpha = -360°$ for an inner boundary. The token can encode α as an integer k such that $\alpha = k \cdot 60°$. To distinguish the two possible final values of k it is sufficient to store k modulo 5 so that we have $k = 1$ for the outer boundary and $k = 4$ for an inner boundary. Therefore, the token only needs three bits of memory.

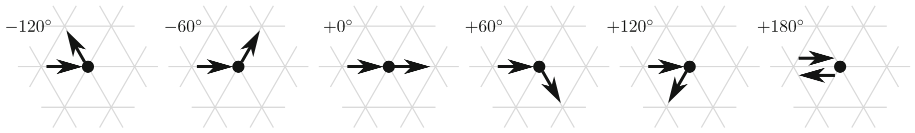

Fig. 2. Determining the external angle α. The incoming and outgoing arrows represent the directions in which the token enters and leaves an agent, respectively. Only the angle between the arrows is relevant; the absolute global direction of the arrows cannot be detected by the agents since they do not posses a common compass.

3 Analysis

We now turn to the analysis of the leader election algorithm. We first show its correctness in Sect. 3.1 and then analyze its running time in Sect. 3.2. Due to space constraints, some of the supporting lemmas and their proofs are omitted; they can be found in the analysis section of the full version of this paper [9].

3.1 Correctness

To show the correctness of the algorithm we must prove that eventually a single particle irreversibly declares itself to be the leader of the particle system and no other particle ever declares itself to be the leader. Any agent on an inner boundary can never cause its particle to become the leader; even if the algorithm reaches the point at which there is exactly one candidate c on some inner boundary, c will withdraw its candidacy in the boundary identification phase. Therefore, we can focus exclusively on the behavior of the algorithm on the unique outer boundary. We focus only on the major theorem here.

Theorem 1. *The algorithm solves the leader election problem, w.h.p.*

Proof. We must show that eventually a single particle irreversibly declares itself to be the leader of the particle system and no other particle ever declares itself to be the leader. Again, we consider only the agents on the outer boundary as agents on an inner boundary will never cause their particles to declare themselves as leaders. Once every particle has finished the boundary setup phase, every agent has finished the segment setup phase, and every candidate has finished the identifier setup phase, with high probability[3] there is a unique candidate c^* that has the highest identifier on the outer boundary. Since c^* has the highest identifier, it does not withdraw its candidacy during the identifier comparison phase. In contrast, every other candidate $c \neq c^*$ eventually withdraws its candidacy because the token sequence of c^* eventually traverses c.seg. Therefore, such an agent c cannot cause its particle to become the leader. Once c^* is the last remaining candidate on the outer boundary, it eventually triggers the solitude verification phase because the token sequence of c^* eventually traverses c^*.seg while c^* is not already performing solitude verification. After verifying that it is the last remaining candidate, c^* executes the boundary identification phase and determines that it lies on the outer boundary. It then instructs its particle to declare itself the leader of the particle system. □

3.2 Running Time

Recall from Sect. 1.3 that the running time of an algorithm for leader election is defined as the number of asynchronous rounds until a leader is declared. Since the given algorithm always establishes a leader on the outer boundary, we can limit our attention to that boundary. Let n be the number of particles in the system and L be the number of agents on the outer boundary.

The first two phases of the algorithm, namely the boundary setup and segment setup phases, consist entirely of computations based on local neighborhood information. Therefore, these phases can be completed instantly by each particle upon its first activation. Since each particle is activated at least once in every round, every particle completes these first two phases after a single round. When an agent becomes a candidate, it initiates the identifier setup phase. We have the following lemma.

Lemma 1. *All candidates on the outer boundary complete the identifier setup phase after $\mathcal{O}(\log^2 n)$ rounds, w.h.p.*

After the identifiers have been generated, they are compared in the identifier comparison phase. In this phase, a set of digit tokens, one for each agent on the boundary, traverses the boundary against the direction of the cycle spanning it. Each agent can store at most two tokens. The tokens are not allowed to overtake each other, so agents maintain the order of the tokens when forwarding them. Note that a token is never delayed unless it is blocked by tokens in front of it.

[3] This w.h.p. guarantee results from there being a small but nonzero probability that either (a) all agents flip tails and become non-candidates in the segment setup phase, or (b) more than one candidate generates the same highest identifier in the identifier setup phase. See [9] for more details.

Therefore, an agent a forwards a token whenever a.pred can hold an additional token. Finally, an agent forwards at most one token for each activation.

We define the number of *steps* a token has taken as the number of times it's been forwarded from one agent to the next since its creation. Let T be the earliest round such that at its beginning every agent on the outer boundary has created its digit token. We have the following lemma.

Lemma 2. *At the beginning of round $T + i$ for $i \in \mathbb{N}$, each digit token on the outer boundary has taken at least i steps.*

Next, the following lemma provides an upper bound on the running time of the solitude verification phase.

Lemma 3. *For an extended segment of length ℓ, the solitude verification phase takes $\mathcal{O}(\ell)$ rounds.*

The boundary identification phase is only executed once a candidate determines that it is the last remaining candidate on the boundary. The following lemma provides an upper bound for the running time of this phase.

Lemma 4. *The boundary identification phase on the outer boundary takes $\mathcal{O}(L)$ rounds.*

Finally, we can show the following runtime bound.

Theorem 2. *The algorithm solves the leader election problem in $\mathcal{O}(L)$ rounds, w.h.p.*

Theorem 2 specifies the running time of the leader election algorithm in terms of the number of *agents* on the outer boundary. Let C be the number of *particles* on the outer boundary. Since each particle on the outer boundary corresponds to at most three agents on the outer boundary, we have that the algorithm solves the leader election problem in $\mathcal{O}(C)$ rounds, w.h.p.. Moreover, the number of particles on the outer boundary is obviously at most n; thus, the runtime bound can also be formulated as $\mathcal{O}(n)$ rounds, w.h.p.. Note that compared to the $\mathcal{O}(C)$ bound, the $\mathcal{O}(n)$ bound is quite pessimistic since the number of particles on the outer boundary can much lower than n. For example, a solid square of n particles only has $C = \mathcal{O}(\sqrt{n})$ particles on its outer boundary.

4 Conclusion

In this paper we presented a randomized leader election algorithm for programmable matter which requires $\mathcal{O}(n)$ asynchronous rounds with high probability. The main idea of this algorithm is to use coin flips to set up random identifiers for each leader candidate in such a way that at least one candidate has an identifier of logarithmic length, leading to a unique leader w.h.p.. In the full version of this paper [9], we consider several variants of the leader election problem and detail how our algorithm can be modified to solve them. These variants include allowing particle systems to contain both expanded and contracted particles, enforcing that all particles terminate their executions of the algorithm (instead of requiring only the leader to terminate), and improving the with high probability guarantee on electing a leader to a with probability 1 guarantee without changing the $\mathcal{O}(L)$, w.h.p. runtime bound.

References

1. Adleman, L.M.: Molecular computation of solutions to combinatorial problems. Science **266**(11), 1021–1024 (1994)
2. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distrib. Comput. **18**(4), 235–253 (2006)
3. Boneh, D., Dunworth, C., Lipton, R.J., Sgall, J.: On the computational power of DNA. Discrete Appl. Math. **71**, 79–94 (1996)
4. Cannon, S., Daymude, J.J., Randall, D., Richa, A.W.: A Markov chain algorithm for compression in self-organizing particle systems. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, 25–28 July 2016, pp. 279–288 (2016)
5. Chen, H.-L., Doty, D., Holden, D., Thachuk, C., Woods, D., Yang, C.-T.: Fast algorithmic self-assembly of simple shapes using random agitation. In: Murata, S., Kobayashi, S. (eds.) DNA 2014. LNCS, vol. 8727, pp. 20–36. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11295-4_2
6. Chen, M., Xin, D., Woods, D.: Parallel computation using active self-assembly. In: Soloveichik, D., Yurke, B. (eds.) DNA 2013. LNCS, vol. 8141, pp. 16–30. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-01928-4_2
7. Cheung, K.C., Demaine, E.D., Bachrach, J.R., Griffith, S.: Programmable assembly with universally foldable strings (moteins). IEEE Trans. Rob. **27**(4), 718–729 (2011)
8. Chirikjian, G.: Kinematics of a metamorphic robotic system. In: Proceedings of the 1994 IEEE International Conference on Robotics and Automation, IRCA 1994, vol. 1, pp. 449–455 (1994)
9. Daymude, J.J., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Improved leader election for self-organizing programmable matter. CoRR, abs/1701.03616 (2017)
10. Derakhshandeh, Z., Dolev, S., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Brief announcement: amoebot - a new model for programmable matter. In: 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2014, Prague, Czech Republic, 23–25 June 2014, pp. 220–222 (2014)
11. Derakhshandeh, Z., Gmyr, R., Porter, A., Richa, A.W., Scheideler, C., Strothmann, T.: On the runtime of universal coating for programmable matter. In: Rondelez, Y., Woods, D. (eds.) DNA 2016. LNCS, vol. 9818, pp. 148–164. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43994-5_10
12. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: An algorithmic framework for shape formation problems in self-organizing particle systems. In: Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication, NANOCOM 2015, Boston, MA, USA, 21–22 September 2015, pp. 21:1–21:2 (2015)

13. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Universal shape formation for programmable matter. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, 11–13 July 2016, pp. 289–299 (2016)
14. Derakhshandeh, Z., Gmyr, R., Strothmann, T., Bazzi, R., Richa, A.W., Scheideler, C.: Leader election and shape formation with self-organizing programmable matter. In: Phillips, A., Yin, P. (eds.) DNA 2015. LNCS, vol. 9211, pp. 117–132. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21999-8_8
15. Doty, D.: Theory of algorithmic self-assembly. Commun. ACM **55**(12), 78–88 (2012)
16. Fukuda, T., Nakagawa, S., Kawauchi, Y., Buss, M.: Self organizing robots based on cell structures - CEBOT. In: Proceedings of the 1988 IEEE International Conference on Intelligent Robots and Systems, IROS 1988, pp. 145–150 (1988)
17. Kernbach, S. (ed.): Handbook of Collective Robotics - Fundamentals and Challanges. Pan Stanford Publishing, Singapore (2012)
18. McLurkin, J.: Analysis and implementation of distributed algorithms for multirobot systems. Ph.D. thesis, Massachusetts Institute of Technology (2008)
19. Patitz, M.J.: An introduction to tile-based self-assembly and a survey of recent results. Nat. Comput. **13**(2), 195–224 (2014)
20. Rothemund, P.W.K., Winfree, E.: The program-size complexity of self-assembled squares (extended abstract). In: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, Portland, OR, USA, 21–23 May 2000, pp. 459–468 (2000)
21. Winfree, E., Liu, F., Wenzler, L.A., Seeman, N.C.: Design and self-assembly of two-dimensional DNA crystals. Nature **394**(6693), 539–544 (1998)
22. Woods, D.: Intrinsic universality and the computational power of self-assembly. In: Proceedings of MCU 2013, pp. 16–22 (2013)
23. Woods, D., Chen, H.-L., Goodfriend, S., Dabby, N., Winfree, E., Yin, P.: Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In: Proceedings of the 4th Conference on Innovations in Theoretical Computer Science, ITCS 2013, pp. 353–354 (2013)
24. Yim, M., Shen, W.-M., Salemi, B., Rus, D., Moll, M., Lipson, H., Klavins, E., Chirikjian, G.S.: Modular self-reconfigurable robot systems. IEEE Robot. Autom. Mag. **14**(1), 43–52 (2007)