Convex hull formation for programmable matter

Convex Hull Formation for Programmable Matter

Joshua J. Daymude[1]
Computer Science, CIDSE, Arizona State University, Tempe, AZ, USA
jdaymude@asu.edu
0000-0001-7294-5626

Robert Gmyr
Department of Computer Science, University of Houston, Houston, TX, USA
rgmyr@uh.edu

Kristian Hinnenthal
Department of Computer Science, Paderborn University, Paderborn, Germany
krijan@mail.upb.de
0000-0001-9464-295X

Irina Kostitsyna
Department of Mathematics and Computer Science, TU Eindhoven, the Netherlands
i.kostitsyna@tue.nl
0000-0003-0544-2257

Christian Scheideler
Department of Computer Science, Paderborn University, Paderborn, Germany
scheidel@mail.upb.de

Andréa W. Richa[2]
Computer Science, CIDSE, Arizona State University, Tempe, AZ, USA
aricha@asu.edu

Abstract

We envision *programmable matter* as a system of nano-scale agents (called *particles*) with very limited computational capabilities that move and compute collectively to achieve a desired goal. We use the *geometric amoebot model* as our computational framework, which assumes particles move on the triangular lattice. Motivated by the problem of *shape sealing* whose goal is to seal an object using as little resources as possible, we investigate how a particle system can self-organize to form an object's convex hull. We give a fully distributed, local algorithm for convex hull formation and prove that it runs in $\mathcal{O}(B + H \log H)$ asynchronous rounds, where B is the length of the object's boundary and H is the length of the object's convex hull. Our algorithm can be extended to also form the object's ortho-convex hull, which requires the same number of particles but additionally minimizes the enclosed space within the same asymptotic runtime.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Computational geometry; Computing methodologies → Self-organization.

Keywords and phrases programmable matter; self-organization; convex hull; ortho-convex hull; distributed algorithms; computational geometry

[1] Supported in part by NSF Awards CCF-1637393 and CCF-1733680.
[2] Supported in part by NSF Awards CCF-1637393 and CCF-1733680.

1 Introduction

In recent years, research in *self-organizing programmable matter* has become increasingly popular in many fields with potential for broad applications. The vision is to create a system that can change its physical properties like shape, density, conductivity, or color in a programmable fashion based on either user input or autonomous sensing. As a prominent example, works in *molecular self-assembly* (e.g., [9, 18, 21]) have achieved self-organizing structures at the nanoscale using DNA "tiles" as fundamental building blocks. Such passive approaches, which utilize elements that rely strictly on interactions with their environment to self-organize, can be contrasted with approaches utilizing active elements (e.g., [5, 14, 22]). In this paper, we use the *geometric amoebot model* [4, 5] as our computational framework, in which nanoscale agents with limited computational capabilities (called *particles*) move on the triangular lattice and exchange information in order to collectively achieve a given goal without any outside intervention.

The study of programmable matter can be motivated, for instance, by applications in the medical area. Here, one could envision small particles to locate and repair small wounds in the human body by building structures or coating surfaces. Furthermore, particles might be employed to capture harmful substances or cells. Outside of medicine, programmable matter could also be used to repairing small tubes or explore arduous or difficult terrain, for example. Under various theoretical models, classical problems such as *shape formation* (e.g., [6, 14, 18, 22]), *coating* (e.g., [2, 7]), and *shape recognition* [13] have recently been investigated in the context of programmable matter. Somewhat in between these lies the problem of *shape sealing*, in which the goal is to seal an object, i.e., enclose it by a cycle of particles. Apart from minimizing the runtime to solve sealing problems, another intriguing question is how to minimize the required number of particles, a problem directly related to forming an object's *convex hull*. Note that an object's convex hull is at most as large as its surface, but in many cases can be significantly smaller. Although computing convex hulls has been investigated extensively in computational geometry and distributed computing, to the best of our knowledge, it has never been rigorously studied for programmable matter.

1.1 The Amoebot Model

In the *amoebot model*, originally proposed in [5] and described in full[1] in [4], programmable matter consists of individual, homogeneous computational elements called *particles*. In its geometric variant, the underlying geometry is the infinite triangular lattice[2] $G_\Delta = (V, E)$ (see Fig. 1a). Each particle occupies either a single node in V (i.e., it is *contracted*) or a pair of adjacent nodes in V (i.e., it is *expanded*), as in Fig. 1b. Particles move via a series of *expansions* and *contractions*: a contracted particle can expand into an unoccupied adjacent node to become expanded, and completes its movement by contracting to once again occupy a single node. For an expanded particle, we refer to the node it last expanded into as its *head* and the other node it occupies as its *tail*.

Two particles occupying adjacent nodes are said to be *neighbors*. Neighboring particles can coordinate their movements in a *handover*, which can occur in one of two ways: a contracted particle P can "push" an expanded neighbor Q by expanding into a node occupied by Q, forcing it to contract, or an expanded particle Q can "pull" a contracted neighbor P

[1] We omit details which are not necessary for convex hull formation.
[2] Our past works refer to G_Δ as the *equilateral triangular grid graph* G_{eqt} or the triangular lattice Γ.

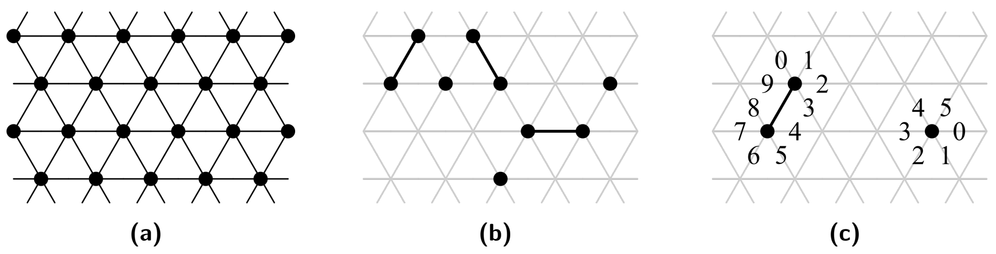


Figure 1 (a) A section of the triangular lattice G_Δ; nodes of V are shown as black circles and edges of E are shown as black lines. (b) Expanded and contracted particles; G_Δ is shown as a gray lattice, and particles are shown as black circles. Particles with a black line between their nodes are expanded. (c) Two particles with different offsets for their port labels.

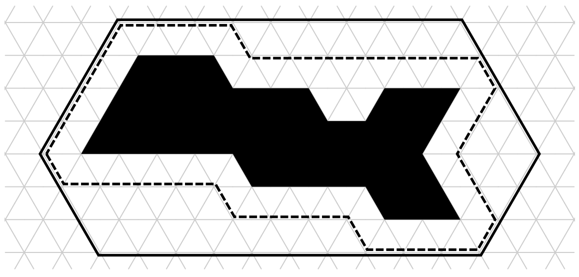
by contracting, forcing P to expand into the node it is vacating.

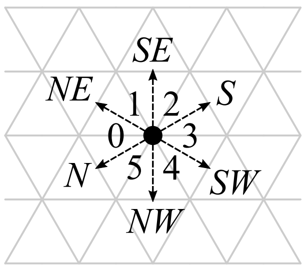
Each particle keeps a collection of ports — one for each edge incident to the node(s) it occupies — that have unique labels from its own local perspective. Although each particle is anonymous, lacking a unique identifier, a particle can locally identify any given neighbor by its labeled port corresponding to the edge between them. 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 share a coordinate system or global compass and may have different offsets for their port labels (see Fig. 1c). Each particle has a constant-size local memory which it and its neighbors can directly read from and write to for communication. However, particles do not have any global information and — due to the limitation of constant-size memory — cannot locally store the total number of particles in the system nor any estimate of this value.

We assume the standard asynchronous model of computation from distributed computing (see, e.g., [16]), where a system progresses through *atomic actions*. A classical result under this model states that for any concurrent asynchronous execution of atomic actions, there exists a sequential ordering of actions producing the same end result, provided conflicts that arise in the concurrent execution are resolved. In the amoebot model, an atomic action corresponds to a single particle activation in which a particle can perform an arbitrary, bounded amount of computation involving its local memory and the memories of its neighbors and at most one expansion or contraction. We assume conflicts involving concurrent memory writes or simultaneous particle expansions into the same unoccupied node are resolved arbitrarily such that at most one particle is writing into a given memory location or expanding into a given node at a time. Thus, while in reality many particles may be active concurrently, it suffices when analyzing our algorithms to consider a sequence of activations where only one particle is active at a time. We assume the resulting activation sequence is *fair*: for any inactive particle P at time t, P will be activated again at some time $t' > t$. An *asynchronous round* is complete once every particle has been activated at least once.

1.2 Problem Description

An instance of the *(Ortho-)Convex Hull Formation Problem* is $(\mathcal{P}, O)$ where $\mathcal{P}$ is a finite, connected system of initially contracted particles and $O \subset V$ is a finite and simply connected set of nodes representing the *object*. We further assume that the particle system $\mathcal{P}$ contains


(a)



(b)

Figure 2 (a) An example of an object (black), enclosed by its convex hull (solid line) and its ortho-convex hull (dashed line). (b) A particle's local labeling of the six half-planes composing the convex hull: the half-plane between its local 0 and 5-labeled edges is N, and the remaining half-planes are labeled accordingly.

a unique leader particle[3] ℓ initially adjacent to O, and that O does not contain any *tunnels* of width 1; that is, the graph induced by $V \setminus V(O)$ is 2-connected.

Let $O^* \subset V$ denote the *convex expansion* of O, that is, the minimal convex set of nodes such that O is completely contained in O^*. We define the *convex hull*[4] of O, denoted $H(O)$, to be the set of nodes in $V \setminus O^*$ that are adjacent to the nodes of O^* (see Fig. 2a). Following the definition of Fink and Wood [10], we define a set of nodes $S \subset V$ to be *ortho-convex* if for every connected straight line of nodes $L \subset V$, $L \cap S$ is either empty or connected[5]. We can define the *ortho-convex expansion* $O_o^* \subset V$ and the *ortho-convex hull* of O, denoted $H_o(O)$, analogously: O_o^* is the minimal ortho-convex set of nodes such that O is completely contained in O_o^*, and $H_o(O)$ is the set of nodes in $V \setminus O_o^*$ adjacent to the nodes of O_o^*. Note that an object's ortho-convex hull is always contained in its convex hull.

Given an instance $(\mathcal{P}, O)$ of the (Ortho-)Convex Hull Formation Problem with $|\mathcal{P}| \geq |H(O)| = |H_o(O)|$, the goal is to reconfigure $\mathcal{P}$ so that every node of the (ortho-)convex hull of O is occupied by a contracted particle.

1.3 Our Contributions

We present a fully distributed, local algorithm for the Convex Hull Formation Problem that runs in $\mathcal{O}(B + H \log H)$ asynchronous rounds, where B is the length of the object's boundary and $H = |H(O)|$ is the length of the object's convex hull. We present the algorithm in two parts: Section 2 describes how a single particle with unbounded memory can find the convex hull, and Section 3 shows how a system of particles with bounded memory can emulate this single-particle algorithm. We then present several extensions to our algorithm in Section 4. We first extend our algorithm to also solve the Ortho-Convex Hull Formation Problem in an additional $\mathcal{O}(H)$ asynchronous rounds. Finally, we show that the assumptions that O does not have any 1-width tunnels and that $|\mathcal{P}| \geq H$ can both be lifted; that is, we can still form an object's convex hull even if it has narrow tunnels and can partially form it if we have insufficient particles.

[3] Such a particle can be determined in $\mathcal{O}(n)$ asynchronous rounds with high probability using a slightly modified version of the leader election algorithm in [3], where $n = |\mathcal{P}|$ is the number of particles and by *with high probability* we mean with failure probability upper bounded by $1/\text{poly}(n)$.

[4] Note that our definition of a convex hull differs slightly from the classical geometric definition, in which the convex hull of O would be the nodes of O^* adjacent to $V \setminus O^*$.

[5] The ortho-convex hull is often also referred to as the *$\mathcal{O}$-convex hull* or *orthogonal convex hull*.

1.4 Related Work

The convex hull problem is arguably one of the best-studied problems in computational geometry (see, e.g., [1]), and various parallel algorithms have been proposed to solve it. The best result known for the standard representation of the convex hull is an $\mathcal{O}(\log n/\log\log n)$ time algorithm on $\mathcal{O}(n\log\log n/\log n)$ processors [11], and the best result known for non-standard convex hull representations is an $\mathcal{O}(\log^* n)$ time algorithm on $\mathcal{O}(n/\log^* n)$ processors [12]. Both algorithms make excessive use of the power of a CRCW PRAM and are very complex, so it is not known whether there is any way of applying them to programmable matter.

Only a few distributed algorithms are known for the convex hull problem. An example is a distributed algorithm for the geometric ring (i.e., processes with geographic locations that form a ring) that can find a convex hull with $\mathcal{O}(n\log^2 n)$ messages [20]. Miller and Stout [17] present polylogarithmic algorithms for various interconnection networks such as a hypercube under the assumption that the input points are ordered by their x-coordinates. A result for arbitrary interconnection networks was presented by Diallo et al. [8]. For p processors and n points, where $n \gg p$, their algorithm has a runtime of $\mathcal{O}(T_{\text{sequential}}/p + T_s(n,p))$, where $T_s(n,p)$ is the time needed to sort n data with the given interconnection network.

Naturally, all of the parallel and distributed algorithms presented so far require the coordinates of the nodes to be known, and typically use a divide-and-conquer approach. However, as our particles only have a finite memory, and we might have only very few particles compared to the size of the object's boundary, it is impossible for the particle system to store geographic locations on the boundary, let alone order them to employ a divide-and-conquer approach. Although the convex hull problem has been extensively studied in the distributed setting, to the best of our knowledge, there only exist sequential algorithms to compute the ortho-convex hull (see, e.g., [15] and the references therein).

2 The Single-Particle Algorithm

We first consider a particle system composed of a single particle P with unbounded memory, and present an algorithm for accurately estimating the convex hull of O. We define the *boundary* $B(O)$ of O as the set of all positions in $V \setminus O$ that are adjacent to a position in O, and assume P is initially placed somewhere on $B(O)$. The main idea of this algorithm is to let P perform a clockwise traversal of $B(O)$, internally maintaining an estimate of the convex hull.

In particular, the convex hull can be represented as the intersection of six half-planes $\mathcal{H} = \{N, NE, SE, S, SW, NW\}$, which P can label using its local compass as in Fig. 2b. Particle P estimates the location of these half-planes by maintaining six counters $\{d_h : h \in \mathcal{H}\}$, where each counter d_h represents the L_1-distance[6] from the position of P to half-plane h. If at least one of these counters is equal to 0, P is on the current estimate of the convex hull.

Each counter is initially set to 0, and P updates them as it moves. Let $[6] = \{0,\ldots,5\}$ denote the six directions P can move in, corresponding to its contracted port labels. At any time, P behaves as follows. It first computes the direction $i \in [6]$ to move toward following the right-hand rule, which results in a clockwise traversal of $B(O)$. Note that i is unique if the object O does not have a tunnel of width 1, i.e., its boundary does not intersect itself. It then updates its distance counters by setting $d_h \leftarrow \max\{0, d_h + \delta_{i,h}\}$ for all $h \in \mathcal{H}$, where δ_i

[6] The L_1-distance between two nodes on G_Δ is the number of edges in a shortest path between them. Analogously, the L_1-distance between a node and a half-plane is the number of edges in a shortest path between the node and any node on the half-plane.

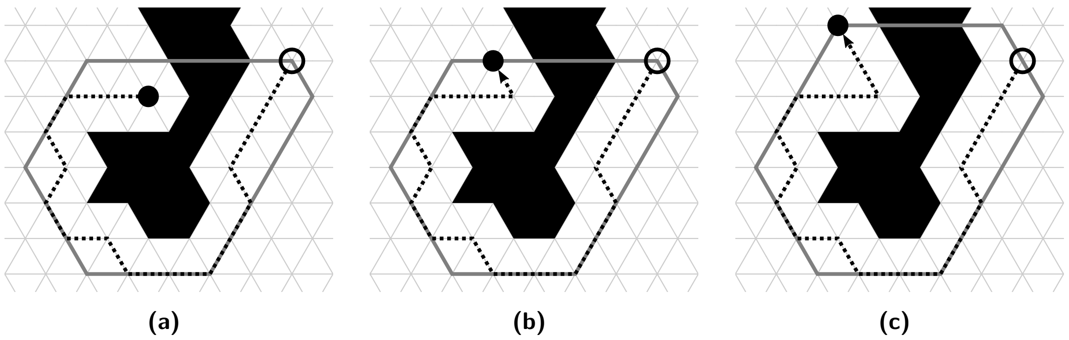


Figure 3 An illustration of the movement of a particle (black dot) with its current convex hull estimate (gray line) after having traversed the path (dotted line) from its starting point (black circle). (a) $d_h \geq 1$ for all $h \in \mathcal{H}$, therefore the particle's next move does not push any half-plane. (b) $d_N = 0$ and the particle's next move is in global NW direction. (c) Half-plane N has been pushed.

is defined as follows:

$$\delta_* = (N, NE, SE, S, SW, NW)$$
$$\delta_0 = (1,1,0,-1,-1,0) \qquad \delta_1 = (0,1,1,0,-1,-1) \qquad \delta_2 = (-1,0,1,1,0,-1)$$
$$\delta_3 = (-1,-1,0,1,1,0) \qquad \delta_4 = (0,-1,-1,0,1,1) \qquad \delta_5 = (1,0,-1,-1,0,1)$$

Thus, every movement decreases the distance counters of the two half-planes to which it gets closer, and increases the distance counters of the two half-planes from which it gets farther away. Whenever P moves toward a half-plane to which its distance is already 0, the value stays 0, essentially "pushing" the estimation of the half-plane one step further. An example of such a movement is given in Fig. 3.

Finally, P needs to detect when its convex hull estimate matches the actual convex hull. To do so, it stores six terminating bits $\{b_h : h \in \mathcal{H}\}$, where a terminating bit b_h is equal to 1 if P has visited half-plane h (i.e., if its distance to h has been 0) since it last pushed any half-plane, and 0 otherwise. Whenever P moves without pushing a half-plane, it sets $b_h = 1$ for all h such that $d_h = 0$ after the move. Otherwise, it sets $b_h = 0$ for all h. If after a move all six terminating bits are 1, P contracts and terminates.

Analysis

Before we show how this single-particle algorithm can be emulated by a system of particles with bounded memory, we analyze its correctness and runtime. Note that, when only one particle is in the particle system, each particle activation is also an asynchronous round. For a given round i, let $H_i(O) \subset V$ be the set of all nodes enclosed by P's estimate of the convex hull of O after round i, i.e., all nodes in the closed intersection of the six half-planes. We first show that P's estimate of the convex hull represents the correct convex hull after at most one traversal of the object's boundary, and does not change afterwards.

▶ **Lemma 1.** *If P completes its traversal of the object's boundary in round i^*, $H_i(O) = O^*$ for all $i \geq i^*$.*

Proof. First, note that since the particle exclusively walks on the boundary of O, $H_i(O) \subseteq O^*$ for all rounds i. Furthermore, $H_i(O) \subseteq H_{i+1}(O)$ for any round i. Finally, once the particle has traversed the whole boundary, it has visited a node of each half-plane corresponding to O^*, and thus $H_{i^*}(O) = O^*$. ◀

We now show P finishes if and only if the convex hull estimate is correct.

▶ **Lemma 2.** *Suppose $H_i(O) \subset O^*$ after some round i. Then $b_h = 0$ for some half-plane h.*

Proof. Suppose to the contrary that after round i, $H_i(O) \subset O^*$ but $b_h = 1$ for all $h \in \mathcal{H}$; let i be the first such round. First, note that $|H_i(O)| \geq 1$, since in order for the particle to set b_h to 1 it must have had a distance of at least 1 to half-plane h after some round $i' < i$. Therefore, at least 3 different half-planes must have been pushed already. Since i is the first round such that $b_h = 1$ for all h, the particle must have just reached a half-plane (w.l.o.g., say N) after having visited the half-planes NE, SE, S, SW, and NW in that order and without pushing any half-planes in between visits. However, since it must have pushed at least 3 half-planes already, it must have previously visited the NW half-plane, and therefore it must have traversed the whole boundary already. This leads to a contradiction by Lemma 1. ◀

▶ **Lemma 3.** *Suppose $H_i(O) = O^*$ for the first time after some round i. Then P terminates at some node of the convex hull after at most one traversal of the boundary.*

Proof. Since i is the first round to satisfy $H_i(O) = O^*$, particle P must have just reached a node u with distance 0 to a half-plane, w.l.o.g., say N. Since P will no longer push any half-planes and the next visited node must also have distance 0 to half-plane N, $b_N = 1$ after the next move. Subsequently, P will visit every other half-plane without pushing any of them, so every bit b_h will be set to 1 before P visits u again. Particle P sets its last terminating bit, say $b_{h'}$, once it visits a node v with distance 0 to h' for the first time; therefore, P terminates at $v \in B(O) \cap H(O)$. ◀

The previous lemmas immediately imply the following theorem. Let $B = |B(O)|$.

▶ **Theorem 4.** *The single-particle algorithm terminates after $t^* = \mathcal{O}(B)$ asynchronous rounds with particle P at a node $u \in B(O) \cap H(O)$ and $H_{t^*}(O) = H(O)$.*

3 The Convex Hull Algorithm

Next we show how a system of n particles each with only constant-size memory can emulate the single-particle algorithm of Section 2. Recall that we assume $n = |\mathcal{P}|$ is sufficiently large to form the convex hull of O and that a unique leader particle ℓ adjacent to the object has already been elected. This leader ℓ is primarily responsible for emulating the particle with unbounded memory in the single-particle algorithm. To do so, it utilizes the other particles in the system as distributed memory. More precisely, as ℓ moves, it will create a line of particles behind it that will be used to store the distances d_h from ℓ to half-plane h as binary counters. Once these measurements are complete, ℓ uses them to lead the other particles in forming the convex hull.

3.1 A Binary Counter of Particles

We begin by describing how to coordinate a particle system as a binary counter that supports increments and decrements by one as well as zero-testing. This description builds upon previous work on collaborative computation under the amoebot model, where an increment-only binary counter was detailed [19]. Suppose that the participating particles are organized as a simple path with the leader at its start: $\ell = P_0, P_1, P_2, \ldots, P_k$. Each particle P_i has a bit value $P_i.bit \in \{\emptyset, 0, 1\}$, where $P_i.bit = \emptyset$ implies P_i is not part of the counter; i.e., it is beyond the most significant bit. A *final token* f represents the end of the counter. Although not necessary for increments and decrements, utilizing f will allow the leader to zero-test the counter locally and efficiently. Thus, if the particle holding f is P_i with $0 < i \leq k$, then the counter value is represented by the bits of each particle from the leader ℓ (holding the least significant bit) up to and including P_{i-1} (holding the most significant bit).

The leader ℓ is responsible for initiating counter operations, but the rest of the particles will only need local information and communication to carry these operations out. To increment the counter, the leader ℓ simply generates an increment token c^+ (assuming it was not already holding a token). Now consider this operation from the perspective of any particle P_i holding a c^+ token, where $0 \leq i \leq k$. If $P_i.bit = 0$, P_i can simply consume c^+ and set $P_i.bit \leftarrow 1$. Otherwise, if $P_i.bit = 1$, this increment needs to be carried over to the next most significant bit. As long as P_{i+1} is not already holding a token, P_i can forward c^+ to P_{i+1} and set $P_i.bit \leftarrow 0$. Finally, if $P_i.bit = \emptyset$, this increment has been carried over past the counter's end, so P_i must also be holding the final token f. In this case, P_i simply forwards f to P_{i+1} and sets $P_i.bit \leftarrow 1$.

Decrements are similar; when considering this operation from the perspective of any particle P_i holding a c^- token, where $0 \leq i \leq k$, the cases for $P_i.bit \in \{0, 1\}$ are anti-symmetric to those for the increment, with two exceptions. First, we only allow P_i to consume c^- and set $P_i.bit \leftarrow 0$ if P_{i+1} is not also holding a c^-. While not necessary for the correctness of the decrement operation, this will enable conclusive zero-testing. Second, if $P_i.bit = 1$ and P_{i+1} is holding f, then P_i is the most significant bit. So this decrement shrinks the counter by one bit; thus, P_i consumes c^-, takes f from P_{i+1}, and sets $P_i.bit \leftarrow \emptyset$.

Finally, the zero-test operation: if P_1 is holding a decrement token c^- and $P_1.bit = 1$, ℓ cannot perform the zero-test conclusively. Otherwise, the counter value is 0 if and only if $\ell.bit = 0$, P_1 is holding the final token f, and P_1 is not holding an increment token c^+.

We use one optimization to overcome adversarial particle activation sequences in our asynchronous setting, improving the running time of processing counter operations. The algorithm remains conceptually the same, but we allow each particle to buffer up to two tokens in a queue instead of only storing one. Since queues are first-in-first-out, this does not change the behavior or correctness of our counters; thus, we opted for a simpler presentation for sake of clarity. We note that although other optimizations are possible — such as directly updating neighbor's bits instead of passing tokens (in some cases) or canceling c^+ and c^- tokens held by the same particle — they will not improve the asymptotic performance.

Correctness

We now show the *safety* of our increment, decrement, and zero-test operations for the distributed counter. More formally, we will show that given any sequence of these operations, our distributed binary counter will eventually yield the same values as a centralized counter.

If our distributed counter was fully synchronized, meaning at most one increment or decrement token is in the counter at a time, it is very easy to see that the distributed counter exactly mimics a centralized counter, but with a linear slowdown in the length of the binary counter. Our counter instead allows for many (potentially interleaved) increments and decrements to be processed in parallel. As long as the c^+ and c^- tokens are prohibited from overtaking one another, thereby altering the order the operations were initiated in, it is easy to see that the counter will correctly process as many tokens as there's capacity for.

So it remains to prove the correctness of the zero-test operation. We will prove this in two parts: first, we show the zero-test operation is always eventually available. We then show that if the zero-test operation is available, it is always reliable; i.e., it always returns an accurate indication of whether or not the counter's value is 0.

▶ **Lemma 5.** *If at time t zero-testing is unavailable (i.e., particle P_1 is holding a decrement token c^- and $P_1.bit = 1$) then there exists a time $t' > t$ when zero-testing is available.*

Proof. We'll argue by induction on i — the number of consecutive particles starting at some particle P_x that are holding c^- tokens and have their bits set to 1 — that there exists a time $t^* > t$ where P_x can consume c^- and set $P_x.bit \leftarrow 0$. If $i = 1$, then P_{x+1} is not holding another c^- and thus P_x can process its c^- at its next activation (say, at $t^* > t$).

Now suppose $i > 1$ and the induction hypothesis holds up to $i - 1$. Then at time t, every particle P_j with $x \leq j < x + i$ is holding a c^- token and has $P_j.bit = 1$. By the induction hypothesis, there exists a time $t_1 > t$ at which P_{x+1} is activated and can consume its c^- token, setting $P_{x+1}.bit \leftarrow 0$. So the next time P_x is activated (say, at $t^* > t_1$) it can do the same, consuming its c^- token and setting $P_x.bit \leftarrow 0$. This concludes our induction.

Suppose P_1 is holding a decrement token c^- and $P_1.bit = 1$ at time t, leaving the zero-test unavailable. Applying the above argument to P_1, there must exist a time $t^* > t$ such that P_1 can process its c^- and set $P_1.bit \leftarrow 0$. Since the increment and decrement tokens remain in order and can't skip over particles, P_1 will not be holding a c^- token when ℓ is next activated (say, at $t' > t^*$) allowing ℓ to perform a zero-test. ◀

▶ **Lemma 6.** *If the zero-test operation is available, then it reliably decides whether the counter's value is* 0*.*

Proof. We prove a stronger result than the one we use for leader zero-testing: if the zero-test operation is available, the value of the counter is $v = 0$ if and only if some particle P_{i+1} is holding the final token f but not an increment token c^+ and $P_i.bit = 0$. Argue by induction on the number of increment and decrement tokens to reach the most significant bit (MSB) of the counter. If no operations have been applied, then the counter has its initial value $v = 0$, P_1 holds f, and $\ell.bit = 0$.

Now suppose the induction hypothesis holds for the first t operations, and consider the $(t+1)$-th operation to reach the MSB. If $v = 0$ after the t-th operation, then by the induction hypothesis we have that P_{i+1} holds f but not a c^+ and $P_i.bit = 0$. The $(t+1)$-th operation cannot be a decrement since the counter does not support negative values, so it is an increment c^+ on the MSB, $P_i.bit$. When this operation is processed, $P_i.bit \leftarrow 1$ and c^+ is consumed, yielding $v = 1$. Thus $v > 0$ and $P_i.bit \neq 0$, so the property is satisfied. Otherwise, if $v > 0$ after the t-th operation, then assuming some P_{i+1} holds f, by the induction hypothesis either P_{i+1} is also holding a c^+, $P_i.bit = 1$, or both. In any case where P_{i+1} is also holding a c^+, this increment token will still be on P_{i+1} after P_i tries to process the $(t+1)$-th operation. We still have $v > 0$ and P_{i+1} is holding both f and a c^+ so the property is satisfied.

So it remains to consider when $v > 0$, P_{i+1} is only holding f, and $P_i.bit = 1$. If the $(t+1)$-th operation is an increment c^+ on the MSB, then P_i performs a carry over by setting $P_i.bit \leftarrow 0$ and passing c^+ to P_{i+1}. The value of the counter increased by one, so still $v > 0$, and now P_{i+1} is holding both f and a c^+, satisfying the property. Otherwise, the $(t+1)$-th operation is a decrement c^- on the MSB. If the MSB is held by ℓ (i.e., P_1 holds f and $\ell.bit = 1$), then applying c^- to ℓ results in the base case of our induction where $v = 0$. So suppose the MSB is held by $P_i \neq \ell$. Then $v > 1$; after this decrement is applied, the counter shrinks by one bit and $v > 0$. Applying c^-, $P_i.bit \leftarrow \emptyset$ and P_i takes f from P_{i+1}. So it suffices to show that $P_{i-1}.bit \neq 0$. For the c^- to have reached P_i, P_{i-1} must have carried it over, leaving $P_{i-1}.bit = 1$. If an increment c^+ reached P_{i-1} afterwards, it couldn't have been carried over (setting $P_{i-1}.bit = 0$) because P_i was already holding c^-. But a decrement c^-

couldn't have been applied to P_{i-1} either, since $P_{i-1}.bit = 1$ and P_i was holding another c^-. Therefore, at the time P_i takes f, $P_{i-1}.bit = 1$. ◀

Runtime

To analyze the runtime of our distributed binary counters, we use a *dominance argument* between asynchronous and parallel executions, originally applied to self-organizing particle systems in [2]. More specifically, we build upon the analysis of [19] where a dominance argument was used to bound the running time of an increment-only distributed counter. The general idea of the argument is as follows. First, we prove that the counter operations are, in the worst case, at least as fast in an asynchronous execution as they are in a simplified parallel execution. We then give an upper bound on the number of parallel rounds required to process these operations; combining these two results also gives a worst case upper bound on the running time in asynchronous rounds.

Let a configuration C of the distributed counter encode each particle's bit value and any increment or decrement tokens it might be holding. A configuration is *valid* if there is exactly one particle (say, P_i) holding the final token f, $P_j.bit = \emptyset$ if $j \geq i$ and $P_j.bit \in \{0, 1\}$ otherwise, and if a particle P_j is holding a c^+ or c^- token, then $j \leq i$. A *schedule* is simply a sequence of configurations $(C_0, \ldots, C_t)$. Let S be a sequence of m total increment, decrement, and empty (do nothing) operations that is *nonnegative*, i.e., for all $0 \leq i \leq m$, the first i operations have at least as many increments as decrements.

▶ **Definition 7.** A *parallel counter schedule* $(S, (C_0, \ldots, C_t))$ is a schedule $(C_0, \ldots, C_t)$ such that each configuration C_i is valid and, for every $0 \leq i < t$, C_{i+1} is reached from C_i by satisfying the following for each particle P_j:

1. If $j = 0$, then $P_j = \ell$ generates the next operation according to S.
2. P_j is holding c^+ in C_i and either $P_j.bit = 0$, causing P_j to consume c^+ and set $P_j.bit \leftarrow 1$, or $P_j.bit = \emptyset$, causing P_j to additionally give the final token f to P_{j+1}.
3. P_j is holding c^- and has $P_j.bit = 1$ in C_i, so P_j consumes c^-. If P_{j+1} is holding f in C_i, P_j takes f from P_{j+1} and sets $P_j.bit \leftarrow \emptyset$; otherwise it simply sets $P_j.bit \leftarrow 0$.
4. P_j is holding c^+ and has $P_j.bit = 1$ in C_i, so P_j forwards c^+ to P_{j+1} and sets $P_j.bit \leftarrow 0$.
5. P_j is holding c^- and has $P_j.bit = 0$ in C_i, so P_j forwards c^- to P_{j+1} and sets $P_j.bit \leftarrow 1$.

Such a schedule is said to be *greedy* if the above actions are taken whenever possible.

Using the same sequence of operations S and a fair asynchronous activation sequence A, we compare a greedy parallel counter schedule to an *asynchronous counter schedule* $(S, (C_0^A, \ldots, C_t^A))$, where C_i^A is the resulting configuration after asynchronous round i completes according to A. For a given (increment or decrement) token c, let $I_C(c)$ be the index of the particle holding c in configuration C if such a particle exists, or ∞ if c has already been consumed. For any two configurations C and C' and any token c, we say C *dominates* C' *with respect to* c — denoted $C(c) \succeq C'(c)$ — if and only if $I_C(c) \geq I_{C'}(c)$. We say C *dominates* C' — denoted $C \succeq C'$ — if and only if $C(c) \succeq C'(c)$ for every token c.

▶ **Lemma 8.** *Given any fair asynchronous activation sequence A beginning at a valid configuration C_0^A and any nonnegative sequence of operations S, there exists a greedy parallel counter schedule $(S, (C_0, \ldots, C_t))$ with $C_0 = C_0^A$ such that $C_i^A \succeq C_i$ for all $0 \leq i \leq t$.*

So it suffices to bound the number of rounds a greedy parallel counter schedule requires to process its counter operations. The following lemma shows that the counter can always process a new increment or decrement operation at the start of a parallel round.

▶ **Lemma 9.** *Consider any counter token c in any configuration C_i of a greedy parallel counter schedule $(S,(C_0,\ldots,C_t))$. In C_{i+1}, c either has been carried over once ($I_{C_{i+1}}(c) = I_{C_i}(c)+1$) or has been consumed ($I_{C_{i+1}}(c) = \infty$).*

Unlike in the asynchronous setting, zero-testing is always available in the parallel setting.

▶ **Lemma 10.** *The zero-test operation is available at every configuration C_i of a greedy parallel counter schedule $(S,(C_0,\ldots,C_t))$.*

We can synthesize these results to bound the running time of our distributed counter.

▶ **Theorem 11.** *Given any fair asynchronous activation sequence A and any nonnegative sequence S of m operations, the distributed binary counter processes all operations in $\mathcal{O}(m)$ asynchronous rounds.*

Proof. Let $(S,(C_0,\ldots,C_t))$ be the greedy parallel counter schedule corresponding to the asynchronous counter schedule defined by A and S in Lemma 8. By Lemma 9, the leader ℓ can generate one new operation from S in every parallel round. Since we have m such operations, the corresponding parallel execution requires m parallel rounds to generate all operations in S. Also by Lemma 9, assuming in the worst case that all m operations are increments, the parallel execution requires an additional $\lceil \log_2 m \rceil$ parallel rounds to process the last operation. If ever the counter needed to perform a zero-test, we have by Lemmas 10 and 6 that this can be done immediately and reliably. So all together, processing all operations in S requires $\mathcal{O}(m + \log_2 m) = \mathcal{O}(m)$ parallel rounds in the worst case, which by Lemma 8 is also an upper bound on the worst case number of asynchronous rounds. ◀

3.2 Estimating the Convex Hull

We can now combine the movement rules of the single-particle algorithm with our distributed, multi-particle binary counter to enable the leader to measure the convex hull $H(O)$.

The leader ℓ must first orient the particle system as a spanning tree rooted at itself. This is achieved using the *spanning tree primitive* (see, e.g., [7]). Recall that all non-leader particles are assumed to be initially idle and contracted. If an idle particle P is activated and has a non-idle neighbor, then P becomes a *follower* and sets *P.parent* to this neighbor. This primitive continues until all idle particles become followers in the spanning tree; however, ℓ can immediately begin estimating the convex hull without waiting for the entire tree to form.

Two adjustments must be made before we describe the algorithm in full. First, we originally described how a path of particles could act as a distributed binary counter supporting increments and decrements by one as well as zero-tests. In the full algorithm, we require each particle to participate in up to six binary counters: one for each distance d_h, where h is one of the six half-planes. This is easily realized: for each half-plane $h \in \mathcal{H} = \{N, NE, SE, S, SW, NW\}$, particle P keeps a bit $P.bit_h \in \{\emptyset, 0, 1\}$, a final token f_h denotes the end of the counter keeping d_h, and increment and decrement tokens are tagged c_h^+ and c_h^-, respectively. Second, we will allow each particle to emulate up to two bits of each counter instead of one. The mechanics of the counter operations remain exactly as in Section 3.1, but this emulation will ensure that the counters remain connected and continuous as the particles move. These adjustments increase the memory load per particle by only a constant factor, so the constant-size memory constraint remains satisfied.

Imitating the single-particle algorithm of Section 2, ℓ performs a clockwise traversal of the boundary of the object O using the right-hand rule, updating its distance counters along the way. It terminates once it has moved in all six directions without pushing a half-plane, which it detects using its terminating bits b_h. In the multi-particle setting, we need to carefully consider both how ℓ interacts with its followers as it moves and how it updates its counters.

Rules for Distributed Counters

In general, increment and decrement tokens are handled as in Section 3.1 regardless of the type of particle holding them, with the exception of two additional considerations. The first concerns carry over operations for increment tokens. In a simple path of particles, this operation is easy: an increment token (say c_h^+) is carried over as long as necessary, in the worst case reaching the end of the counter. If this occurs, the counter's length is simply extended by one by forwarding the final token f_h to the next particle. However, the full algorithm maintains a tree of follower particles instead of a simple path. To handle this, we enforce that the counters are only extended along followers on the object's boundary.

Second, there may be times in the algorithm's execution in which a particle is only emulating one bit of each counter instead of two (due to leader role-swaps, described below). In order to keep the counters connected and as close to the leader as possible, we use the following rules. Let particles P and $Q = P.parent$ be such that Q is only emulating one bit of a counter d_h and is not holding f_h. If P is (i) emulating two bits of d_h, (ii) emulating the most significant bit of d_h and is holding f_h, or (iii) only holding f_h, P can forward the less significant counter entity it's holding to Q if P is activated, or Q can take the less significant counter entity from P if Q is activated.

Rules for Leader Computation and Movement

First suppose ℓ is contracted. If all its terminating bits b_h are equal to 1, then ℓ has estimated the convex hull, completing this phase. If the zero-test operation is unavailable, ℓ must do nothing and wait; otherwise, let $i \in [6]$ be its next move direction according to the right-hand rule. It first calculates whether the resulting move would push one or more half-planes using the update vector δ_i: let $\mathcal{H}' = \{h \in \mathcal{H} : \delta_{i,h} = -1 \text{ and } d_h = 0\}$ be the set of half-planes being pushed, and recall that since zero-testing is available, ℓ can locally check if $d_h = 0$. It then looks at the position v in direction i; ℓ can move to v if it is not holding any tokens for the first bit of its counters (the least significant bits) and either (i) v is unoccupied, or (ii) v is occupied by some contracted particle P and for each counter d_h, ℓ is either emulating two bits of d_h or is holding f_h. In case (i), ℓ simply expands to v. In case (ii), however, P is blocking ℓ from moving forward, so ℓ initiates a role-swap with P (described below). If a movement occurs as a result of either case, ℓ generates the appropriate increment and decrement tokens according to δ_i. It also updates its terminating bits according to the following rule: if $\mathcal{H}' \neq \emptyset$ (implying that ℓ is about to push one or more half-planes), then $b_h \leftarrow 0$ for all $h \in \mathcal{H}$; otherwise, for each h such that $d_h + \delta_{i,h} = 0$, set $b_h \leftarrow 1$.

In a role-swap between a leader ℓ and a non-leader P, ℓ gives P the least significant bits of its counters, the newly generated increment and decrement tokens of δ_i, and its terminating bits; promotes P to become the new leader (setting $P.parent \leftarrow \emptyset$); and demotes itself to become a follower (setting $\ell.parent \leftarrow P$). Note that this role-swap still keeps the counters connected, a fact we'll prove in Lemma 14.

Finally, if ℓ is expanded, let P be its follower child emulating bits of the counters. Then if P is contracted, ℓ pulls P in a handover.

Rules for Follower Movement

Consider any follower P. If P is expanded and has no children in the spanning tree nor any idle neighbor, then it simply contracts. If P is contracted and is following the tail of its expanded parent $Q = P.parent$, it is possible for P to push Q in a handover. Similarly, if Q is expanded and has a contracted child P, it is possible for Q to pull P in a handover. However, we need to be careful about what handovers we allow; if P is not emulating counter bits but Q is, then it is possible that a handover between P and Q could disconnect the counters (see Fig. 4). So we only allow these handovers if either (i) both are emulating counter bits, like P_3 and P_4 in Fig. 4; (ii) neither are emulating counter bits, like R_1 and R_2 in Fig. 4; or (iii) one is not emulating counter bits but the other is and holds the final token for every counter it's participating in, like P_5 and P_6 in Fig. 4.

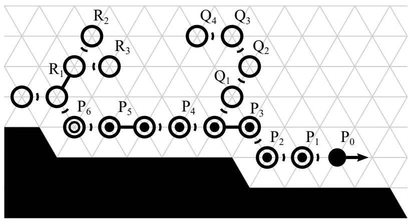


Figure 4 The leader P_0 (black dot) and its followers (black circles). Followers with dots are emulating counter bits, and P_6 holds the final token. Allowing Q_1 to handover with P_3 would disconnect the counter, while all other potential handovers depicted are safe.

3.3 Forming the Convex Hull

Once the counters contain an accurate estimation of the convex hull, the leader ℓ can simply lead the rest of the particle system in tracing it out by traversing the convex hull in clockwise order. While moving along the convex hull, ℓ uses its distributed counters exactly as in Section 3.2 to detect when it reaches a vertex of the convex hull, at which point it turns 60° to follow the next half-plane, and so on. Below, we give the movement rules for the leader and the followers, which are very similar to the movement rules of the previous phase, and explain a mechanism to detect termination.

Rules for Leader Computation and Movement

In this phase, the leader ℓ follows a rule set similar to but simpler than that of the estimation phase (Section 3.2). For brevity, we only describe the important differences. When ℓ gets activated for the first time after finishing the previous phase, it sets a *hull flag*, indicating that it occupies a node of the convex hull. We denote a particle whose hull flag is set as a *hull particle* (and all others as *non-hull*). Then, whenever ℓ is contracted, it uses its counters to determine $i \in [6]$, the next direction along the convex hull to move in. The rules for whether or not it can move in direction i are exactly as before; a successful movement can either be an expansion or a role-swap. If either type of movement occurs, ℓ generates the appropriate increment and decrement tokens according to δ_i.

To detect termination, the leader internally stores a counter that counts the number of turns taken during its traversal. The counter is reset to 0 whenever ℓ observes something other than a contracted hull particle in its movement direction. Any convex hull on the triangular lattice G_Δ has six unique turns. Thus, if the counter reaches 7 — implying that

ℓ has traversed the whole boundary by only role-swapping with contracted particles — ℓ continues along the convex hull until it is also adjacent to the object, and becomes *finished*.

Rules for Follower Movement

Followers perform handovers whenever possible to follow the leader, obeying the rules given in Section 3.2 to ensure the distributed counters are never broken. Furthermore, there are three important additions. The first ensures that a particle only sets its hull flag when it moves onto the convex hull. Consider a non-hull particle P and a neighboring hull particle Q. If P pushes Q, P sets its *pre-hull flag* indicating that it will become a hull particle after contraction. Similarly, if Q pulls P, Q sets P's pre-hull flag. Then, when P contracts, either by being pushed or by contracting itself, it becomes a hull particle.

The second addition ensures that every node of $H(O)$ is eventually occupied by a contracted particle. Whenever a hull particle P expands, it sets a flag if there is a contracted non-hull particle Q following its tail. This flag blocks the next hull particle from pushing P in a handover since it may not be able to see Q and would block Q from entering the convex hull. Expanded hull particles must pull contracted non-hull children whenever possible; when P and Q perform a handover, P resets its flag.

Finally, when a contracted follower observes its parent is finished, it also becomes finished.

3.4 Correctness

Correctness of Movement

We want to show that the leader ℓ can estimate the convex hull by moving and performing zero-tests (emulating the single particle of Section 2) and can form the convex hull. We already proved in Lemmas 5 and 6 that ℓ will always eventually be able to perform a reliable zero-test. We need the following result before showing ℓ can eventually role-swap if necessary.

▶ **Lemma 12.** *If the leader ℓ is only emulating one bit of a counter d_h and is not holding the final token f_h at time t, then there exists a time $t' > t$ when ℓ is either emulating two bits of d_h or holding f_h.*

Proof. Suppose ℓ is only emulating one bit of a counter d_h and is not holding f_h at time t. Argue by induction on i, the number of consecutive particles starting at $\ell = P_0$ that are only emulating one bit of d_h and are not holding f_h. If $i = 1$, then P_1 must either be (i) emulating two bits of d_h or (ii) emulating the most significant bit (MSB) of d_h and holding f_h, or (iii) only holding f_h. In cases (i) and (ii), P_1 can forward a bit to ℓ at its next activation (say, at time $t' > t$) while in case (iii) P_1 can forward f_h instead.

Now suppose $i > 1$ and the induction hypothesis holds up to $i - 1$. Then P_{i-1} is only emulating one bit of d_h and is not holding f_h while P_i satisfies one of the three cases above. As in the base case, after the next activation of P_i (say, at $t_1 > t$), P_{i-1} is either emulating two bits of d_h or is holding f_h. Therefore, by the induction hypothesis, there exists a time $t' > t_1$ when ℓ is also either emulating two bits of d_h or holding f_h. ◀

We can now prove the correctness of the leader's movements. This relies in part on previous work on the spanning forest primitive [7], where movement for a spanning tree following a leader particle was shown to be correct. In fact, the correctness of our algorithm's follower movements follows directly from this previous analysis.

▶ **Lemma 13.** *If the leader ℓ is contracted, it can always eventually move in direction $i \in [6]$. If ℓ is expanded, it can always eventually perform a handover with a follower.*

Proof. First suppose ℓ is contracted, and let v be the node in direction i. If v is unoccupied, ℓ can simply expand in direction i immediately. Otherwise, ℓ needs to perform a role-swap with the particle occupying v. This is only allowed when, for each counter, ℓ is emulating two bits or holding the final token. Lemma 12 shows this is always eventually true, implying ℓ can eventually perform the role-swap. Now suppose ℓ is expanded. Then, by previous work on the spanning forest primitive [7], ℓ can always eventually contract. ◀

Correctness of the Counters

To show that the moving, two-bits per particle distributed counters used in our Convex Hull Algorithm are correct, we build on the correctness proofs of Section 3.1. It is easy to see that since the six d_h counters never interact with one another, each is still individually correct. Similarly, allowing each particle to emulate up to two bits of each counter instead of just one does not affect the counters' correctness; if anything, allowing particles to see more bits in their local neighborhoods only improves the counters' abilities.

The distributed counters used in our algorithm work on a spanning tree of particles rooted at the leader ℓ. As ℓ moves, it maintains a simple path of followers who keep the counter bits, analogous to Section 3.1. We first show that this simple path of particles emulating counter bits never disconnects or intersects itself, destroying the integrity of the binary counters.

▶ **Lemma 14.** *The distributed binary counters never disconnect or intersect.*

Proof. We first show that the counters remain connected throughout the algorithm. By the spanning forest primitive [7], the particle system cannot physically become disconnected. So the only way to disconnect a counter d_h is to insert a follower that is not emulating bits of d_h between two particles that are. There are two ways this could occur. A contracted follower not emulating bits of d_h could perform a handover with an expanded follower that is (as in Fig. 4), separating the counter from its more significant bits. Alternatively, the leader ℓ could role-swap without leaving behind a bit to keep d_h connected. Both of these movements were explicitly forbidden in Section 3.2, so the counters remain connected.

Next, we argue that the counters can never intersect themselves, forming a cycle and corrupting the order of the bits. Recall that $B = |B(O)|$. In the estimation phase, ℓ traverses the boundary $B(O)$ clockwise. Since O was assumed to not have tunnels of width 1, ℓ only intersects its own traversal once it has completely circumnavigated $B(O)$. At this point, there are at least $\min\{|\mathcal{P}|, \lceil (B+1)/2 \rceil\}$ (possibly expanded) particles following ℓ along $B(O)$, and only these particles can be storing counter bits because the counters are explicitly extended along the boundary (Section 3.2). But the maximum distance ℓ can be from a half-plane is $H = |H(O)|$, the length of the convex hull, and thus the maximum value any counter d_h could attain is H. Since exactly $\lfloor \log_2 H \rfloor + 1$ bits are required to represent H in binary and $H \geq 6$ on G_Δ, the largest number of bits any d_h could require is:

$$\lfloor \log_2 H \rfloor + 1 < \left\lceil \frac{H+1}{2} \right\rceil = \min\left\{ H, \left\lceil \frac{H+1}{2} \right\rceil \right\},$$

which is at most $\min\{|\mathcal{P}|, \lceil (B+1)/2 \rceil\}$ because $B \geq H$ and we assumed $|\mathcal{P}| \geq H$. So even if every particle was only emulating one bit of each counter, the counters do not intersect. This holds even more obviously for the formation phase: we need not worry about the boundary anymore, and clearly $\lfloor \log_2 H \rfloor + 1 < \lceil (H+1)/2 \rceil$. ◀

We now show that this path of followers is always long enough to store the counters, even as the path is initially forming in the estimation phase.

▶ **Lemma 15.** *There are always sufficiently many particles to maintain the distributed binary counters.*

Proof. In the estimation phase, suppose a counter attains a value $d_h \in \{0, 1, \dots, H\}$. For this to be possible, ℓ must have moved into at least d_h new positions in its clockwise traversal of the boundary $B(O)$. Since we assume $|\mathcal{P}| \geq H$, there exists a simple path of at least $\lceil (d_h + 1)/2 \rceil$ (possibly expanded) followers following ℓ along $B(O)$. The value d_h can be expressed in $\max\{1, \lfloor \log_2(d_h) \rfloor + 1\}$ binary bits. Therefore, even in the worst case where each particle is only emulating one bit instead of two, we have $\lceil (d_h + 1)/2 \rceil \geq \max\{1, \lfloor \log_2(d_h) \rfloor + 1\}$ bits available in our counter, which is sufficient.

For the estimation phase to terminate, ℓ must traverse the entire boundary $B(O)$ at least once. So at the start of the formation phase, ℓ has at least $\min\{|\mathcal{P}|, \lceil (B + 1)/2 \rceil\}$ (possibly expanded) followers organized in a simple path behind it along $B(O)$. It follows from the proof of Lemma 14 that this number of particles is sufficient to store even the maximum value of d_h, which requires $\lfloor \log_2 H \rfloor + 1$ bits. ◀

Correctness of Termination

By Lemma 13, we have that the leader ℓ can exactly emulate the movements of the single particle in Section 2. So as a direct result of Theorem 4, ℓ completes the estimation phase in the multi-particle setting at a node both on the boundary and the convex hull of O with a correct estimate of the convex hull. So it remains to show that ℓ terminates the formation phase only after the convex hull has been formed.

▶ **Lemma 16.** *The leader ℓ finishes if and only if the convex hull is occupied by $H = |H(O)|$ contracted hull particles.*

Proof. We first show if ℓ finishes, then the convex hull is fully occupied by contracted hull particles. Since ℓ has finished, its turn counter reached 7 without resetting. Therefore, ℓ has completed a clockwise traversal of the convex hull using only role-swaps with contracted hull particles, and thus the hull must be fully occupied by contracted hull particles.

If the convex hull is already fully occupied by contracted hull particles, then the leader simply role-swaps through them, eventually making 7 turns and finishing since there are no expanded particles or gaps in the convex hull to reset the turn counter. ◀

3.5 Runtime Analysis

We now bound the worst-case number of asynchronous rounds for the leader ℓ to estimate and form the convex hull. As in Section 3.1, we use dominance arguments to show that the worst-case runtime of a carefully defined parallel schedule is no less than the worst-case number of asynchronous rounds required by our algorithm. The first dominance argument will show that the counter bits are forwarded quickly enough to avoid blocking leader movements; the second will relate the time required for the leader to trace the object's boundary and convex hull to the running time of our algorithm. Both build upon previous work [2].

Let a configuration C of a counter d_h encode the number of bits of d_h emulated by each particle, starting with ℓ and ending with the particle holding f_h. By Lemma 14 we know d_h remains connected, so $C = [C(0), \dots, C(k)]$ where each $C(i) \in \{1, 2\}$. For counter configurations C and C', $C \succeq C'$ if and only if $\sum_{i=0}^{j} C(i) \geq \sum_{i=0}^{j} C'(i)$ for all j.

▶ **Definition 17.** A *parallel bit schedule* $(C_0, \dots, C_t)$ is a sequence of counter configurations such that for every $0 \leq i < t$, C_{i+1} is reached from C_i such that one of the following holds for all $0 \leq j \leq k$:

1. Particle P_j does not forward or receive any bits, so $C_{i+1}(j) = C_i(j)$.
2. The leader $\ell = P_0$ role-swaps forward, so $C_{i+1}(0) = C_i(0) - 1 = 1$ and $C_{i+1}(-1) = 1$, shifting the indexes forward.
3. Particle P_k holding f_h either forwards f_h to P_{k-1} or P_{k-1} takes f_h from P_k, so $C_{i+1}(k) = C_i(k) - 1 = 0$ and $C_{i+1}(k-1) = C_i(k-1) + 1 = 2$.
4. Particle P_j takes one bit from P_{j+1} and gives one bit to P_{j-1}, so $C_{i+1}(j+1) = C_i(j+1) - 1 = 1$, $C_{i+1}(j) = C_i(j) = 1$, and $C_{i+1}(j-1) = C_i(j-1) + 1 = 2$.

Such a schedule is *greedy* if the above actions are taken whenever possible.

If one were to view an element of a counter configuration C emulating two bits as a contracted particle and two adjacent elements in C each emulating one bit as an expanded particle, Definition 17 exactly corresponds to the definition of a *parallel (movement) schedule* in [2]. In fact, the way we forward and take tokens (Section 3.2) can be exactly mapped onto expansions, contractions, and handovers of particles. So the next result follows immediately from Lemmas 2 and 3 of [2] and the observation that ℓ can only role-swap when $C(0) = 2$.

▶ **Lemma 18.** *Suppose at round $0 \leq i \leq t-2$ of greedy parallel bit schedule $(C_0, \ldots, C_t)$, the leader ℓ is emulating only one bit of a counter d_h and is not holding the final token f_h. Then within the next 2 parallel rounds, ℓ will be emulating a second bit of d_h or will be holding f_h.*

It remains to bound the running time for estimating and forming the convex hull.

▶ **Definition 19.** A *parallel tree-path schedule* $((C_0, \ldots, C_t), L)$ is a schedule $(C_0, \ldots, C_t)$ such that the particle system in C_0 forms a tree of contracted particles rooted at the leader ℓ, L is a (not necessarily simple) path in $G_\Delta \setminus O$ starting at the position of ℓ in C_0 and, for every $0 \leq i < t$, C_{i+1} is reached from C_i such that the following hold for every particle P:

1. Any counter operations involving P are processed according to the parallel counter schedule $(S, (C_0, \ldots, C_i))$, where S is induced by the change vectors δ_i associated with L.
2. Any bit forwarding operations implicitly involving P are processed according to the parallel bit schedule $(C_0, \ldots, C_i)$.
3. The next position in L is unoccupied and $P = \ell$ expands into it.
4. The next position in L is occupied by a particle and $P = \ell$ role-swaps with it.
5. P contracts, leaving the node occupied by its tail empty in C_{i+1}.
6. P is part of a handover with a neighboring particle Q, preferring a non-hull neighbor if P is a hull particle.
7. P occupies the same nodes in C_i and C_{i+1}.

Properties 1 and 2 of Definition 19 are analyzed in Theorem 11 and Lemma 18, respectively. Moreover, Lemma 18 shows that, in the parallel execution, the leader will never be blocked from performing a role-swap (Property 4) for longer than a constant number of rounds. The remaining properties are exactly those of a parallel (movement) schedule defined in [2]. Thus, by Lemmas 3 and 9 of [2], we have the following result:

▶ **Lemma 20.** *If L is the (not necessarily simple) path of the leader's traversal, the leader can traverse this path in $\mathcal{O}(|L|)$ asynchronous rounds in the worst case.*

By Lemma 5 of [2], $B = |B(O)|$ particles organize themselves in a spanning tree rooted at the leader ℓ in at most $\mathcal{O}(B)$ asynchronous rounds. In the estimation phase, ℓ traverses the boundary $B(O)$ at most twice before terminating, requiring $\mathcal{O}(B)$ asynchronous rounds in the worst case, by Lemma 20. The formation phase is slightly more complicated, but detailed analysis shows that ℓ may have to traverse the convex hull $H(O)$ up to $\log_2 H$

times, each time halving the number of remaining non-hull particles needed to fill $H(O)$ with contracted particles. Each traversal requires $\mathcal{O}(\log_2 H)$ asynchronous rounds in the worst case, by Lemma 20. Putting it all together, we conclude with the following theorem.

▶ **Theorem 21.** *Our Convex Hull Algorithm solves the Convex Hull Formation Problem for an object O in $\mathcal{O}(B + H \log H)$ asynchronous rounds in the worst case.*

4 Extensions

In this section we describe some extensions to the Convex Hull Algorithm. First, we show how the algorithm can be extended to rearrange the hull particles into the object's convex hull. We then describe modifications of the algorithm to maintain its correctness when there are insufficient particles to cover the convex hull, and when the object has tunnels of width 1.

4.1 Forming the Ortho-Convex Hull

Using the Convex Hull Algorithm, we can easily rearrange the hull particles to occupy all nodes of the object's ortho-convex hull. The idea of the algorithm, which we refer to as the *Ortho-Convex Hull Algorithm*, is to progressively transform the structure of hull particles forming the convex hull into the object's ortho-convex hull by successively moving particles at vertices of the structure of hull particles in the direction of the object.

We describe the algorithm in three steps: First, we describe how the hull particles are transformed into a directed cycle along the object's convex hull. Second, we introduce *local movements* to transform the structure of hull particles into the object's ortho-convex hull. Finally, we describe a method to determine termination.

Constructing the Directed Cycle

After the completion of the Convex Hull Algorithm, all nodes are finished and contracted and the leader is adjacent to the object. To initiate the Ortho-Convex Hull algorithm, the leader first performs a broadcast. Upon receiving the broadcast, every hull particle P, which must be adjacent to two other hull particles, declares them as its predecessor and successor, respectively, on the directed cycle in clockwise direction along the convex hull. More specifically, if P is a follower, it sets its successor to be its parent in the spanning tree, and if P is the leader, it sets its successor to be its adjacent hull particle whose successor is not P. Correspondingly, the nodes set their predecessors.

Local Movements

Let P be a hull particle. We say P is *convex*, if is contracted and there is only a single node between its successor and predecessor in clockwise direction around P. Correspondingly, we say it is *reflex*, if there is only a single node in counter-clockwise direction. Examples of convex and reflex particles are P_1 and P_3 in Figure 5a, respectively.

First, P waits until both its successor and predecessor have completed the first step. Afterwards, whenever it is convex and the node u between its successor and predecessor is not a node of the object, P can perform a *local movement* onto u. More specifically, if u is not occupied by a particle, P simply expands onto u. If otherwise u is occupied by a particle P', P essentially swaps with P' by declaring P' as a hull particle, setting its successor and predecessor accordingly. P itself sets its hull flag to 0 and declares P' as its parent. In

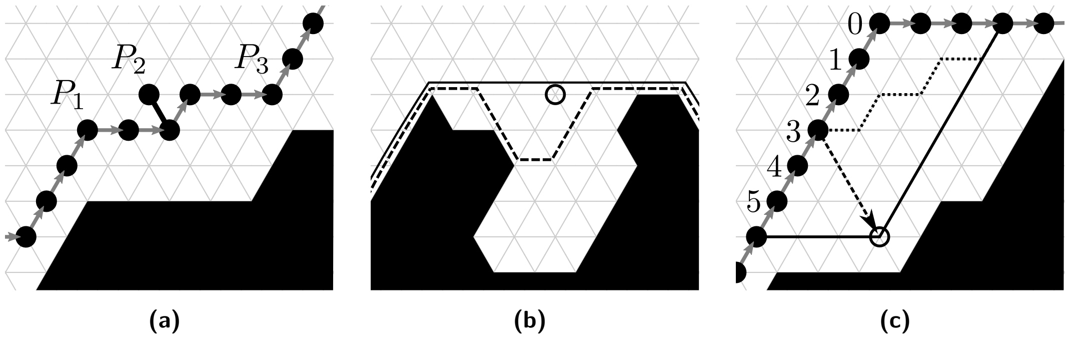


Figure 5 (a) Hull particle P_1 is on a convex vertex of the structure of hull particles and can perform a local movement in SE direction. P_2 has just performed a local movement in SE direction and already updated the predecessor and successor pointers. P_3 is on a reflex vertex. (b) The first case for node u (black circle) and the sets U^* (nodes below the dashed line) and U_i (nodes below the solid line) in the proof of Lemma 23. (c) An example of the particle movement as described in the proof of Lemma 25.

both cases P also has to update the successor pointer of its (former) predecessor, and the predecessor pointer of its (former) successor, to point to u.

Whenever a hull particle P is expanded, it simply contracts if there is no non-hull follower whose parent is P, and, otherwise, pulls in a follower if possible. As in the Convex Hull Algorithm, non-hull followers simply push and pull other non-hull particles whenever possible.

Termination Detection

After the leader has set its successor and predecessor in the first step of the algorithm, it sends a token with bit 1 to itself. Every particle P that is contracted and has received the token sets the token's bit to 0, if P can perform a local movement. It then forwards the token to its successor. Whenever the leader receives the token with bit 0, it resets the bit to 1 and forwards it. When the leader receives the token with bit 1, the ortho-convex hull has been constructed and the leader finishes the algorithm. Every contracted particle whose neighbor has finished finishes as well.

Analysis

It remains to show that the correctness and runtime of the Ortho-Convex Hull Algorithm. In the following, we say *the convex hull has been formed* if all nodes of the ortho-convex hull are occupied by contracted hull particles.

▶ **Lemma 22.** *The leader does not terminate before the ortho-convex hull has been formed.*

Proof. Assume to the contrary the leader terminates although a local movement is still possible. Let P_1 be the leader and $C = (P_1, P_2, \ldots, P_k = P_1)$ be the cycle of hull nodes, where P_{i+1} is the successor of P_i. Note that C never changes during the execution of the algorithm. We regard the execution of the algorithm as a sequence of activations, and refer to the t's global activation as *step t*. If a contracted particle P_i cannot perform a local movement after step t, but can perform a local movement after step $t' > t$, then P_{i-1} or P_{i+1} must have performed a local movement at some step between t and t'.

Since P_1 terminates, the token must have traversed the whole cycle, and whenever a particle forwarded the token it was contracted and not able to perform a local movement.

Furthermore, there must be a first step t such that afterwards the token is stored by some particle P_j, $1 < j < k$, and a particle P_i, $i < j$, can perform a local movement. By our choice of t, the token was not forwarded to P_j in step $t-1$ (otherwise we could choose t smaller), and therefore already stored by P_j at the beginning of step $t-1$ (i.e., P_j was expanded or not activated in step $t-1$). Since P_j is not able to perform a local movement the next time it is activated, it is also not able to perform a local movement after step $t-1$. P_1 can never perform a local movement since it is adjacent to the object. By our choice of t, also no other particle in between P_1 and P_j (including P_i) is able to perform a local movement at step $t-1$. Therefore, P_i can impossibly be able to perform such a movement at step t. ◀

▶ **Lemma 23.** *The hull particles eventually form the ortho-convex hull. Afterwards, no local movement is possible anymore.*

Proof. Let $U_i \subset V$ be the set of nodes enclosed by the hull particles after the i-th local movement. We show that (1) U_i is ortho-convex and contains O for all i, (2) if U_i is not a minimal ortho-convex set containing O then a local movement is possible. Together with the fact that $U_{i+1} \subset U_i$, we obtain the first claim. The second claim immediately follows from (2).

We first show (1) by induction on i. Initially, the particles form the convex hull of O, so U_0 is ortho-convex and contains O. Now let U_i be ortho-convex and contain O and consider the next local movement the algorithm performs. W.l.o.g., assume a particle P whose successor is at E and whose predecessor is at SW performs a local movement into SE direction onto node u. Clearly, U_{i+1} still contains O after the movement. Since U_i is ortho-convex, every straight line of nodes (i.e., any of the lines in W, NW, and NE direction) that includes u only contains a connected set of nodes from U_i. Since the W, NW, and NE neighbors of u are not in U_i, and $U_{i+1} = U_i \setminus \{u\}$, the same must hold for U_{i+1}.

For the proof of (2), assume to the contrary that U_i is not minimal but no local movement is possible. Then every particle that is a convex vertex of the polygon of hull particles must be adjacent to the object, and thus for every convex node $u \in U_i$ we have $u \in O$. Since U_i is not minimal, there must be a set $U^* \subset U_i$ that is ortho-convex and contains O. Therefore, and since U^* cannot have any holes, there must be a node $u \in U_i$, $u \notin U^*$ that lies on the *border* of U_i, i.e., that is adjacent to a node of $V \setminus U_i$. Then there must be a convex vertex of U_i, which must therefore also be in O, in some direction, w.l.o.g., say W. Furthermore, there must either be a convex vertex or a reflex vertex (which might be u itself) in the opposite direction E (the first case is depicted in Figure 5b). In both cases, there must exist a node of O in direction E. Since $u \notin U^*$, and U^* contains O, U^* cannot be ortho-convex, which leads to a contradiction. ◀

The following lemma immediately follows from the definition of the algorithm.

▶ **Lemma 24.** *After the ortho-convex hull has been formed, the token traverses the cycle at most twice before the leader terminates.*

We now turn to the runtime of the algorithm.

▶ **Lemma 25.** *The ortho-convex hull is formed within $O(|H(O)|)$ rounds. After an additional $O(n)$ rounds, all particles have terminated.*

Proof. We first show that the ortho-convex hull is formed (and all particles are contracted) within $O(|H(O)|)$ rounds. Consider the structure of hull particles forming the object's convex hull at the beginning of the algorithm. Note that any two subsequent convex vertices of the structure, e.g., western and the north-western vertex, are connected by a straight line of hull particles such that at least one hull particle of that sequence is adjacent to the object. For example, let u be the NW convex vertex, and let v be the first hull node in direction SW from u, and w be the first hull node in direction E, respectively, that are adjacent to the object. All local movements that are performed as a direct or indirect consequence of u's first local movement can only be at hull particles in between these three nodes. Therefore, w.l.o.g., it suffices to analyze the execution of the algorithm for these nodes and argue that the algorithm performs analogously on the other five convex vertices.

We enumerate the hull particles between u and v in SW direction from 0 to k, i.e., $u = 0$ and $v = k$ (see Figure 5c for an illustration), and define d_i to be the distance from i's initial position s_i to its final position t_i (black circle in the figure) adjacent to the object in SE direction. We will now show that i has reached t_i after at most $2(d_i + i)$ rounds. First, note that by definition of the ortho-convex hull no node of the parallelogram spanned from t_i in the directions W and NE can be a node of the object (solid lines in the Figure 5c). Therefore, it can easily be shown by induction on the number of rounds that after $2i$ rounds, particle $j \leq i$ has performed $\min(\lceil (i-j)/2 \rceil, d_i)$ local movements in SE direction (nodes on the dashed line in Figure 5c). Note that since at the beginning of the algorithm all particles are contracted, the movement of a hull particle will never by delayed by any follower outside the convex hull, which can also easily be shown by induction. In the next round i will perform its first local movement, and, by following the inductive argument, will have reached t_i after an additional $2d_i$ rounds. The same can be shown analogously for the nodes spanned from u to w. As both d_i and i are bounded above by $|H(O)|$, the algorithm takes $O(|H(O)|)$ rounds.

After the ortho-convex hull has been formed, by Lemma 24 the token traverses the cycle at most twice, which takes at most $O(|H(O)|)$ rounds. Finally, the termination broadcast from the leader takes an additional $O(n)$ rounds. ◀

The previous lemmas immediately imply the following theorem.

▶ **Theorem 26.** *The algorithm solves the Ortho-Convex Hull Problem in time $O(|H(O)|)$. After an additional $O(n)$ rounds, all particles have terminated.*

4.2 Partially Forming the Convex Hull with Few Particles

For the description of the Convex Hull Algorithm we assumed that $|\mathcal{P}| \geq |H(O)|$, which certainly is a necessary condition to solve the Convex Hull Problem. However, considering practical applications, it might happen that there are not enough particles to actually cover the hull. In this section we show how the algorithm can be modified in order to partially fill the convex hull in case that $|\mathcal{P}| < |H(O)|$. More specifically, our goal is that all particles eventually are contracted hull particles and occupy a segment of the object's convex hull, maintaining the property that $\mathcal{P}$ is connected to O.

To handle the case that there are not enough particles to form the convex hull, we have to address two issues. First, it might happen that during the traversal of the convex hull in the second phase of the Convex Hull Algorithm the particle system temporarily disconnects from the object. Second, the termination criterium will never be satisfied, and thus the particles traverse the convex hull indefinitely. We show how the algorithm can be modified to cope with both issues.

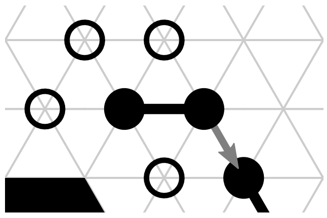


Figure 6 The center particle P is in a critical situation, if it is expanded, only its tail still being adjacent to the object, and there is no non-hull follower whose parent pointer points to P's tail (which must lie in one of the circled nodes). A contraction by P could possibly, but not necessarily, violate connectivity to the object.

Maintain Connectivity

It is easy to see that if the particle system ever disconnects from the object, then it does so because a particle in a *critical situation*, which is depicted in Figure 6, performs a contraction. To circumvented losing connectivity, we disallow a particle P in a critical situation to perform a contraction. Instead, P sends a token along the line of hull particles in the direction of ℓ. If any particle that stores the token is adjacent to a non-hull particle, a *continue* token is forwarded back to P, instructing P to continue with the algorithm and ignore subsequent critical situations. If otherwise the token is received by ℓ without ever encountering a non-hull particle, and ℓ has not already reached the position of P (in which case the convex hull has been filled in the course of the token's traversal), ℓ sends back an *abort* token.

Every hull particle that receives the abort token begins to move in the opposite direction. That is, from now on it pulls its parent and pushes its only child, whenever possible. The parent pointers need to be updated accordingly. The leader simply contracts whenever it is expanded. Once P becomes contracted, it finishes. In turn, whenever a particle is contracted and its only child is finished, it finishes itself.

We first show that if P receives a continue token, it is safe to continue with the algorithm. Note that as non-hull particles are given preference to enter the hull, no other particle than P will ever encounter a critical situation afterwards. Since P ignores all subsequent critical situations, the algorithm's exection will not be interrupted by this check again.

▶ **Lemma 27.** *If P receives a continue token, then $\mathcal{P}$ cannot disconnect from the object.*

Proof. As there is no non-hull particle whose parent pointer points to P's tail, but there is a non-hull particle adjacent to a hull particle somewhere, ℓ must have performed a swap with a non-hull particle in the second phase of the Convex Hull Algorithm already. Consider the first such situation, and let P' be the particle in front of ℓ with which it swaps. As ℓ has completely traversed the boundary of the object in clockwise order in the first phase of the algorithm, P' must be connected to ℓ over a sequence of particles around the object in clockwise order. As the convex hull is the shortest cycle around the object, there must be at least $\lfloor 1/2|H(O)|\rfloor$ particles. By definition of the algorithm, there will also be at least $\lfloor 1/2|H(O)|\rfloor$ particles on the convex hull, which means that at least one hull particle is always adjacent to the object. ◀

Next, we show that if ℓ creates an abort token, there are not enough particles to cover the convex hull anyway, and thus moving backwards is viable and correct.

▶ **Lemma 28.** *If P receives an abort token, then $|\mathcal{P}| < |H(O)|$.*

Proof. When ℓ sends back an abort token, it must have received a token from P that never encountered a non-hull particle anywhere. Therefore, at this point there cannot be any non-hull particle, and ℓ is not adjacent to P, i.e., all particles lie on the convex hull, which still is only partly filled. Therefore, $|\mathcal{P}| < |H(O)|$. ◀

Finally, it is easy to see that the above modification elongates the runtime of the Convex Hull Algorithm only by an additional $O(|H(O)|)$.

Detect Termination

If the particles do not already finish according to the above modification, then they will indefinitely traverse the convex hull as the leader never sees a contracted hull particle in front of it, and consistently resets c. Note that once all particles have become hull particles, the leader will perform a traversal without encountering any non-hull particle. We can therefore detect this situation by using a second termination counter c' which counts the number of turns taken by ℓ independently from c, and which is reset whenever ℓ is adjacent to a non-hull particle. If c' reaches value 7, the leader finishes.

▶ **Lemma 29.** *The leader eventually finishes according to counter c' if and only if $\mathcal{P} \leq |H(O)|$.*

Proof. If the leader finishes because $c' = 7$, then it has performed a complete traversal of the convex hull without encountering a non-hull particle. As particles never leave the convex hull, $\mathcal{P} \leq |H(O)|$. If $\mathcal{P} \leq |H(O)|$, then eventually all particles are hull particles and the leader will terminate according to counter c'. ◀

Note that if $\mathcal{P} = |H(O)|$, then in principal, both termination counters might become 7, which is totally fine.

4.3 Lifting the 2-Connectivity Restriction

Finally, we provide some ideas towards lifting the 2-connectivity restriction, i.e., to allow tunnels of width 1. In this case, the leader could visit the same node twice or even thrice in a single traversal of the object's boundary. This implies two difficulties we need to address. First, the leader cannot simply follow the right-hand rule anymore as there might be several possible movement directions. Instead, it has to remember from which direction it has reached a node. It can easily be seen that this information suffices for the leader to compute its next move according to a clockwise traversal.

The other difficulty lies in the possibility for the leader to run into particles that emulate bits of the counter. Note that this can only ever happen if the leader walks into a *cave*, i.e., a finite component of the subgraph of G induced by $V \setminus O$ in which we additionally remove all bridge nodes. However, under the reasonable assumption that the leader does not start in a cave, we can easily modify the traversal of the leader to never move into a cave. If we also allow the leader to start in a cave, then it would either have to first find its way out, which essentially reduces to the problem of 'escaping a labyrinth', or we would have to find a way to allow the leader to swap itself through particles without breaking the counter. Although we strongly believe that this can be done by employing additional techniques, e.g., by letting particles 'simulate' multiple (but at most 3) particles at once, we leave out a technical description of the necessary modifications.

References

1 Selim G. Akl and Kelly A. Lyons. *Parallel Computational Geometry.* Prentice-Hall, Inc., 1993.

2 Joshua J. Daymude, Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. On the runtime of universal coating for programmable matter. *Natural Computing*, 17(1):81–96, 2018.

3 Joshua J. Daymude, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Improved leader election for self-organizing programmable matter. In *Algorithms for Sensor Systems*, ALGOSENSORS '17, pages 127–140, 2017.

4 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The amoebot model. Available online at `https://sops.engineering.asu.edu/sops/amoebot`, 2017.

5 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: Amoebot - a new model for programmable matter. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 220–222, 2014.

6 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Universal shape formation for programmable matter. In *28th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 289–299, 2016.

7 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Universal coating for programmable matter. *Theoretical Computer Science*, 671:56–68, 2017.

8 Mohamadou Diallo, Afonso Ferreira, Andrew Rau-Chaplin, and Stéphane Ubéda. Scalable 2D Convex Hull and Triangulation Algorithms for Coarse Grained Multicomputers. *Journal of Parallel and Distributed Computing*, 56(1):47–70, 1999.

9 David Doty. Theory of algorithmic self-assembly. *Communications of the ACM*, 55(12):78, 2012.

10 Eugene Fink and Derick Wood. *Restricted-Orientation Convexity.* Monographs in Theoretical Computer Science. An EATCS Series. Berlin, Heidelberg, 2004.

11 Per-Olof Fjällström, Jyrki Katajainen, Christos Levcopoulos, and Ola Petersson. A sublogarithmic convex hull algorithm. *BIT*, 30(3):378–384, 1990.

12 Mujtaba R. Ghouse and Michael T. Goodrich. In-Place Techniques for Parallel Convex Hull Algorithms. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, SPAA, pages 192–203, 1991.

13 Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Fabian Kuhn, Dorian Rudolph, and Christian Scheideler. Shape recognition by a finite automaton robot. In *Abstr. European Workshop on Computational Geometry (EuroCG)*, pages 73:1–73:6, 2018.

14 Ferran Hurtado, Enrique Molina, Suneeta Ramaswami, and Vera Sacristán. Distributed reconfiguration of 2D lattice-based modular robotic systems. *Autonomous Robots*, 38(4):383–413, 2015.

15 Rolf G. Karlsson and Mark H. Overmars. Scanline algorithms on a grid. *BIT*, 28(2):227–241, 1988.

16 Nancy Lynch. *Distributed Algorithms.* Morgan Kauffman, 1996.

17 R. Miller and Q. F. Stout. Efficient parallel convex hull algorithms. *IEEE Trans. Computers*, 37(12):1605–1618, 1988.

18 Matthew J. Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, 2014.

19 Alexandra Porter and Andréa W. Richa. Collaborative computation in self-organizing particle systems. In *Proceedings of the 17th International Conference on Unconventional Computing and Natural Computation*, UCNC '18, 2018. To appear; available online at `https://arxiv.org/abs/1710.07866`.

20 Sergio Rajsbaum and Jorge Urrutia. Some problems in distributed computational geometry. *Theoretical Computer Science*, 412(41):5760–5770, 2011.

21 Damien Woods. Intrinsic universality and the computational power of self-assembly. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 373(2046), 2015.

22 Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *Innovations in Theoretical Computer Science (ITCS)*, 2013.