Mobile RAM and Shape Formation by Programmable Particles

Giuseppe Antonio Di Luna[1], Paola Flocchini[2], Nicola Santoro[3],
Giovanni Viglietta[4(✉)], and Yukiko Yamauchi[5]

[1] Sapienza University of Rome, Rome, Italy
diluna@diag.uniroma1.it
[2] University of Ottawa, Ottawa, Canada
paola.flocchini@uottawa.ca
[3] Carleton University, Ottawa, Canada
santoro@cs.carleton.ca
[4] JAIST, Nomi city, Japan
johnny@jaist.ac.jp
[5] Kyushu University, Fukuoka, Japan
yamauchi@inf.kyushu-u.ac.jp

Abstract. In the distributed model *Amoebot* of programmable matter, the computational entities, called particles, are anonymous finite-state machines that operate and move on a hexagonal tessellation of the plane. In this paper we show how a constant number of such weak particles can simulate a powerful Turing-complete entity that is able to move on the plane while computing. We then show an application of our tool to the classical Shape-Formation problem, providing a new and much more general distributed solution. Indeed, while the existing algorithms allow to form only shapes made of arrangements of segments and triangles, our algorithm allows the particles to form also more abstract and general connected shapes, including circles and spirals, as well as fractal objects of non-integer dimension. In lieu of the existing impossibility results based on the symmetry of the initial configuration of the particles, our result provides a complete characterization of the connected shapes that can be formed by an initially simply connected set of particles. Furthermore, in the case of non-connected target shapes, we give almost-matching necessary and sufficient conditions for their formability.

Keywords: Distributed algorithms · Programmable matter · Amoebot · Shape formation · Turing-computable shapes · RAM simulation

1 Introduction

Several parallel and distributed computing models have been devoted to formalizing computations within the interdisciplinary field of Programmable Matter (PM): see [8,10,11,13]. The PM field envisions a myriad of very small

(micro/nano-sized) entities that are nevertheless able to move and coordinate themselves with the final purpose of solving a specific tasks [15]. Prototypes that will lead to future hardware platforms for PM are being designed and engineered. Examples are the M-blocks: cubes that are able to rearrange themselves by rotations [12], and the Kilobots: small robots that move by vibrations [14]. At the same time, the algorithmic community is formalizing abstract and general models, enabling the development of provably correct algorithms and the feasibility analysis of problems.

In this paper, we consider the popular geometric *Amoebot* model, introduced in [5]. In the Amoebot model, a set of computationally limited identical entities, called *particles*, operate and move on a hexagonal tessellation of the plane (i.e., a triangular grid). Each particle has constant-size memory (i.e., constant with respect to the total number of particles), is anonymous (i.e., it has no ID), is able to communicate only with its direct neighbors on the grid, and it moves by repeating an *expansion action* (in which the particle expands to occupy two neighboring nodes of the grid) and a *contraction action* (in which an expanded particle contracts to a single node of the grid). Research using this model is being carried out within the parallel, distributed, and natural computing fields [2–7]; for a recent survey, see [1]. The main goal of these research efforts is to gain an understanding of the nature and limits of this distributed computational universe.

In this paper, we move one step forward in this quest by providing a construction that simulates a *moving Random-Access Machine* (mRAM) using four particles. Such a construction transforms a set of these weak particles into a powerful Turing-complete entity, able to move on the grid while computing.

We prove the usefulness of our construction by applying it to the well-studied *Shape-Formation* (or Pattern-Formation) problem. In this problem, the particles, initially arranged in an arbitrary connected shape, have to form a given target shape. More precisely, each particle starts with a representation of the target shape in its memory, and coordinates with the other particles to form a suitably scaled-up copy of the shape that includes all particles in the system (this could mean that some particles have to be in the expanded state in the final shape). Usually, the total number of particles n is unknown: as a matter of fact, n cannot be stored in the constant-size memory of a single particle.

Increasingly refined and complex techniques and algorithms have been designed [2,3,5,6], each enlarging the class of shapes that can be formed starting from a simply connected configuration. To date, however, this class includes only target shapes defined as an arrangement of segments and triangles.

Our second contribution is the development, using our mRAM simulation, of a general and universal solution for the Shape-Formation problem: starting from a simply connected shape (i.e., a shape without holes), our algorithm allows the particles to form any feasible *connected* shape for which a "drawing algorithm" exists (i.e., the shape is Turing-computable), including circles and spirals, or more complex fractal objects of non-integer dimension, such as the Sierpinski triangle or the Koch snowflake (see Fig. 1).
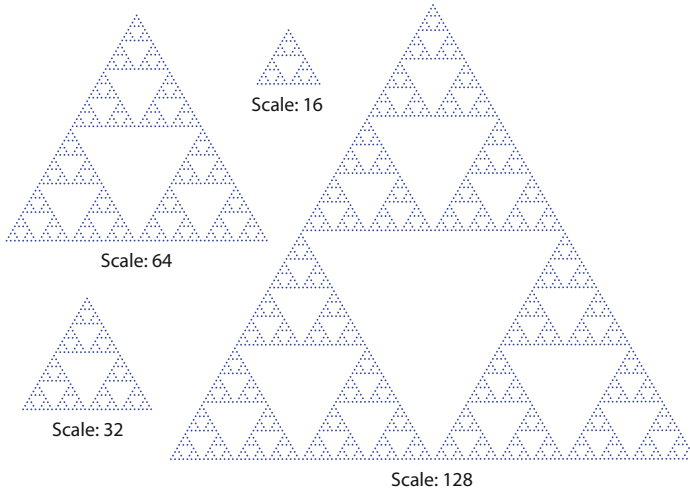
Fig. 1. Sierpinski triangle at various scales, approximated by a system of particles

The aforementioned drawing algorithm is a RAM algorithm that takes as input the number of particles n, and outputs (a representation of) the shape that has to be formed by the n particles. Note that, since the number of instructions of such a drawing algorithm is obviously constant with respect to n,[1] the algorithm itself can fit in the constant-size memory of a single particle, regardless of the number of particles in the system. (By comparison, previous works assumed that each particle has in memory a representation of the target shape expressed as a constant number of segments and triangles [2,3,6].)

With our technique, we can form almost all "feasible" *non-connected* target shapes, and are excluded from our result only very sparse pathological shapes. With regards to the *feasibility* (or not) of a shape, it is known that, depending on the symmetry of the initial configuration, some shapes are not formable regardless of the amount of memory [6]. The negative result of [6] and the positive results of our paper give an almost complete characterization of shapes that are formable starting from a simply connected configuration. Interestingly, in the case of *connected* shapes, the characterization is complete.

Our distributed algorithm is deterministic, and it works even if the schedule of activations is fair but adversarial. More precisely, in each stage, upon activation, a particle exchanges messages with its neighbors, executes some computation, and possibly moves; there is however no restriction on the number of particles that can be concurrently active in the same stage. Moreover, our algorithm works also when the particles do not have *chirality* (i.e., there is no common notion of a clockwise direction on the plane among the particles).

For space reasons, some details have been omitted; the full version is available at https://arxiv.org/abs/2002.03384.

---

[1] By contrast, the *running time* of the drawing algorithm may be any function of n.

2   Model and Problem

2.1   Particles

In the Amoebot model, a *particle* is a computational entity that lives in an infinite regular triangular grid G embedded in the Euclidean plane $\mathbb{R}^2$ (observe that a regular triangular grid corresponds to a hexagonal tessellation of the plane). A particle may occupy either one vertex of G or two adjacent vertices: in the first case, the particle is said to be *contracted*; otherwise, it is *expanded*. When it is expanded, one of the vertices it occupies is called its *head*, and the other vertex is its *tail*. A particle may move through G by repeatedly expanding toward a neighboring vertex of G and contracting into its head. (The traditional Amoebot model also includes a special type of coordinated move called "handover", but we will not need it in this paper.)

No vertex of G can ever be occupied by more than one particle at a time. So, a contracted particle cannot expand into a vertex that is already occupied by another particle. Also, if two or more particles attempt to expand toward the same (unoccupied) vertex at the same time, only one of them succeeds, chosen arbitrarily by an adversary.

At each *stage*, some particles in the system are *active*, and they perform a *look-compute-move cycle*, and the other particles are *inactive*. An adversarial *scheduler* arbitrarily and unpredictably decides which particles are active at each stage. The only restriction on the scheduler is that it cannot keep a particle inactive forever, but it must activate every particle infinitely often. When a particle is activated for a certain stage, it "looks" at the vertices of G adjacent to its head, discovering if they are currently unoccupied, or if they are head or tail vertices of some particle. All particles are *anonymous* (i.e., they are indistinguishable). Each active particle may then decide to expand, contract, or stay still for that stage. When the next stage starts, a new set of active particles is selected, which observe their surroundings and move, and so on.

Each particle has an *internal state* that it can modify every time it is activated. The internal state of any particle is from a finite set: particles have an amount of "memory" whose size is constant with respect to the size of the system, n.

Two particles can also *communicate* by sending *messages* to one another. Each message is taken from a finite set of predefined messages. An active particle can send a message to another particle provided that their heads are adjacent vertices of G. A particle reads the incoming messages from all its neighbors as soon as it is activated.

Each particle labels its six incident edges with *port numbers*, going from 0 to 5. Each particle uses a consistent numbering that is invariant under translation on G. However, different particles may disagree on which of the edges has port number 0 and whether the numbering should follow the clockwise or counterclockwise order: this is called the particles' *handedness*. So, the handedness of a particle does not change as the particle moves, but different particles may have different handedness.

At each stage, each active particle determines which neighboring vertices are occupied, and it reads the incoming messages. Based on these and on its internal state, the particle executes a *deterministic algorithm* that computes a new internal state, the messages to be sent to the neighbors, and whether the particle should expand to some adjacent vertex, contract, or stay still.

2.2 Shapes

A set S of nodes of G is *formable by n particles* if there exists a configuration of exactly n particles (contracted or expanded) that collectively occupy exactly the nodes in S.

A *shape* is a function mapping a positive integer n to a set S_n of nodes of G that is formable by n particles: this set S_n is called the nth *level* of the shape. If such a function is Turing-computable, then the shape is said to be *computable*. If every level of a shape is a connected set, the shape is said to be *connected*.

A shape is *formable* (under condition $\mathcal{C}$) if there exists a distributed algorithm that, for every n, makes any system of n particles (whose initial configuration satisfies condition $\mathcal{C}$) eventually form a copy of the nth level of the shape, possibly translated, rotated by a multiple of 60°, and reflected. The algorithm should succeed regardless of the port numbers of each particle and the choices of the adversarial scheduler, and guarantee that the system remains still after forming the shape.

2.3 Unbreakable Symmetries

A configuration of particles is said to be *unbreakably k-symmetric*, for some integer $k > 1$, if it has a center of k-fold rotational symmetry that does not coincide with any node of G [6]. Observe that unbreakably k-symmetric configurations exist only for $k = 2$ and $k = 3$.

We can extend this notion to a level of a shape in a natural way: the nth level of a shape is k unbreakably symmetric if it is formable by an unbreakably k-symmetric configuration of n particles. Intuitively, the absence of a central node (and therefore of a central particle) and the fact that the configuration is initially symmetric makes it impossible to break the symmetry. Hence, we have a *necessary* condition $\mathcal{C}$ for the formability of a shape:

Proposition 1. *[6] A shape is formable only under the condition that, if its nth level is not unbreakably k-symmetric, then the initial configuration of a system of n particles seeking to form the shape should not be unbreakably k-symmetric, either.*

Note that, when $k = 1$ the shape is not *unbreakably k-symmetric* in such a case, as we will show in next sections, the symmetry of the target shape is irrelevant. In Sect. 5 we will give a strong *sufficient* condition for the formability of a shape, which, together with Proposition 1, almost characterizes the set of computable shapes that can be formed from a simply connected initial configuration of particles (in the case of *connected* computable shapes, it yields a full characterization).

3 Simulating Random-Access Machines

3.1 Random-Access Machines: Definition

A *Random-Access Machine* (RAM) is a model of computation consisting of a finite set of *registers*, each of which can store a non-negative integer, and a *program* consisting of a finite ordered sequence of *instructions*. Each instruction is of one of two types:

- Inc(r): increment by 1 the value stored in the register r. Then, proceed to the next instruction of the program.
- TestDec(r, i): if the register r is holding the value 0, jump to the ith instruction of the program. Otherwise, decrement r by 1 and proceed to the next instruction.

The registers initially contain the *input* of the RAM, and then the program is executed starting from the first instruction. We can reserve a register r_t to represent a "termination flag", whose value is initially 0. When r_t is incremented, the value of the other registers is taken as the RAM's *output*. Hence, a RAM is a device that can compute integer functions.

3.2 Simulating Turing Machines by RAMs

In [9, Chapter 11], Minsky shows how RAMs can simulate Turing Machines. Specifically, there is a (small) constant c such that, given any Turing Machine T that computes a function f_T, there exists a RAM Q_T with exactly c registers that computes f_T.

   Then, in [9, Chapter 14], he shows that any RAM R can be simulated by a RAM Q'_R having only two registers. This is done by encoding the set of values stored in R's registers as a single integer, which is stored in the first register of Q'_R (the second register of Q'_R is only used for intermediate computations). The code is based on Gödel numbers: for instance, the sequence $(a_1, a_2, a_3, a_4, a_5)$ is encoded as the single integer $2^{a_1} \cdot 3^{a_2} \cdot 5^{a_3} \cdot 7^{a_4} \cdot 11^{a_5}$. In general, the ith integer in the sequence becomes the exponent of the ith prime factor of the code. By the unique-factorization theorem, this encoding is injective and thus non-ambiguous.

   The program of Q'_R can be constructed by locally replacing each instruction of R with a small program that simulates it. For instance, it is possible to increment the ith register of R by multiplying the first register of Q'_R by the ith prime number, which in turn can be done in Q'_R by using the two standard instructions and the auxiliary register. Testing if the ith register of R is non-0 amounts to testing if the first register of Q'_R holds a multiple of the ith prime number, etc. In this paradigm, we can stipulate that Q'_R terminates when its second register holds a 0 and the value of its first register is a multiple of the prime number corresponding to the termination flag of R.

   As an immediate consequence of the above, we have that the RAMs with only two registers can simulate all Turing Machines, and can therefore compute any computable function.

3.3   Simulating RAMs by Particles

Our goal in this section is to simulate a RAM with two registers by a set of four particles. The layout of our simulator is shown in Fig. 2: the four particles always remain collinear throughout the simulation, and they always maintain their order along the line: first the *pivot* P, then the *marker of the second register* M_2, then the *leader* L, and finally the *marker or the first register* M_1. The number of empty locations between P and M_2 (i.e., their distance minus 1) represents the value stored in the second register of the RAM, and the number of empty locations between M_2 and M_1 (i.e., their distance minus 2) represents the value stored in the first register of the RAM.

At all times during the simulation of the RAM's program, L will remember the index of the instruction that is currently being simulated: since the program of a RAM is constant L only needs a constant amount of memory to do so. Now we have to show that such a system can simulate every possible instruction of the RAM. This is done by letting L move between M_2 and M_1: we assume that L knows in which direction it has to move to find each of these two particles, and for convenience we call these directions "left" and "right", respectively, to match Fig. 2.

Fig. 2. A RAM simulator with the first register holding the value 7 and the second register holding the value 3

When L reaches the relevant particle, it communicates with it and causes it to move according to the current instruction of the program. P, on the other hand, always remains still. Each instruction is simulated as follows:

- Inc(Register 1): L moves toward M_1 until it finds it. Then it gives M_1 the order to move one step to the right, and waits until M_1 has moved.
- TestDec(Register 1, i): if L neighbors both M_2 and M_1, it does nothing (and updates the index of the current instruction to i). Otherwise, it reaches M_1 and orders it to move one step to the left. Then L itself moves one step to the left and waits until M_1 has moved.
- Inc(Register 2): L reaches M_1 and orders it to move one step to the right. When M_1 has moved, L reaches M_2 and orders it to move one step to the right. Then L itself moves one step to the right and waits for M_2.
- TestDec(Register 2, i): L reaches M_2 and asks it if it has a neighbor on the opposite side (i.e., P). If M_2 answers affirmatively, L does nothing (and updates the index of the current instruction to i). Otherwise, it orders M_2 to move one step to the left and waits until it has moved; then it reaches M_1 and orders it to move one step to the left; finally, L itself moves one step to the left and waits until M_1 has moved.

3.4   Adding Control Registers for Mobility

In our Shape-Formation algorithm, we need the RAM simulator to be able to move and to perform actions depending on the shape to be formed. In this section we outline the mechanism that our RAM we will use to move, and the specific actions needed for the Shape-Formation algorithm.

Recall that, in order to simulate a generic Turing Machine T, the RAM Q_T needs only a constant number c of registers. We can augment Q_T by adding a fixed number c' of "flag registers", which are set to 1 when the system has to perform certain special operations. Specifically, assume that T is an algorithmic description of a shape, whose output is a sequence of *plotting operations* of the form "move forward", "move right", "draw a point", etc. Our mobile RAM (mRAM) U_T simulates T exactly like Q_T, with one exception: whenever T outputs a plotting operation, U_T sets and then immediately resets the flag register corresponding to that type of operation (the reason will be explained shortly).

Now, we let the mRAM Q'_{U_T} simulate U_T using only two registers. In Q'_{U_T}, the state of each of the c' flag registers of U_T can be checked by verifying, at the end of a simulated instruction, if the value stored in the first register is a multiple of the prime number associated with the flag register.

As our four-particle system simulates Q'_{U_T}, it can also test the c' flag registers of U_T at the end of every simulated instruction. Indeed, the leader particle L can test if the value stored in the first register is a multiple of a given prime p. It can do so by first moving next to M_1, and then counting modulo p the number of steps it takes to move all the way to M_2 (counting modulo p requires only p states). Since $c + c'$ is a finite constant, L only ever needs to test a constant set of primes to determine the states of all the flag registers, which in turn takes a finite amount of memory and time.

When L determines that one of the c' flags is set, it executes the corresponding plotting operation of T. This translates into a "movement operation", which moves the whole system in some direction or orders a specific particle to remain still forever, marking a point of the shape. The exact nature of these movement operations and the details of their implementation will be described in next sections.

4   Basic Shape-Formation Algorithm

The first part of our Shape-Formation algorithm is taken from the "basic algorithm" of [6], while the second part is entirely different, and will be described in Sect. 5.

An assumption of the basic algorithm, is that the initial configuration of the particles is *simply connected*; another assumption is that the initial configuration and the shape to be formed satisfy the necessary condition of Proposition 1.

As we pointed out in Sect. 2, the basic algorithm only deals with shapes that are made of full triangles and segments, but this assumption is not used in the parts of the basic algorithm that we are going to borrow here. The relevant "phases" of the basic algorithm are as follows:

– *Handedness agreement*: all the particles assume the same handedness (by simply setting an internal flag whose meaning is either "my original handedness is correct" or "my original handedness is incorrect").
– *Leader election*: the particles attempt to elect a single leader. If the initial configuration is unbreakably k-symmetric, they may fail to do so, and elect exactly k leaders instead.
– *Straightening*: the particles arrange themselves in k straight line segments, each with a leader located at an endpoint. If $k > 1$, all the leaders are pairwise adjacent (recall that the only possibilities are $k = 2$ and $k = 3$), and the configuration is unbreakably k-symmetric (hence the k line segments have the same length and form angles of $2\pi/k$).

In the next section we are going to show how to proceed from here: we have a configuration consisting of k straight lines and k leaders, where all particles have the same handedness, and we want to reach an arbitrary configuration (i.e., a level of a computable shape) that is unbreakably k-symmetric if $k > 1$.

5   Forming Turing-Computable Shapes
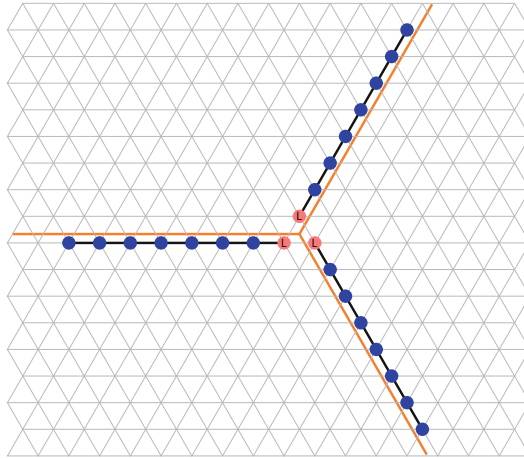
5.1   Starting Configuration

Recall from Sect. 4 that, starting from a simply connected configuration, all the particles in the system can agree on a common handedness and rearrange themselves to form a configuration C_0 consisting of k equal line segments (with $k \in \{1, 2, 3\}$) each containing a leader particle. Moreover, if $k > 1$, then C_0 is unbreakably k-symmetric, and we may assume that also the shape's level to be formed is unbreakably k-symmetric. In this case, each leader is assigned an equal portion of the shape, corresponding to a sector of plane spanned by an angle of $2\pi/k$. These k sectors are called *principal sectors* of the plane, and the k rays separating them are the *principal rays*: Fig. 3 shows an example for $k = 3$.

As a preliminary move, each leader will reach the far end of the line segment on which it is located. This is done by repeatedly "transferring the leadership" to a neighboring particle. In turn, this amounts to sending a special message to that neighbor, whose meaning is "you are now the leader, and I am a regular particle". So, no particle will actually move in this phase.

When a leader has reached the far end of the segment, it starts the next phase, which consists in building and initializing a mRAM simulator that will eventually form the portion of shape that falls into that principal sector. From this time on, the k leaders will act independently of each other, never meeting again and never interacting.

5.2   From a Shape-Generation Algorithm to a Tracing mRAM

As stated in Sect. 2, we assume that the shape to be formed is computable, i.e., there exists an algorithm A that generates its nth level S_n given the number n as input. By this we mean that A outputs the coordinates of all the points

Fig. 3. The final configuration of the basic algorithm, which is the starting point C_0 of our new algorithm. Each leader is assigned a trail of followers, and will guide them in the formation of the part of shape that falls into its principal sector.

of S_n, in an arbitrary order and in some arbitrary coordinate system, and then terminates. Recall that, by definition of shape, there exists a configuration C_f of n particles, expanded or contracted, which collectively form S_n. Additionally, due to Proposition 1, if $k > 1$, then S_n must be unbreakably k-symmetric, and therefore we may take C_f to be an unbreakably k-symmetric configuration of particles. As such, C_f can be translated and rearranged so that its center coincides with the center of C_0. Also, C_f can be decomposed into k equal subsets, called *principal subsets*, each of which entirely lies in a principal sector of C_0, with one exception: some expanded particles of C_f may be crossing a principal ray, and therefore have the head in a principal sector and the tail in another. These will be called *trespassing particles*. Now, from A we can easily produce a modified algorithm A' that only outputs the points that are occupied by one of the principal subsets of C_f, say, C'_f.

Furthermore, given A', we can construct another algorithm A'' that "traces" C'_f. That is, A'' output the points of C'_f in a specific order: it starts generating the points of C'_f that lie next to a principal ray, from the closest to the principal ray's endpoint to the farthest, and then proceeds to the next ray parallel to the principal ray, and so on (each of these rays is called a *scanline*). This is done by taking every point p on every scanline, in order, and executing A'. If A' generates p, then p is generated by A''. If A' generates points that appear on the same scanline but after p, then A'' proceeds along the scanline by one step and executes A' again, etc. Otherwise, A'' moves to the first point of the next scanline, etc. As soon as A' generates only points that lie on already-visited scanlines, A'' terminates. Hence, A'' is a procedure that terminates in a finite amount of time.

As A'' executes A', it can also detect if each location p it generates should be occupied by a single particle or by an expanded particle (indeed, this information is part of the description of C'_f). If it is an expanded particle, and the other location p' occupied by the same particle is on the next scanline, A'' does not generate p: it will generate only p' as soon as it scans it. Along with p', A'' will also generate a "direction of expansion", which goes from p' to p. This is called the *delayed-deployment rule*, and its purpose will be explained later.

An improvement we can make on A'' is that it not only generates the locations of the particles of C'_f, but it also generates the "movements" it makes through the plane to generate them (we introduced these *movement operations* in Sect. 3). Specifically, the possible movement operations are: (1) advance by one step along the current scanline; (2) move to the next scanline; (3) go back by one step along the current scanline; (4) deploy a contracted particle; (5) deploy a particle that will expand in direction d; (6) terminate.

The third type of operation is repeatedly used right after moving to the next scanline, in order to reach its endpoint e and then resume scanning. The only caveat is that, if e should be occupied by a trespassing particle belonging to $C_f \setminus C'_f$, it is skipped by A'', and the scan will not be resumed from e but from the location right before it. This type of behavior is called *trespasser-avoidance rule*, and is illustrated in Fig. 4.

Since we have an algorithm A'', there exists a Turing Machine T that computes the same function: it takes an integer n as input, and it outputs a sequence of movement operations that trace (a sector of) S_n. From Sect. 3 we know that there is a RAM Q_T with c registers that produces the same output when the number n is initially stored in its first register. We can then construct the mobile mRAM U_T, with $c' = 6$ flag registers, each of which corresponds to one of the movement operations above, and is set and reset whenever the corresponding operation has to be performed.

Finally, we can construct the mRAM Q'_{U_T}, which simulates U_T using only two registers, provided that the value 2^n is initially stored in its first register. This mRAM will be simulated by each leader in the system and its trail of particles. In the rest of this section we will show how the RAM simulator of Sect. 3 can be expanded to implement the movement operations above and therefore yield a Shape-Formation algorithm.

5.3   Initializing the Machine

Let us focus on a single leader particle L and its trail of n/k particles. Recall that the leader L is now located at the endpoint of its trail of particles that is farthest from the center of C_0. We want the last four particles on the trail, including L, to start executing the simulator of Q'_{U_T} as described in Sect. 3. First, however, it is necessary to initialize the simulator by storing the value 2^n in its first register, which amounts to placing particle M_1 at a distance of $2^n + 2$ steps away from particle M_2 (refer to Fig. 2).
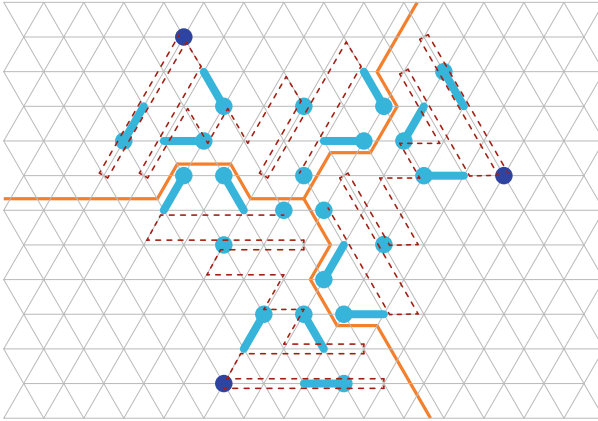
Fig. 4. A configuration forming the nth level of a shape, subdivided into its three principal subsets, each having two trespassing particles. The dashed lines represent the order in which the principal subsets are traced by the algorithm A''. The dark particles represent the last dots of their respective principal subsets.

At first L switches states with its only neighbor, which now becomes the new leader L. The two neighbors of L become M_1 and M_2, and then M_2 sends a message to its other neighbor, which becomes the pivot P.

Then, L has to "push" M_1 exactly 2^n times in order to create the input to the simulator. By "push" we mean that L reaches M_1, orders it to move forward by one step, and waits until M_1 has expanded and then contracted again.

In order to repeat this action the right number of times, the entire trail counts in binary from 0 to $2^n - 1$: this is done by letting each particle hold k binary digits, initially all set to 0. After each push, M_1 adds 1 to the binary number formed by its k digits. L reaches M_1 and takes the carry bit of the addition from M_1. If the carry bit is 1, L adds 1 to its own k digits. If the carry bit of this addition is 1, L moves all the way to M_2 and orders it to increment its k bits by 1, etc. As soon as a carry bit is 0, a message is sent back to L and forwarded by the trail's particles. When the message reaches L, it proceeds with the next push operation, etc.

When the last particle in the trail gets a carry bit of 1, it knows that M_1 has been pushed exactly 2^n times, and it forwards this information to L, which therefore stops pushing and proceeds with the simulation.

5.4   Tracing the Shape

Let the last particle in the trail be called the *rear particle*. The leader will coordinate the simulation of the mRAM that traces the shape pretending that the "pen" is held by the rear particle. That is, the simulator will go on with the computation until a movement operation is reached. If such operation is of type 4 or 5 (i.e., the deployment of a particle), then the rear particle is "dropped", which

corresponds to drawing a point of the shape.As explained in Sect. 3, L knows if a movement operation has to be performed by checking, at the end of every simulated instruction of the mRAM, if the value of the first register is a multiple of some prime number and the value of the second register is 0. This implies, in particular, that when a movement operation is executed, the particles P and M_2 are next to each other.

In the following we will explain in greater detail how the system behaves to implement the six possible movement operations.

1. Advancing by one step means that the four-particle mRAM simulator, as well as the whole trail of particles up to the rear particle, has to move by one step along the current scanline. First, L moves next to M_1 and orders it to advance by one step, waiting until it has moved. Then it goes next to M_2 and gives it the same order. L then moves away from M_2 by one step and waits for it. In the meantime, M_2 gives the same order to P and then advances toward L. P does the same as M_2, and so on. When the rear particle is reached by the message, it advances and sends an "all done" message to its predecessor, which forwards it to L. Then, L knows that it may proceed with the simulation.

2. Moving to the next scanline means that the entire trail and four-particle mRAM simulator must move "sideways" by one step, in order to relocate themselves on the next scanline s. The structure of this operation is similar to the previous one. L moves to M_1 and orders it to move sideways onto s. There can be no misunderstanding on the direction of movement, since we are assuming that all particles have already reached an agreement on a common handedness. L waits until M_1 has moved, and then goes to M_2 and gives it the same order. M_2 forwards the order to P, which forwards it to the next particles, etc. When the message reaches the rear particle, it forwards an "all done" message to L, and then moves onto s. When any particle receives the "all done" message, it waits until its predecessor has moved onto s, then it forwards it to its successor and moves onto s. When L has finally moved onto s, the procedure is over.

3. To go back along the current scanline, the operations of item 1 are executed in reverse order. That is, L moves to M_2 and forwards a message to the rear particle, which then moves backwards, and all other particles up to L follow one by one. Then L goes to M_1 and orders it to move backwards.

4. Deploying a contracted particle means that the rear particle has to stop where it is and remain there forever. This is simple to accomplish: L goes to M_2 and forwards a message to the rear particle. When the rear particle gets the message, it orders its predecessor to become the new rear particle and forward an "all done" message to L. From this point on, the old rear particle will stop following the trail, and no other particle will ever communicate with it again.

5. The deployment of a particle that will expand in direction d is similar to the previous item. The only difference is that, after the rear particle has transferred its role to its predecessor, it also expands in direction d. The direction is encoded by taking the forward direction of the trail as 0 and

numbering all other directions from 1 to 5 in clockwise order. There can be no misunderstanding, since we are assuming that all particles have already reached an agreement on a common handedness.

6. When the termination operation is reached, the simulation is over, and a *dismantling procedure* has to be executed, which will be described next.

5.5   Dismantling the Machine

There is one last problem to solve: when the pivot P has been deployed, only three particles are left to deploy: L, M_1, and M_2. Since three particles are too few to simulate a tracing RAM, they have no way to reach their final locations without "losing the way". It is impossible to adapt our algorithm without making an extra assumption on the shape to be formed.

Recall that the algorithm A'' generates all the movement operations that are necessary to draw the principal subset C'_f of the configuration C_f, by tracing it one scanline at a time. The last point plotted by A'' on the last scanline visited is said to be the *last dot* of C'_f (see Fig. 4). A subset of the grid graph G is called a *neighborhood* of point p with radius r if it contains precisely all the vertices of G that have distance at most r from p.

Assumption 1. *For each level S_n of the shape, there exists a configuration C_f of n particles that forms S_n (such that C_f is unbreakably k-symmetric if S_n has to be formed from an unbreakably k-symmetric initial configuration, cf. Proposition 1) and, for each principal subset C'_f, there exists a neighborhood of the last dot, with radius independent of n, that contains at least four particles of C'_f.*

If we make Assumption 1 on the shape to be formed, we can complete our algorithm with a dismantling procedure, which makes L, M_1, M_2, and P reach their final positions without getting lost in the attempt. To this end, we must first modify the tracing algorithm A'' that we previously designed, and produce a new tracing algorithm A'''. Let p_1 be the particle occupying the last dot of C'_f, and let p_2, p_3, p_4 be the three particles of C'_f closest to p_1 (other than p_1 itself). Ties are broken arbitrarily. The new algorithm A''' proceeds exactly like A'', except that it skips the deployment operations (i.e., the operations of type 4 or 5) for p_1, p_2, p_3, p_4, and terminates when the "pen" reaches p_1.

Then we construct another algorithm F that, with input n, generates the paths that three particles have to take from p_1 to reach the locations of p_2, p_3, p_4, making sure to avoid the locations of the other particles of C_f (this is possible because we chose p_2, p_3, p_4 to be closest to p_1). Additionally, if some of the p_is are expanded, F outputs this information, as well as the direction of expansion. It is clear that F is computable and terminates in a finite amount of time.

As before, we observe that there is a Turing machine T that executes A''', and a RAM Q_T that simulates T. However, this time we add an extra register to Q_T, which will store the input value n and will never erase it: this value will be passed as input to F. Again, the mobile mRAM U_T is constructed, which is simulated by a 2-register RAM Q'_{U_T}, which in turn is simulated by particles L,

M_1, M_2, and P. When these four particles are done executing A''', they erase all registers except the one containing n, and execute a simulation of F (again, by simulating a 2-register RAM that executes F). Whenever F outputs a step in a path from p_1 to p_2, p_3, or p_4, this step is memorized by the leader L. At the end of the simulation, L has the three complete paths stored in its memory. Note that this is possible even if the memory of L is of constant size: by Assumption 1, the three paths are contained in a neighborhood of p_1 with radius independent of n, and therefore their length is bounded by a constant.

When the simulation of F is terminated, L starts moving toward P, and orders M_1 and M_2 to do the same. When M_1, L, M_2, and P are in four consecutive locations, L communicates the three paths to the others. At this point, each of the four particles knows what to do to reach its final position, and they all do so in an orderly fashion.

5.6   Conclusion

Our Shape-Formation algorithm demonstrates the following:

Theorem 2. *Under Assumption 1, any computable shape is formable from any simply connected initial configuration.*

The proof of correctness is straightforward, and it relies on two basic facts:

– A four-particle simulator and its trail of particles will never get in the way of other four-particle simulators. This is because each of them stays in its own principal sector; the only exceptions are the trespassers, which are actually not an obstacle due to the trespasser-avoidance rule.
– The particles that have already been deployed will not get in the way of the four-particle simulator that deployed them. This is because the movement operations always make the simulator travel through locations where no particle has been deployed, yet. In particular, the delayed-deployment rule serves this purpose: a deployed particle will always expand toward a location that will never be traversed by the simulator again.

Note that Assumption 1 only excludes shapes whose levels are very sparse around the last dots of their principal subsets. In particular, connected shapes abundantly satisfy the assumption, which in this case reduces to the necessary condition of Proposition 1. Therefore, we have the following characterization of formable connected computable shapes:

Corollary 1. *A necessary and sufficient condition for a connected computable shape to be formable from a simply connected initial configuration is that, if the initial configuration is unbreakably k-symmetric, then also the corresponding level of the shape is unbreakably k-symmetric.*

Our results open several interesting questions. In particular: what happens when the starting configuration is not simply connected? To what other problems can our mobile RAM be applied?

References

1. Daymude, J., Hinnenthal, K., Richa, A., Scheideler, C.: Computing by programmable particles. In: Flocchini, P., Prencipe, G., Santoro, N. (eds.) Distributed Computing by Mobile Entities. Lecture Notes in Computer Science, vol. 11340, pp. 615–681. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11072-7_22

2. Derakhshandeh, Z., Gmyr, R., Richa, A., Scheideler, C., Strothmann, T.: An algorithmic framework for shape formation problems in self-organizing particle systems. In: Proceedings of NanoCom, pp. 21:1–21:2 (2015)

3. Derakhshandeh, Z., Gmyr, R., Richa, A., Scheideler, C., Strothmann, T.: Universal shape formation for programmable matter. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 289–299 (2016)

4. Derakhshandeh, Z., Gmyr, R., Richa, A., Scheideler, C., Strothmann, T.: Universal coating for programmable matter. Theor. Comput. Sci. **671**, 56–68 (2017)

5. Derakhshandeh, Z., Gmyr, R., Strothmann, T., Bazzi, R., Richa, A., Scheideler, C.: Leader election and shape formation with self-organizing programmable matter. In: Proceedings of the International Conference on DNA Computing and Molecular Programming, pp. 117–132 (2015)

6. Di Luna, G.A., Flocchini, P., Santoro, N., Viglietta, G., Yamauchi, Y.: Shape formation by programmable particles. Distrib. Comput. **33**(1), 69–101 (2019). https://doi.org/10.1007/s00446-019-00350-6

7. Luna, G.D., Flocchini, P., Prencipe, G., Santoro, N., Viglietta, G.: Line recovery by programmable particles. In: Proceedings of the International Conference on Distributed Computing and Networking (2018)

8. Michail, O., Skretas, G., Spirakis, P.: On the transformation capability of feasible mechanisms for programmable matter. In: Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP), pp. 136:1–136:15 (2017)

9. Minsky, M.: Computation: Finite and Infinite Machines. Prentice-Hall Inc., Upper Saddle River (1967)

10. Naz, A., Piranda, B., Bourgeois, J., Goldstein, S.: A distributed self-reconfiguration algorithm for cylindrical lattice-based modular robots. In: Proceedings of the IEEE International Symposium on Network Computing and Applications, pp. 254–263 (2016)

11. Patitz, M.J.: An introduction to tile-based self-assembly and a survey of recent results. Nat. Comput. **13**(2), 195–224 (2013). https://doi.org/10.1007/s11047-013-9379-4

12. Romanishin, J.W., Gilpin, K., Rus, D.: M-blocks: Momentum-driven, magnetic modular robots. In: Proceedings of the International Conference on Intelligent Robots and Systems, pp. 253–263 (2013)

13. Rothemund, P.: Folding DNA to create nanoscale shapes and patterns. Nature **440**(7082), 297–302 (2006)

14. Rubenstein, M., Ahler, C., Nagpal, R.: Kilobot: a low cost scalable robot system for collective behaviors. In: Proceedings of the International Conference on Robotics and Automation, pp. 3293–3298 (2012)

15. Toffoli, T., Margolus, N.: Programmable matter: concepts and realization. Phys. D **47**(1), 263–272 (1991)