Efficient Parallel Self-reconfiguration Algorithm for MEMS Microrobots

Hicham Lakhlef, Hakim Mabed, and Julien Bourgeois
FEMTO-ST/DISC, University of Franche-Comté, 1 Cours Leprince Ringuet, 25201, Montbeliard, France
$\{hlakhlef, hmabed, julien.bourgeois\}\ @femto - st.fr$

***Abstract*—In this paper we propose a distributed and efficient parallel self-reconfiguration algorithm for MEMS microrobots. MEMS microrobots perform various missions and tasks in a wide range of applications including odor localization, firefighting, medical service, surveillance and security, and search and rescue. To achieve these tasks the self-reconfiguration for MEMS microrobots is required. The self-reconfiguration with shared map does not scale. Because with the map (predefined positions of the target shape) each node should store all predefined positions of the target shape, therefore this is not always possible as MEMS nodes have a low-memory capacity. In this paper, we present an efficient self-reconfiguration algorithm without predefined positions of the target shape, which reduces the memory usage to a constant complexity. This algorithm improves the energy consumption by minimizing the amount of displacement and the number of messages.**

***Index Terms*—Parallel Algorithms; Distributed Algorithms; Self-reconfiguration; Optimality; Logical Topology; Physical Topology**

I. Introduction

Micro electro mechanical system (MEMS) is a technology that enables the batch fabrication of miniature mechanical structures, devices, and systems. MEMS are miniaturized and low-power devices that can sense and act. It is expected that these small devices, referred to as MEMS nodes, will be mass-produced, making their production cost almost negligible. Their applications require a massive deployment of nodes, thousands or even millions [7], [26] which will give birth to the concept of Distributed Intelligent MEMS (DiMEMS) [3]. A DiMEMS device is composed of typically thousands or even millions of MEMS nodes. Some DiMEMS devices are composed of mobile MEMS nodes [1], some others are partially mobile [32] whereas other are not mobile at all [3]. Due to their small size and the batch-fabrication process, MEMS microrobots are potentially very cheap, particularly through their use in many areas in our lifetime [11], [30]. At the present time, swarm robotics is gaining increasing attention since large-scale swarms of robots can perform various missions and tasks in a wide range of applications including odor localization, firefighting, medical service, surveillance and security, and search and rescue [22], the self-reconfiguration for MEMS microrobots is necessary to do these tasks. One of the major challenges in developing a microrobot is to achieve a precise movement to reach the destination position while using a very limited power supply. Many different solutions have been studied for example, within the $Claytronics$ project [1], [2], [8], [12], [19] each microrobot helps its neighbor to move to the desired position, which introduce the idea of a collaborative way of moving.

In the literature, the self-reconfiguration can be seen from two different points of view. First, it can be defined as a protocol, centralized or distributed, which transforms a set of nodes to reach the optimal logical topology from a physical topology [10]. On the other hand, the self-reconfiguration is built from modules which are autonomously able to change the way they are connected, thus changing the overall shape of the network [8], [24]. This process is difficult to control, because it involves the distributed coordination of a large numbers of identical modules connected in time-varying ways. The range of exchanged information and the amount of displacement, determine the communication and energy complexity of the distributed algorithm. When the information exchange involves close neighbors, the complexity is moderate and the resulting distributed the self-reconfiguration algorithm scales gracefully with network size.

This work takes place within the Claytronics project and aims at optimizing the logical topology of the network through rearrangement of the physical topology as we will see in the next sections.

II. Related Works

Many terms refer to the concept of self-reconfiguration. In several works on wireless networks the terms used are $redeployment$ and $self - organization$, this last term is also used to express the partitioning and clustering of ad-hoc networks or wireless networks to groups called cliques or clusters. Also, the self-organization term can be found in protocols for sensors networks to form a sphere or a polygon from a center node [28]. Others algorithms for the redeployment of sensor networks in [14], [23]. In [24], a protocol of self-configuration where the desired configuration is grown from an initial seed module, after a generator uses a 3D CAD model of the target configuration and outputs a set of overlapping bricks which represent this configuration.

A growing number of research on self-reconfiguration for microrobots using centralized algorithms have been done, among them we find centralized self-assembly algorithms [21]. Other approaches give each node a unique ID and a predefined position in the final structure; see for instance [27]. The drawback of these methods is the centralized paradigm and the need for nodes identification. More distributed approaches include [4], [5], [9], [13], [25]. In [29] a deterministic distributed

algorithm for the reconfiguration of modular robots to straight chain configuration.
Claytronics, is the name of a project led by Carnegie Mellon University and Intel corporation. Many works have already been done within the Claytronics project. In [6], [8], the authors propose a metamodel for the reconfiguration of catoms starting from an initial configuration to achieve a desired configuration using $creation$ and $destruction$ primitives. The authors use these two functions to simplify the movement of each catom. Another scalable algorithm can be found in [19]. In [2], a scalable protocol for Catoms self-reconfiguration is proposed, written with the MELD language [1], [20] and using the creation and destruction primitives. In all these works, the authors assume that all Catoms know the correct positions composing the target shape at the beginning of the algorithm and each node is aware of its current position. The first self-reconfiguration without predefined positions of the target shape appears in [15], [16], [18]. However, these solutions are not parallelized. In [17] a another solution that guarantees the connectivity of the network through the execution time of the algorithm, this solution is not optimal.

III. Contributions

We introduce a state model where each node can see the state of its physical neighbors to achieve the self-reconfiguration for distributed MEMS microrobots, using the states the nodes collaborate and help each other. Contrary to existing works, in our algorithm each node has no information on the correct positions (predefined positions) of the target shape, and movement of microrobots is fully implemented.
We propose an efficient, distributed and parallelized algorithm for nodes self-reconfiguration where each node can communicate only with its physical neighbors. We study the case of a self-reconfiguration from a chain of microrobots to a square. The performance of the self-organization algorithm is evaluated according to the number of movements, the amount of memory used and the time taken. In this paper the MEMS network is organized initially as a chain. By choosing a straight chain as the initial shape, we aim to study the performance of our approach in extreme case. Indeed, the chain form represents the worst physical topology for many distributed algorithms in terms of fault tolerance, propagation procedures and convergence. First, the number of direct contacts between microrobots is minimal and secondly the average distance between two robots (in term of number of hops) is of $(n+1)/3$ where n is the number of microrobots.
To assess the distributed algorithm performance, we present the simulation results and we compare to former results.

The rest of the paper is organized as follows: Section 4 discusses the model and some definitions. Section 5 discusses the proposed algorithm, analyzes the number of sent messages and the number of movements, it discusses memory space required and shows the generalization of the algorithm. Section 6 details the simulation results. Finally, section 7 summarizes our conclusions and illustrates our suggestions for future work.

IV. Model, definitions and tools

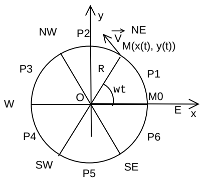

Figure 1. Node modeling, in each movement the node travels the same distance

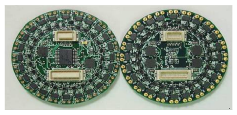

Figure 2. Two catoms.

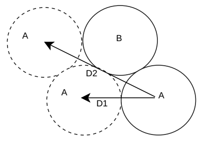

Figure 3. Traveled distance in one movement = 2R, the node A travels 2R in one movement

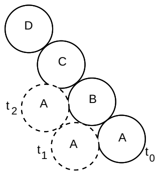

Figure 4. Message transmission, there will be message exchange if the node needs to know the state of a non-neighbor node

Within Claytronics, a Catom (figure 2) that we call in this paper a node is modeled as a sphere which can have at most six neighbors without overlapping (See figure 3). Within Claytronics, each node is able to sense the direction of its physical neighbors (east (E), west (W), north-east (NE), south-east (SE), south-west (SW) and north-west (NW)). The starting physical topology is a chain of n nodes linked together. A chain corresponds to a connected set of nodes where all nodes have two neighbors excepting the two extremities representing only one neighbor. We will take the example of nodes that have neighbors in NW and SE directions and we will show after how to generalize. A node A is in neighbor's list of node B if A touch physically B (figure 3). Communications are only possible through contact, which means that only neighbors can have a direct communication. Therefore, if the node didn't have neighbor at the previous or the current round of the algorithm it cannot join the other nodes of the network, so it is lost. Because it does not know where is the group or at least one node connected to the group. Thus, the algorithms of self-reconfiguration have to make sure that no node will be lost.

Consider the connected undirected graph *G = (V, E)* modeling the network, where $v \in V$, is a node that belongs to the network and, $e \in E$ a bidirectional edge of communication between two physical neighbors. For each node $v \in V$, we denote the set of neighbors of v as $N(v)$. Each node $v \in V$ knows the set of its neighbors in G, denoted $N(v)$.
We define the following terminology:
$Non{-}Snap{-}Connectivity$: there is a non-snap-connectivity if the graph that models the network is connected only at the end of the algorithm.
We call the *highest number of movements* the highest number of movements was performed by a node belongs to the network.

To calculate the highest number of movements we define the following:
Consider the figure 1 which represents a microrobot. We say that a microrobot has done a single movement if the distance between its former position and its new position is exactly twice the radius $D1 = 2R$. For example, if the node is in a position at a distance $D2$ (see the figure 3) from the former position it has done two movements. We have $360°$ can be divided to six equal angles each one has $60°$, since the perimeter at an angle a is $P_a = \pi\ Ra/180\ and\ P = 2\pi R$ we find $P1 = P2 = P3 = P4 = P5 = P6$, this means that the node can have without overlapping at most six neighbors and in each movement the node travels Ra (with $a = 60°$) from $m0$ to m. In this paper, we assume that the change of message (consultation) between two physical neighbors is carried without complexity (0 message), while the distance between two physical neighbors is zero and in the tools of simulation the node can see directly the state of its physical neighbor. If a node *to decide* needs to know the state of node which is not its physical neighbor message exchange is required, for example in the figure 4:

- At t_0: the node A needs to know the state of B to move to the new position, this movement is done without message exchange.
- At t_2: if A is in the new position and it needs to know the state of D to move then D sends a message to C informing its state to C that forwards the message to A. So, in this case there is a message exchange and A must wait two rounds to decide.
- But if at t_0 or at t_1 a message has been sent from D to C, so A at t_2 can have the state of D with a simple consultation of C's state.

It is important to minimize the number of movements regarding the energy and time of execution, the space of memory used, therefore the number of states per node.

Theorem 4.1: [1] Let y be an odd\even square number (y is an integer that is the square of an integer), then the next odd\even square number is $y + 4\sqrt{y} + 4$

Theorem 4.2: Let y be a square number (y is an integer that is the square of an integer), then if y is odd\even the next even\odd square number is $y + 2\sqrt{y} + 1$

V. Proposed Protocol

A. Parallel Algorithm with Unsafe Connectivity (PAUC)

In this section we present our protocol that ensures the property of non-snap-connectivity.

To form the matrix of our square with $\sqrt{N} \times \sqrt{N}$ nodes, we begin (according to *Theorem 4.4* and *Theorem 4.5*) with an incremental process with a correct square (for example 1x1). Then, we add each time a new sub-layer contains $3T+2$ nodes, with $T \times T$ is the last square. After, we add another sub-layer with $T+2$ nodes taking positions at the W direction relative to nodes of the last shape. If N is even, at the last layer we add $2T+1$ nodes, with $T \times T$ is the last square, figures 5 and 6 show an example. The choice of the middle node depends on the optimality of parallelism. To apply an optimal parallelism the middle node mi which is also the initiator of the algorithm should be found with the followings :

[1]The character "\" means respectively (resp.) in lemmas and theorems

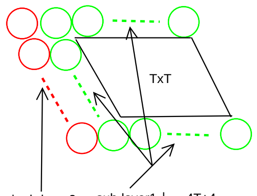

Figure 5. Represents how many nodes added to reach the next square when n is odd

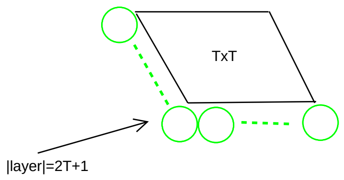

Figure 6. Represents how many nodes added in the last layer to reach the last square when n is even

Let N be the network size, $n = \left\lfloor\sqrt{N}\right\rfloor \left\lfloor\sqrt{N}\right\rfloor$ and $dif = N - n$ then:

- If n is odd and $dif < 2\sqrt{n}$ then the middle node will be $mi = (n+1)/2$, as the case in figure 8.
- If n is odd and $dif \geq 2\sqrt{n}$ then the middle node will be $mi = ((n+1)/2) + \sqrt{n} - 2$
- If n is even and $dif < 2\sqrt{n}$ then the middle node will be $mi = n/2 - ((\sqrt{n}/2) - 1)$, as the case in figure 7.
- If n is even and $dif \geq 2\sqrt{n}$ then the middle node will be $mi = (n/2 - ((\sqrt{n}/2) - 1)) + \sqrt{n} - 2$.

The middle node mi can be found by knowing the network size, an end node of the chain initializes a counter and broadcasts it, each node receives this message increments the counter until its arrives to the concerned node mi, that will have the satisfied predicate $medChain(v)$.

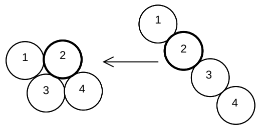

Figure 7. Represents an example of initiator finding when n is even, in this example the initiator is the node 2

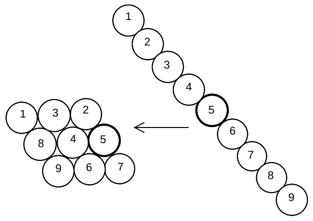

Figure 8. Represents an example of initiator finding when n odd, in this example the initiator is the node 5

Variables and Predicates

- initiator$_v$(): node v that initializes the algorithm.
- state$_v(X)$: v takes the state $X \in \{well, bad, int, nper, sint, rint, mnper, top, bottom\}$, v cannot take the states $well$ and bad in the same time.
- moveAround$state_v(u, P_x)$: move around neighbor u that has the state $state$ in such a way u becomes v's neighbor in the direction x relative to v.
- moveTo$_v$(P$_{N_x}$): move to the old position of the former neighbor at direction x relative to v.

B. Description and analysis

The algorithm PAUC (presented here after) runs in rounds,

Predicates checked only in the first round

1) initiator$_v() \equiv medChain(v)$.
2) state$_v(bad) \equiv connected_v() \wedge \neg initiator_v()$.
3) state$_v(well) \equiv initiator_v()$.
4) state$_v(nper) \equiv initiator_v()$.

Predicates checked in each round

5) thisRound$\equiv GetCurrentRound()$.
6) hasN$_{nw}v(thisRound) \equiv (N_{nw}(v) = u) \wedge state_u(bad)$.
7) N$_{nw}$lastRound$_v(LastRound) \equiv hasN_{nw}v(thisRound - 1)$.
8) hasN$_{se}v(thisRound) \equiv (N_{se}(v) = u) \wedge state_u(bad)$.
9) N$_{se}$lastRound$_v(LastRound) \equiv hasN_{se}v(thisRound - 1)$.
10) state$_v(top) \equiv (N_{se}(v) = u, initiator_u()) \vee (N_{se}(v) = u, state_u(top))$.
11) state$_v(bottom) \equiv (N_{nw}(v) = u, initiator_u()) \vee (N_{nw}(v) = u, state_u(top))$.
12) state$_v(sint) \equiv (N_e(v) = u, initiator_u())$.
13) state$_v(nper) \equiv (N_{se}(v) = u, state_u(sint))$.
14) state$_v(rint) \equiv (N_e(v) = u, state_u(well)) \wedge (N_{ne}(v) = u, \neg state_u(well), \neg state_u(int))$.
15) state$_v(mnper) \equiv (N_e(v) = u, state_u(nper)) \wedge (\neg state_v(nper)) \wedge (state_v(int) \vee state_v(well))$.
16) state$_v(nper) \equiv (N_e(v) = u, state_u(mnper)) \wedge (\neg state_v(mnper)) \wedge (N_{se}(v))$.
17) state$_v(int) \equiv ((N_e(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{se}(v) = u, state_u(well)) \wedge (N_w(v))) \vee ((N_{se}(v) = u, state_u(rint))) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_w(v)))$.
18) state$_v(int) \equiv ((N_e(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{se}(v) = u, state_u(well)) \wedge (N_w(v))) \vee (N_e(v) = u1, N_{se}(v) = u2, state_{u1}(int), state_{u2}(int)) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_w(v)))$.
19) state$_v(well) \equiv ((N_e(v) = u, state_u(int)) \wedge (N_{nw}(v))) \vee ((N_e(v) = u, state_u(int)) \wedge (N_{se}(v)))$.
20) state$_v(well) \equiv (N_w(v) = u, state_u(well)))$.
21) state$_v(well) \equiv state_v(bad) \wedge (\neg N_{se}(v)) \wedge (N_w) \wedge (N_{nw}(v) = u, state_u(well))$.
22) state$_v(well) \equiv state_v(bad) \wedge (\neg N_{se}(v)) \wedge (N_w) \wedge (N_{nw}(v) = u, state_u(well)) \wedge ((N_e(v) = u1, state_{u1}(well)))$.
23) moveTo$_v$(P$_{N_{nw}}$) $\equiv state_v(top) \wedge state_v(bad) \wedge N_{nw}lastRound_v(LastRound) \wedge \neg hasN_{nw_v}(thisRound)$.
24) moveTo$_v$(P$_{N_{se}}$) $\equiv state_v(bottom) \wedge state_v(top) \wedge state_v(bad) \wedge N_{se}lastRound_v(LastRound) \wedge \neg hasN_{se_v}(thisRound)$.
25) moveAround$int_v(u, P_{se}) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{nw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{sw}(v) = u, state_u(int), state_u(top)) \wedge (\neg state_u(nper)) \wedge (((N_{se}(v) = u1, \neg N_e(u1) = u) \wedge N_{nw}(v)) \vee (N_{se}(v) = u1, N_e(u1) = u))$.
26) moveAround$well_v(u, P_{se}) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{nw}(v)) \wedge state_v(top) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{sw}(v) = u, state_u(well), state_u(top))$.
27) moveAround$int_v(u, P_e) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{nw}(v) \wedge \neg N_e(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{se}(v) = u, state_u(int), state_u(top)) \wedge (\neg state_u(nper))$.
28) moveAround$well_v(u, P_e) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{nw}(v) \wedge \neg N_e(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{se}(v) = u, state_u(well), state_u(top))$.
29) moveAround$well_v(u, P_{ne}) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{sw}(v) \wedge \neg N_{se}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_e(v) = u, state_u(well), state_u(top)) \wedge (\neg state_u(nper))$.
30) moveAround$int_v(u, P_{ne}) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{sw}(v) \wedge \neg N_{se}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_e(v) = u, state_u(int), state_u(top)) \wedge (\neg state_u(nper))$.
31) moveAround$well_v(u, P_{ne}) \equiv (\neg N_e(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{nw}(v) = u, state_u(well), state_u(bottom))$.
32) moveAround$well_v(u, P_{se}) \equiv (\neg N_w(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_e(v) = u, state_u(well), state_u(bottom), (\neg state_u(nper))$.
33) moveAround$well_v(u, P_e) \equiv (\neg N_w(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{ne}(v) = u, state_u(well), (\neg state_u(nper))$.
34) moveAround$int_v(u, P_{ne}) \equiv (\neg N_w(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{nw}(v) = u, state_u(int), state_u(bottom)) \wedge (N_{ne}(v) \vee N_{se}(v))$.
35) moveAround$int_v(u, P_e) \equiv (\neg N_w(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{ne}(v) = u, state_u(int), state_u(bottom), (\neg state_u(nper))$.
36) moveAround$well_v(u, P_e) \equiv (\neg N_w(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge state_v(bottom) \wedge (N_{ne}(v) = u, state_u(well), state_u(mnper), (\neg state_u(nper))$.

PAUC

in each round satisfied predicates are chosen to run. The distributed algorithm seeks the desired form by using an incrementally process. In a completed increment, the nodes that build it belong already to the form; these nodes will help neighbor nodes and future neighbor nodes to get correct positions. In others words, nodes in a completed increment act as a landmark.

The middle node (mi) of the chain declares itself as an initiator with the predicate (1). Initially, all nodes are initialized with the state bad except the initiator (2), the initiator takes the states $well$ and $nper$ with (3) and (4). Nodes that are above the initiator take the state top with predicate (10), the other nodes that are under the initiator take the state $bottom$ (11). Nodes having the state $well$ or int are nodes already in the target shape and cannot move, they became steady.

To make an optimal parallelism and correct square, the number of nodes having the state top (10) to be in the same line as the initiator(in the E direction relative to the initiator) must be equal to the number of nodes having state $bottom$ (11) to be in the same line as the initiator if N is odd. If N is even, another node is added to the nodes having state top. The state $nper$ is used to achieve this purpose. That is, the node having the state $nper$ does not permit to some nodes to move around it. The initiator takes the state $nper$ (4), by taking this state the initiator and each node has this state does not allow to neighbor nodes having the state $bottom$ to move around it in order to join the line of the initiator (to became in the E direction relative to the initiator). This is done with guard $(\neg state_v(nper))$. The state $mnper$ is an intermediate state used to propagate the state $nper$ to the other concerned nodes, that will keep the parallelism optimal. The state $sint$ (12) is another intermediate state used as a reference to the first node that can get the state $nper$, the state $sint$ is an indispensable because the second node that should take the state $nper$ (13) is not a neighbor node of the initiator which is the first node that takes the state $nper$. The node that has a neighbor in the E direction having the state $nper$ takes the state $mnper$ (15). The node that has the initiator as neighbor node in the E direction takes the state $sint$ (12). The other (next) nodes that will take the state $nper$ are nodes having in the E direction a neighbor that has the state $mnper$ (16). Therefore, the node having the state $nper$ does not allow neighbor nodes having the state $bottom$ to join the line of the initiator, as these nodes are checking the predicates (30), (32), (35) and (36).

The state int is an intermediate state used to add a non-complete layer to the square shape. Thus, the nodes that have neighbors having the state $well$ take the state int with predicate (17). The first node that changes its state to int is the one in the line of the initiator. After, the state int is propagated to nodes that have neighbors having the $well$ state. Notice that, nodes take the state int if they have at least one neighbor having the state $well$, but when making a new layer there is one node that will not have a neighbor having the state $well$ and it should take the state int, the state $rint$ (14) is used to deal with this case. Notice that, nodes with the state $well$ and nodes with state int together do not composites a square, it well be a square if all nodes having the state int have in the W direction a neighbor node, this neighbor node has the state $well$. Therefore, the wave of state changing to $well$ begins with predicates (19), (20), (21) and (22).

With the predicate (23) the nodes having the states top and bad descend towards the center of the chain. As well as, nodes having the state $bottom$ and bad mount towards the center of the chain with the predicates (24). Therefore, with predicate (23), the node that has the state top and bad takes the position of its former neighbor in the NW direction. With predicate (23) the node having the state $bottom$ and bad takes the position of its former neighbor in the SE direction (24). With predicates (6), (7), (8) and (9), the node cheeks if it had a neighbor in the SE or NW direction in the previous round.

With predicate (25)/(26), the node v that has the states top and bad moves around a node u having the states top and $int/well$, node u becomes a neighbor in SE direction relative to v. With predicate (27)/(28) the node v that has the states top and bad moves around a node u having the states top and $int/well$, node u becomes a neighbor in E direction relative to v. With predicate (29)/(30), the node v that has the states top and bad moves around a node u having the states $bottom$ and $well/int$, node u becomes a neighbor in NE direction relative to v. With predicate (31)/(32), the node v that has the states $bottom$ and bad moves around a node u having the states $bottom$ and $well/int$, node u becomes a neighbor in NE/SE direction relative to v. With predicate (33)/(34) the node v that has the states $bottom$ and bad moves around a node u having the states $bottom$ and $well/int$, node u becomes a neighbor in E/NE direction relative to v. And with (35)/(36), the node v that has the states $bottom$ and bad moves around a node u having the states $bottom$ and $int/well$, node u becomes a neighbor in E direction relative to v.

Theorem 5.1: If N is the network size, $n = \left\lfloor \sqrt{N} \right\rfloor \left\lfloor \sqrt{N} \right\rfloor$ and $\eta = \left\lceil \sqrt{N} \right\rceil \left\lceil \sqrt{N} \right\rceil$, then:

- if n is odd and $n = N$ then the highest number of movements will be $((n + \sqrt{n} - 2)/2)$.
- if n is odd and $n < N$ then the highest number of movements will be $(\eta/2 - 1)$.
- if n is even and $n = N$ then the highest number of movements will be $(n/2 - 1)$.
- if n is even and $n < N$ then the highest number of movements will be $((\eta + \sqrt{\eta} - 2)/2)$.

Example: Figure 9 shows an example with explanation.

- At $t0$: with predicate (2) each node takes the state b (bad), with (10) nodes (node 1) which are above the initiator (node 2) take the state t (top), with (11) nodes (nodes 3 and 4) located under the initiator take the state B ($bottom$), with (3) the initiator takes the state w ($well$), and with (4) it takes the state n ($nper$).
- To arrive at the next step $t1$: node 1 moves around node 2 using the predicate (28), and node 3 moves around node 2 using (31). The node 1 takes the state s with (12). Node 1 cannot do another motion around node 2 since node 2 has the state n.

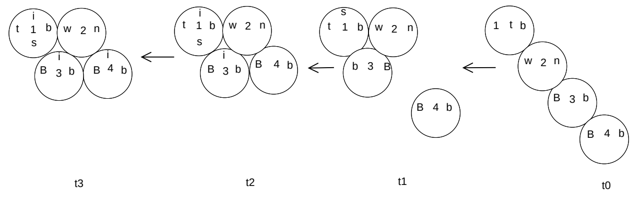

Figure 9. Represents an example of execution of PAUC with four nodes

- To arrive at the next step $t2$: node 4 moves take the position of its former neighbor (node 3) using the predicate (23). Node 1 and node 3 take the state i with predicates (17) ans (18).
- At $t3$: the target shape is obtained. Node 4 takes the state i with (17).

C. The seven states minimum

In this section we prove that seven states are necessary and sufficient to obtain the algorithm convergence. Obviously, with a single state, nodes have no way to distinguish whether they are in a good position or not and therefore if the node should move or not. Let us suppose a variant of PAUC with two states bad and $well$, with these two states, we can say that the node that has $well$ state is a steady node and is belonging to the target shape, and the node with bad state moves around nodes having $well$ state, thereby with these two states, nodes collaborate between them to make a next layer and change the state from bad to $well$. Suppose a set S of nodes having the $well$ state are correctly in the target shape. Depends on some conditions C the set B of nodes with bad state will change their state to $well$ in order to make a new layer, however as we have only two states the other nodes that are B's neighbors have likewise these C conditions and they will change their states to $well$ and became steady, although they are not even at the layer being built. So, PAUC is executed and the target shape is lost.
Two additional states are required, the state top and $bottom$, these two states are indispensable to avoid deadlock in PAUC. Indeed, in PAUC there is the predicate (23) executed by nodes to descend to the middle of the chain, and the predicate (24) executed to rise to the middle of the chain, if we remove the states top and $bottom$ from the predicates of movement the nodes will remain in their position by running (23)/(24) after (24)/(23), or (29)/(33) and (33)/(29) cyclically. Therefore, PAUC will not get finished. A variant of PAUC with four states bad and $well$ and top and $bottom$ is impossible, the reasons are the same used to prove the impossibility with the two states $well$ and bad.

The solution is to add an intermediate state int to separate neighboring nodes having the state bad and nodes having $well$ state. By adding this state, the node that has int state can change the C conditions that will be C'. Such a way, B's neighbors cannot change their state to $well$ with C', because they are not forming a new correct layer.

Let us suppose a variant of PAUC with six states bad, $well$, top, $bottom$, and int, $\neg int$. With six states the deadlock is avoided, and the conditions to change the state to $well$ are managed. However, the nodes having the state int are making a new layer adjacent to the current correct square $\sqrt{Z} * \sqrt{Z}$, the number of node having int added is $3\sqrt{Z}+2$. Therefore, as $\sqrt{Z} * \sqrt{Z}$ + $3\sqrt{Z}+2$ is not a square root (from Theorem 4.4 and Theorem 4.5), the shape is not a square. To become a square we have to add $\gamma = \sqrt{Z}+2$ nodes, these nodes will be at the direction W relative to nodes having the state int. These γ nodes can get the state $well$ because the shape is a square or an intermediate square.

With six states bad, $well$, top, $bottom$, int, and $\neg int$, the parallelism will not be optimal and the energy consumption will not be well balanced between nodes. To make an optimal parallelism and to make well balanced consumption of energy, PAUC makes two rectangles in parallel where the union gives a square. Also, to propagate the state $nperm$ to the concerned nodes of the middle, we have to use another state $mnper$, the states $\neg nper$ and $\neg mnper$ are used to check if the neighbor node has the state $nper$ and $mnper$ respectively. The state $sint$ is another intermediate state used as a reference to the second node that can get the state $nper$, the state $sint$ is an indispensable state because the second node that should take the state $nper$ is not a neighbor node of the initiator which is the first node that takes the state $nper$. The first node that changes its state to int is the one that has a neighbor having the state $nper$. After, the state int is propagated to nodes that have neighbors having the $well$ state. Notice that, nodes take the state int if have at least one neighbor having the state $well$, but when making a new layer there is one node that will not have a neighbor having the state $well$ and it should take the state int, therefore the state $rint$ is used to help these nodes to take the state $well$. ■

D. Complexity of sent messages

PAUC needs only the messages of middle node finding ($O(N/2)$). The most interesting action for message exchange in the algorithm is the one activated by state changing predicates, from bad to int and from int and bad states to $well$, it is obvious that if a node changes its state before it is sure of the $well$ state of other nodes that have moved before it in the current layer, the procedure will completely go in the opposite direction of the desired objective and the self-reconfiguration desired, the predicates (15), (16), (17), (18), (19), (20), (21), and (22) ensure without exchanging of message that the node changes its state only if all nodes that have moved before have changed their states to int or $well$, therefore the first node that begins the construction of the new layer does not need to wait for the message of the first node that began the previous layer. Since the node that is currently checking the predicates (15), (16), (17), (18), (19), (20), (21), and (22) can have this information by consulting (message) the state of its neighbors. In other words, the message was being sent before the node needs to know the state of its sender, when the node needs to know it, it will find the message at its physical neighbor. So we do not need to transmit information from the node blocked necessarily in a good position with the *well* or *int* state to other nodes which are forming the new layer which explains that throughout the algorithm in any case we do not need to transmit information between two non-neighboring nodes of the new layer. This efficiency is explained by the fact that synchronization in state changing is not required for nodes that are in the same layer. As consequence, PAUC needs only the messages of middle node finding.

E. Generalization of the algorithm

Presented algorithm PAUC is specific to a chain case where nodes form initially a straight line oriented toward SE-NW directions. In this section we describe how the algorithm can be generalized to any kind of initial chain with any directionWe start by explaining how the two end nodes are selected whatever are the directions of the straight chain. The node that has only one neighbor situated either in SW, SE or E direction is the first end node. The second end node has only one neighbor situated either in NW, NE or W. For the other nodes, every node in the chain can deduce the orientation of the chain by analyzing the orientation of its two neighbors, they use the orientation of their two neighbors to determine the orientation of the formed chain. Generally, every node after the detection of the chain orientation, noted D-$\overline{D}$, runs a variant of the PAUC algorithm depending of the orientation $D \in \{W, NW, NE\}$. The variant of PAUC algorithm, $PAUC^D$, represents an adaptation of the the original PAUC algorithm (corresponding to $PAUC^{NW}$) to the two other possible orientations with changing the directions in predicates. For instance, if the initial chain is oriented NE-SW, the algorithm $PAUC^{NE}$ is called, and the square form is realized using moves of type $moveAroundbad_v(u, P_w)$, $moveAroundwell_v(u, P_w)$ and $moveAroundwell_v(u, P_{nw})$. The usage of these three predicates is described in figure 10 that presents an example with nodes having the state *bottom* moving around nodes having the state *well* or *int*.

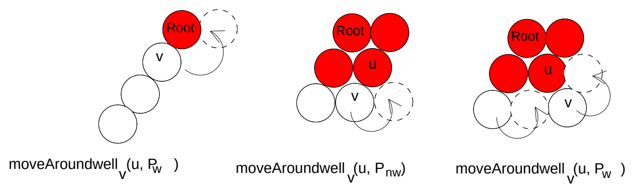
Root
v
moveAroundwell$_v$(u, P$_w$)
Root
u
v
moveAroundwell$_v$(u, P$_{nw}$)
Root
u
v
moveAroundwell$_v$(u, P$_w$)

Figure 10. Moves adaptation in the case of NE-SW chain. Dark nodes are nodes having the state *well* or *int*

VI. Simulation

We have done the simulation with the Dynamic Physical Rendering Simulator (DPRSsim) [31]. In our simulations the radius of the node is 1 mm. We simulated with a laptop with processor Intel(R) Core(Tm) i5, 2.53 GHz with 4 Go of memory. We note in the figures of simulation, $PAUC1$ for the values odd of n with $t(\eta) = \eta/2 - 1$ and $p(n) = ((n + \sqrt{n} - 2)/2)$. And $PAUC2$ for the values even of n, with $v(n) = (n/2 - 1)$ and $s(\eta) = ((\eta + \sqrt{\eta} - 2)/2)$

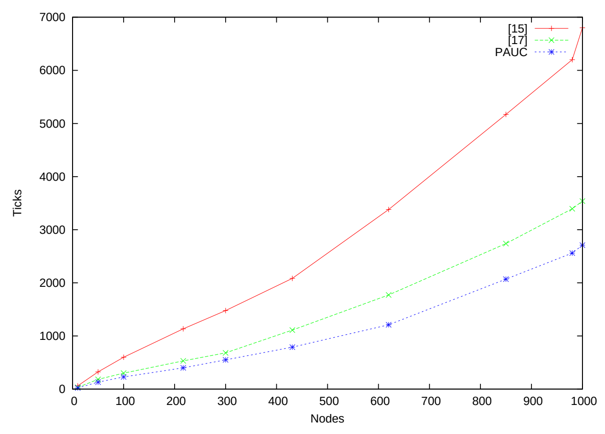

Figure 11. Execution time

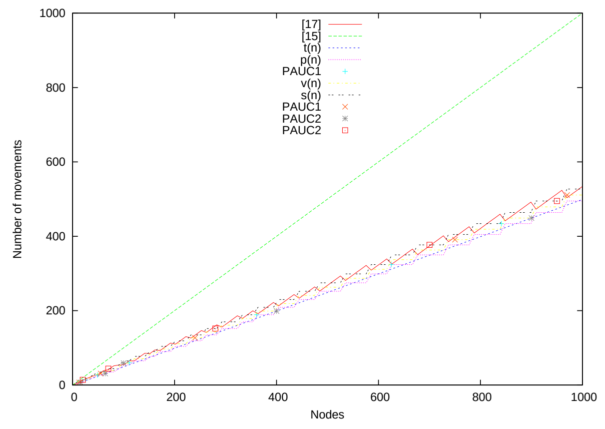

Figure 12. Highest number of movements

The simulation results come to agree our theoretical results. The figure 11 represents the execution time in ticks by the number of nodes; this figure compares the execution time of the algorithm proposed in this paper to those given in [15] and [17]. Figure 12 presents the highest number of movements found in this paper compared to the one in [15]. The effects of parallelism appear well in the curve representing the execution time of PAUC, [15] and [17], as in PAUC movements of microrobots are in parallel which optimizes the time of the algorithm and PAUC makes two rectangles in the same time that their union gives the target shape. An interesting thing to notice here is that optimizing the execution time of the algorithm will have a direct effect on messages therefore a gain on communication, and if the algorithm is fast then the critical information arrives early at the concerned node. Also, if the task is a heavy parallel computation, therefore if the algorithm is faster, the parallel computing will be fast and light on the nodes because the tasks are well distributed. In figure 11 we see that whenever the network size increases the difference increases dramatically. We remark in figure 12 that the number of movements in PAUC is much lower, which will increases the probability of lifetime of nodes, therefore the probability that the node continues its task (its movements), this is also improving the energy consumption. However, PAUC needs seven states per node and the algorithm in [15] needs three states per node, and the algorithm in [17] needs ten states.

VII. Conclusion

In this paper, we presented a new method to complete the self-reconfiguration where the nodes do not know the fixed positions of the target shape but only the aimed shape; nodes collaborate and help each other by analyzing the characteristics of the target shape. Compared to the literature works this algorithm is scalable because each node needs only seven state to achieve the self-reconfiguration. Nodes in our paper can perform the algorithm regardless the place where they are because the algorithm is independent of the map, that what we call portability. The proposed algorithm is characterized by a constant memory needs and message exchange is limited to neighboring consultations. Consequently, system reconfiguration is fast. we have shown the self-reconfiguration parallelized possibility without predefined positions of the target shape. The proposed algorithm optimizes the execution time and the number of movements, each node needs seven states to help and collaborate with neighbors, its execution time and highest number of movements are much better than those in literature works. However, some open problems remain; we will study the fault tolerance on self-reconfiguration in microrobots networks. The study of the effect of self-reconfiguration on the permutation routing where the objective will be to optimize the path of a node to go to the correct position where it finds its correct data.

VIII. ACKNOWLEDGMENTS

This work is supported by the Labex ACTION program (contract ANR-11-LABX-01-01), ANR/RGC (contracts ANR-12-IS02-0004-01 and 3-ZG1F) and ANR (contract ANR-2011-BS03-005). The authors wish to express their appreciation to the four anonymous reviewers for their constructive comments.

References

[1] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, *Meld: A Declarative Approach to Programming Ensembles,* In Proceedings of the IEEE Int. Con. on Intelligent Robots and Systems, October, 2007.

[2] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, Padmanabhan Pillai, and Jason D. Campbell, *A Language for Large Ensembles of Independently Executing Nodes,* In Proc. of the Int. Con. on Logic Programming, 2009.

[3] J. Bourgeois and S.C. Goldstein. *Distributed Intelligent MEMS: Progresses and perspectives*, In the 3-rd Int. Conf. ICT Innovations, Ohrid, Macedonia, September, 2011.

[4] H. Bojinov, A. Casal, T. Hogg, *Emergent structures in modular self-reconfigurable robots,* Proceedings of the IEEE Int. Con. on Robotics and Automation, vol. 2, pp. 1734-1741, Los Alamitos, 2000.

[5] Z. J. Butler, K. Kotay, D. Rus, K. Tomita, *Generic decentralized control for lattice-based self-reconfigurable robots*, International Journal of Robotics Research 23(9):919-937, 2004

[6] D. Dewey, S. S. Srinivasa, M. P. Ashley-Rollman, M. D. Rosa, P. Pillai, T. C. Mowry, J. D. Campbell, and S. C. Goldstein, *Generalizing Metamodules to Simplify Planning in Modular Robotic Systems,* In Proceedings of IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems, September, 2008

[7] T. Ebefors, J.U. Mattsson, E. K. lvesten, and G. Stemme, *A walking a silicon microrobot*, in The 10th Int. Conf. on Solid-State Sensors and Actuators (Transducers '99), pages 1202-1205, Sendai, Japan, June 1999.

[8] S. Funiak, P. Pillai, M. P. Ashley-Rollman, J. D. Campbell, and S. C. Goldstein, *Distributed Localization of Modular Robot Ensembles,* In Proceedings of Robotics: Science and Systems, June, 2008.

[9] C. Jones, M. J. Mataric, *From local to global behavior in intelligent self-assembly.In: Proceedings of the 2003 IEEE International Conference on Robotics and Automation*,vol. 1, pp. 721-726, Los Alamitos, 2003.

[10] S. Jeon, C. Ji, *Randomized Distributed Configuration Management of Wireless Networks: Multi-layer Markov Random Fields and Near-Optimality* CoRR abs/0809.1916, 2008.

[11] S. Hollar, A. Flynn, C. Bellew, and K.S.J. Pister, *Solar powered 10mg silicon robot*,In MEMS, Kyoto, Japan, January 2003.

[12] M. E. Karagozler, A. Thaker, S. C. Goldstein, D. S. Ricketts, *Electrostatic Actuation and Control of Micro Robots Using a Post-Processed High-Voltage SOI CMOS Chip*,IEEE International Symposium on Circuits and Systems (ISCAS), May 2011.

[13] K. Katoy, D. Rus, M. Vona, and C. McGray, *The Self-reconfiguring Robotic Molecule*,in Proceedings of the 1998 IEEE International Conference on Robotics and Automation, Leuven, 1998.

[14] F. Kribi, P. Minet, A. Laouiti, *Redeploying mobile wireless sensor networks with virtual forces*, IFIP Wireless Days, Paris, France,2009.

[15] H. Lakhlef, H. Mabed, J. Bourgeois, *Distributed and Efficient Algorithm for Self-reconfiguration of MEMS Microrobots*, in the 28th ACM Symposium On Applied Computing, Coimbra, Portugal, March 2013.

[16] H. Lakhlef, H. Mabed, J. Bourgeois, *Dynamicity to Save Energy in Microrobots Reconfiguration*, in 10th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC-2013), Italy, December 2013.

[17] H. Lakhlef, H. Mabed, J. Bourgeois, *Parallel Self-reconfiguration for MEMS Microrobot*, in the 7-th IEEE Region 8 International conference on Computer as a Tool, Zagreb, Croatia, July 2013.

[18] H. Lakhlef, H. Mabed, J. Bourgeois, *Distributed and Dynamic Map-less Self-reconfiguration for Microrobot Networks*, 12th IEEE International Symposium on Network Computing and Applications (NCA 2013), P. 55-60, Cambridge, MA, United States, 2013.

[19] R. Ravichandran, G. Gordon, and S. C. Goldstein: *A Scalable Distributed Algorithm for Shape Transformation in Multi-Robot Systems,* In Proceedings of the IEEE Int. Con. on Intelligent Robots and Systems, October, 2007.

[20] M. D. Rosa, S. C. Goldstein, P. Lee, J. D. Campbell, and P. Pillai, *Programming Modular Robots with Locally Distributed Predicates,* In Proceedings of the IEEE Int. Con. on Robotics and Automation, 2008.

[21] D. Rus, M. Vona, *Crystalline robots: Self-reconfiguration with compressible unit modules*,Autonomous Robots 10(1), 107-124, 2001.

[22] E. Sahin. robotics: from sources of inspiration to domains of application, Swarm Robotics, SAB 2004 International Workshop (Revised Selected Papers) E. Sahin and W. M. Spear (Eds.), Lecture Notes in Computer Science 3342, Springer, 2005.

[23] R. Soua, L. Saidane, P. Minet, *Sensors deployment enhancement by a mobile robot in wireless sensor networks*,IEEE ICN 2010, Les Menuires, France, April 2010.

[24] K.Stoy, R.Nagpal, *Self-reconfiguration using Directed Growth*, 7th International Symposium on Distributed Autonomous Robotic Systems (DARs), France, June23-25, 2004.

[25] W. Shen, P. Will and A. Galstyan, Hormone-inspired self-organization and distributed control of robotic swarms. Autonomous Robots 17(1), 93-105, 2004.

[26] B. Warneke, M. Last, B. Leibowitz, and K.S.J Pister,K.S.J., 2001, *Smart Dust: Communicating with a Cubic-Millimeter Computer*,Computer Magazine, pp. 44-51, 2001.

[27] P. White, V. Zykov, J. C. Bongard, H. Lipson, *Three dimensional stochastic reconfiguration of modular robots* In: Proceedings of Robotics Science and Systems, pp. 161-168. MIT Press, Cambridge , 2005

[28] F. Zambonelli, M.P. Gleizes, M. Mamei, R. Tolksdorf,*Spray Computers: Explorations in Self-Organization*, Journal of Pervasive and Mobile Computing, Elsevier, Vol. 1, p. 1-20, 2005.

[29] S. Wong and J. Walter, *Deterministic Distributed Algorithm for Self-Reconfiguration of Modular Robots from Arbitrary to Straight Chain Configurations*, IEEE International Conference on Robotics and Automation, Germany, May 2013

[30] *http://today.duke.edu/2008/06/microrobots.html*

[31] *http://www.pittsburgh.intel-research.net/dprweb*

[32] *http://smartblocks.univ-fcomte.fr/*