# Experimental Analysis of Programmable Particles

by

Hector Eduardo Dominguez Berdugo

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Computer Science

Ottawa - Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

# Abstract

Autonomous mobile computational entities have been extensively studied in distributed computing. Particularly, studies in Programmable Particles – simple and homogeneous computational elements – have recently increased as they could be used in many situations [11] (e.g., minimally invasive surgeries, autonomous monitoring and repair systems). These particles are able to establish and release bonds, and can move in a self-organizing way to solve a particular problem by means of local interactions.

Several simulators have been implemented to work with distributed algorithms in different types of networks (e.g., *JBotSim* [2], *VisibleSim* [9]) but none of them has been designed especially for Programmable Particles and has been made public.

In this thesis, I will present and experiment with a new multiplatform simulator for Programmable Particles, where the user is able to customize the number of particles in the system, the environment and the algorithms executed by each of them easily.

# Acknowledgements

Firstly, I would like to thank God for giving me the strength, knowledge, ability and opportunity to start and finish these studies. Without his blessings, this achievement would not have been possible.

Secondly, my sincere and deep thanks to my supervisor, Dr. Nicola Santoro, for his continuous support, patience, motivation and guide on this work.

Lastly but not least, I thank all my family for their endless support through these years, my lovely wife, Maria, who has been always walking with me along this way; my parents, Magdalena and Eduardo, who always believed in me and blessed me with their love from the distance; my little brother, Luis, who has inspired me and encouraged to be confident, no matter the country where I live.

This work is for all of them.

# Table of Contents

# List of Algorithms

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Background

The multidisciplinary field of Programmable Matter encompasses researchers from biology, ethology, robotics, control, and computer science, envisioning a future availability of very large numbers of very small (micro- or nano-sized) programmable particles. Recent advances in micro-fabrication and cellular engineering render the production of such particles increasingly possible, and the advent of such a future more realistic. The envisioned applications of such particles include a variety of important situations: smart materials, autonomous monitoring and repair, minimal invasive surgery, etc.

Take for example the inspection of large structures (e.g., bridges, tunnels) looking for defects; this task, paramount for the safety of many users is nowadays performed by humans, usually engineers. Most of the times, these tasks are both time-consuming and costly. Thanks to the use of programmable matter, we can envision the accomplishment of these tasks in a faster and cheaper way, but also with higher levels of safety for both: engineers and users. In general, particles will be able to self-organize in order to achieve a collective goal without the need of a

central control or external intervention, particularly, human intervention [28]. For example:

- ➔ Programmable matter could identify, coat and possibly repair cracks on bridges or tunnels, but also leaks on nuclear reactors without human intervention.
- ➔ Programmable matter could monitor structural conditions in mines and airplanes without human intervention.
- ➔ Programmable matter could be used within the human body to detect and coat the area where an internal bleeding occurs or they could also identify tumor cells.

Therefore, given that there may be many possible applications in the future, the study of programmable matter is necessary.

There are many, widely different, visions of the computational nature of this programmable matter, its capabilities, its fields of existence and application. In general, these kind of systems are classified as passive and active systems [28].

Particles in *passive* systems do not have any intelligence at all, they just move and bond based on their structural properties or as a result of chemical interactions; they might have limited computational capabilities, but they cannot control their movements [31]. Some research examples in passive systems can be found in DNA Computing, Tile Self-assembly Systems, Population Protocols and

Slime Molds. In DNA Computing, studies from Adleman [17] are very famous, showing the feasibility of carrying out computations at the molecular level by solving an instance of the Directed Hamiltonian Path problem using experiments, where a small graph is encoded in molecules of DNA. For Tile Self-assembly Systems, one can find many articles [7] [8] [18] describing natural processes in which small components automatically assemble themselves into large and complex structures (e.g., lipids self-assemble a cell's membrane and crystal formation). In Population Protocols, Angluin et al. [6] study the computational power of networks of small resource-limited mobile agents and proposes two models of computations based on pairwise interactions of finite-state agents. For Slime Molds, Li et al. [16] design a self-organizing path formation protocol inspired in these amoeba-like organisms (*physarum polycephalum*) to facilitate data transfers in the context of Wireless Sensor Networks (WSNs).

Among the *active* systems, the focus has been on *programmable particles*, computational units that can be programmed to act and move in order to solve a specific task by means of local interactions [25]. Within distributed computing, the main model of programmable particles is the *Geometric Amoebot model*, introduced by Derakhshandeh et al. [31], inspired by the behavior of the amoeba, and viewed as a swarm of decentralized autonomous self-organizing entities (or particles), operating on an hexagonal tessellation of the plane [28]. The focus of this thesis is precisely within the Geometric Amoebot model, continuing the ongoing research process.

Several problems have been studied and solved in this model (e.g., [11] [12] [28] [29] [30] [31]). Of particular interest is the work on *shape formation* [12] [21] [30], which is the analogous of the classical class of *pattern formation* problems studied in the field of distributed computing by mobile computational entities [11], requiring the movement of these entities in a defined space until they form a pattern in a finite time.

All of this research has been done from a theoretical point of view. Indeed there are no ready-made specific tools to experiment with those algorithms. JBotSim [2], for example, is a simulator designed for distributed algorithms in dynamic networks, representing a different setting from the studied in this thesis and making impossible its use. Similarly, glSwarm [5] and Sycamore [26] are simulator designed to solve swarming and flocking problems; therefore, their use would lead to the re-implementation of almost all modules, including interfaces to draw particle movements on an hexagonal tessellation of the plane.

The use of such tool would be useful for many reasons, among them:

- ➔ Knowing whether the studied or designed algorithm really solves the problem for which it was created for.
- ➔ Sometimes, it is quite difficult to perform complexity analysis theoretically, but experiments allow algorithm designers to evaluate the efficiency of

algorithms under different scenarios, giving the ability to compare different solutions at the same time.

➔ Observing visually the behavior of particles could help during the design process.

## 1.2 Contributions

In this thesis, we study a composite programmable entity, called *Curling Caterpillar*, introduced and defined in [21], which operates and move in the space defined by the Geometric Amoebot model. Composed of a bonded sequence of programmable units, it has two ends, one acting as the head and the other as a tail. A caterpillar has two stationary states: distended and curled. When *distended* it is a line, when *curled* it forms a compact ball. The units are bonded in the sense that each unit in the sequence is permanently connected to its two neighbours (except for the head and tail that have a single neighbour). The basic operation of a caterpillar is to move into space by transitioning between these two states.

The problem we consider is that of implementing a Curling Caterpillar using amoebots: each caterpillar unit is a programmable particle (i.e., an Amoebot). In particular, we study the problem of making the caterpillar move from a distended state into a curled one. This can also be described as the problem of a line of amoebots having to form a compact ball without ever breaking connection with their original neighbours in the line.

The main contribution of this thesis is the design, implementation and experimental analysis of two algorithms for programmable particles to solve the Compact Ball formation problem for Curling Caterpillars. Specifically, I implemented the *Caterpillar-1* algorithm presented in [21], discovering and correcting some mistakes and inaccuracies in the process; I designed and implemented a new algorithm to solve the same problem, called *Caterpillar-1M*. Additionally, I illustrated through experiments the advantages of *Caterpillar-1M* using different settings (e.g., schedulers, number of particles, head position).

For the implementation and the experimental analysis, in absence of a simulator specific for the programmable particles model, I designed and developed a simulator, called *ProgPa Simulator*. Its aim is to let researchers test algorithms for programmable particles, under different situations and system configurations, finding point of weakness or, at least showing that everything works empirically.

## 1.3 Thesis Organization

This work begins with the theoretical basis of the thesis presented in Chapter 2. It describes more accurately the elements briefly presented in previous sections such as particles capabilities, properties, memories, agreements and system schedulers along with a brief literature review.

Then, Chapter 3 describes the implementation details for the *Caterpillar-1* [21] and *Caterpillar-1M* algorithms.

Chapter 4 shows the experimental results of this work, particularly, the results from the *Caterpillar-1* [21] and *Caterpillar-1M* algorithms implementation on the *ProgPa Simulator*; following by conclusions and discussions of the potential future work of this thesis in Chapter 5.

Appendix A contains manuals to download and install the *ProgPa Simulator* to reproduce experiments along with instructions to extend and implement new features (e.g., schedulers, algorithms, UI elements). Appendix B represents an extension of Chapter 4, with a more detailed and comprehensive results of the simulations and comparisons performed in this work.

# Chapter 2

# Model, Terminology, and System

This chapter introduces the theoretical basis on which the thesis has been built on.

## 2.1 The Geometric Amoebot Model

The *Geometric Amoebot* model is an abstract computational model for programmable matter, and it was designed and intended to enable rigorous algorithm analysis of collective systems at the nano-scale [15]. In this model, programmable matter consists of *particles*: a uniform set of simple computational entities that operate and move in a hexagonal tessellation of the Euclidean space (i.e., in an infinite triangular grid $G_{eqt}$), see Figure 2.1 (a).


(a) (b)

**Figure 2.1**: (a) A portion of $G_{eqt}$. (b) Particles with different offsets on labelings [28]

Each particle occupies either a single node (i.e., it is *contracted*) or a pair of adjacent nodes (i.e., it is *expanded*); and every node can be occupied by at most one particle as in Figure 2.2. Locomotion is achieved through *expansions* and *contractions* and can be described as follow: a *contracted* particle can expand into an empty adjacent node to become *expanded* and occupy two nodes and then, the movement is complete when the expanded particle contracts again to occupy a single node. While a contracted particle has just one component, an expanded particle can be described by two components: the *head* – defined as the node that it last expanded into, and the *tail* – the other node.

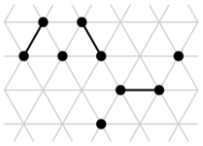


**Figure 2.2**: Expanded and Contracted particles
in a Geometric Amoebot Model [15]

*Neighbors Particles* are two different particles occupying adjacents nodes; and coordination during the movement can be achieved by a *handover*, where two scenarios are possibles:

1. A contracted particle $P$ can push a neighboring expanded particle $Q$ and expand into one of the previously occupied node by $Q$, forcing $Q$ to contract.
2. An expanded particle $P$ can pull a neighboring contracted particle $Q$ to a node occupied by it, forcing $Q$ to expand and $P$ to contract.

Particles can bond to other particles, using such bonds to exchange information. Bonds have unique labels, therefore each particle can uniquely identify each of its outgoing edges, see Figure 2.1 (b).

Each of these particles has two kind of memories: a local private memory to store its state and a local shared memory that can be read and written by neighbor particles. Computationally the particles are finite-state machines and their memory contains only a constant number of bits. Particles are anonymous (there are no unique identifiers), do not have any global information (e.g. the total number of particles in the system).

This model has been recently used in the study of different problems such as leader election [28] [31], shape formation [12] [21] [30], shape recovery [11] and coating problems [28] [29].

In some of this work, it is also assumed that particles have a common *chirality* [28] [29] [30] [31] meaning that all have the same notion of *clockwise direction*, used to enumerate their port; however, they do not have a common sense of orientation. Most recent work does not rely on such assumption [11] [12].

## 2.2 Activation Scheduler

In systems of computational mobile entities (be they autonomous robots or programmable particles) a very important element is the activation scheduler; that

is the set of assumptions made on when an entity becomes active (i.e., executes its program to determine the action to take and possibly move) and which entities are activated at which times.

The Geometric Amoebot model is *synchronous*: time is divided in a discrete sequence of rounds; at each round, a subset of the particles is activated, each simultaneously executes the program (possibly contracting or expanding), terminating at the end of the round.

Which set of entities is activated at what round is decided by an adversary, called *scheduler*. Different schedulers could bring different behaviors in the system, making necessary the use of different algorithms in some cases. From a theoretical point of view, there could be many kind of schedulers, but the most commonly used are following:

**The Semi-Synchronous Scheduler** (SSYNC): in this scheduler, the set of activated in a round is arbitrary, but each particle is activated infinitely often. This is the strongest possible adversary and has been used in [11] [12].

**The Sequential Scheduler** (SEQ): it assumes that in each round only one particle is activated; additionally, each particle is activated infinitely often. This much weaker adversary is assumed in [28] [29] [30] [31], where it is confusingly called “asynchronous”.

**The Fully Synchronous Scheduler** (FSYNC): in this scheduler, all particles are activated in every round is arbitrary. This scheduler has been assumed in [10].

One can obviously define other schedulers; e.g., *k-bounded* schedulers, *k-fair* schedulers and *centralized* schedulers [23]. The assumed scheduler could produce very different results on a system.

For the purpose of this work, we take into consideration a FSYNC scheduler and two variants of Sequential schedulers: a Sequential Round-Robin (SRR) scheduler, where particles are activated in a fixed cyclic order; and a Sequential Random (SR) scheduler, where particles are activated in a random but fair order.

## 2.3 Related Work

### 2.3.1 Existing Results

Among the problems studied using programmable particles are Coating problems, where the idea is to have a layer of particles covering an object of any shape so that it would be possible to measure conditions such as temperatures or cracks in any point of the object without requiring human intervention. In this context, Derakhshandeh et al. [29] propose a worst-case work-optimal universal coating algorithm that allows uniform coating for objects of arbitrary shape and size.

Similarly, Derakhshandeh et al. [31] propose an algorithm for the Leader Election problem in the context of programmable particles where, given a set of particles, the goal is to select one of those particles as the leader. Many problems like the consensus problem where all particles have to agree on some output value, can be solved once a leader particle has been chosen. In their work, Derakhshandeh et al. show that there is no distributed algorithm to solve the leader election problem under the General Amoebot model, but they also show that it is possible under the Geometric Amoebot model.

Comparably to the consensus problem, researchers have observed that shape formation problems can also be solved once there is a leader, given that most shape formation algorithms depend on some seed element. Derakhshandeh et al. [31] take advantage of this and propose a distributed algorithm that solves the line formation problem with work-case optimal work, assuming the existence of an arbitrary connected structure of contracted particles with a unique leader.

In a similar direction, Di Luna et al. [12] describe a universal shape formation algorithm for systems of particles, establishing that, given a shape $S_F$, the system can form a shape geometrically similar to $S_F$, deterministically and starting from any simply connected configuration $S_0$.

### 2.3.2 Related Models

Although substantially different in settings and assumptions, there are some research areas which are close in spirit to the one of programmable particles. For

example, in Swarm Robotics it is assumed the existence of a collection of autonomous robots that operate and freely move in 2D and 3D Euclidean spaces; they are computationally very powerful (Turing Machines). These kind of robots have been used to solve different problems such as graph exploration [22], gathering [4] and shape formation problems [24] but they as an example, Flocchini et al. [22] consider and analyze the problem of exploring an anonymous ring by oblivious, anonymous and asynchronous robots scattered in the ring where robots can see the environment but they are unable to communicate. Agathangelou et al. [4] study the problem of gathering any number of autonomous (*fat* - non-transparent unit-disc) robots under the assumption of agreement on the orientation of axes of their local coordination system and full visibility (all robots can see all other robots). Flocchini et al. [24] study the Arbitrary Pattern Formation problem using anonymous robots without communication nor memory but with full visibility. This work shows a strongly dependence between the system performance and the common agreement of robots in regards to their environment.

In the field of Modular Robotic Systems, where the focus is mainly related to aspects such as the design, fabrication and motion planning of autonomous machines; there have been studies of self-reconfiguring robots that are able to change their shape, re-arranging the connectivity of their parts in order to perform new tasks, recover from damage and adapt to new circumstances [31]. For example, Chirikjian [13] presents the concept and kinematics of Metamorphic Robotic Systems viewed as a subclass of self-reconfigurable robots and as a large

swarm of physically connected robotic modules which collectively act as a single entity; and showing the following properties: self-reconfigurability without external help, a large number of homogeneous modules and physical constraints to ensure contact between modules.

Similarly to the Amoebot Model, the Nubot model has been presented by Woods et al. [14]. It is inspired in biomolecules and molecular programming and it is defined as an asynchronous, non-deterministic cellular automaton augmented with rigid-body movement rules (push / pull) and random agitations. It has been used to build simple shapes such as lines and squares but unlike the Amoebot Model, it is possible to add new particles to the system as needed.

# Chapter 3

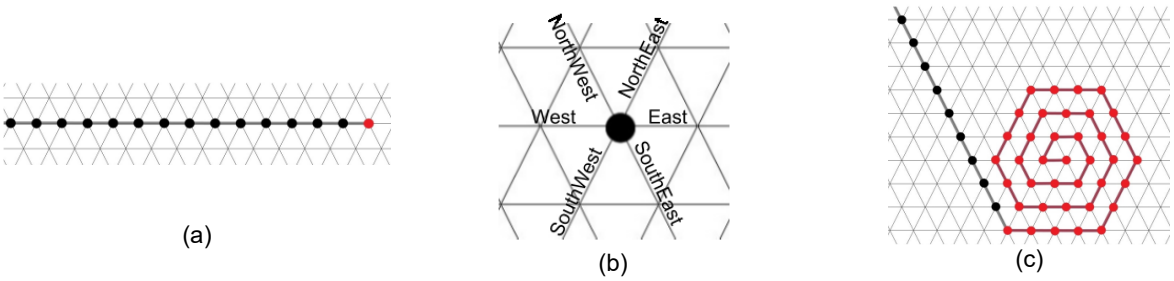
# Experiments

This chapter describes and presents the algorithms implemented and used to experiment with the *ProgPa Simulator*.

## 3.1 Curling Caterpillars

Flocchini et al. [21] have recently created the Curling Caterpillars problem definition based on the Amoebots model described in Chapter 2 where, additionally to the assumptions described there, it assumes that a fully connected line of contracted-follower particles exist with a leader located in one of the two ends of the line. This line is called Caterpillar, see Figure 3.1 (a). Moreover, it also assumes that each particle shares a global orientation used to name its port numbers in a counterclockwise manner as follow: East, North East, North West, West, South West and South East. See Figure 3.1 (b).

The objective for the particle system is to form a *Compact Ball* in a self-organized way keeping the bonding between particles in all the process. A *Compact Ball* is defined as a circular rigid shape, where there exists a central particle coated by one or more layers of particles [21], and each layer consists of equidistant particles to the central node. See Figure 3.1 (c).




(a) (b) (c)

**Figure 3.1**: (a) Caterpillar: initial configuration, the red particle represents the leader [21] (b) Global orientation, (c) Compact Ball in a given point of the solving process [21]

### 3.1.1 Algorithm: Caterpillar-1

Caterpillar-1 is the first version of the algorithm presented by Flocchini et al. [21], where each particle can be in six different states: Core *Leader*, *First Root*, *Root*, *Follower*, *Retired* and *Terminated*; particles can start moving on their own only when they are in the *Root* state, there is no memory on particle ports and each particle can send seven type of messages: *Root-Activation-0, Root-Activation*, *Ack*, *Complete*, *Termination*, *Rotate* and *Fill*.

First of all, during the description of this algorithm, it is important to keep in mind the following definitions: the *next particle* of a given particle is defined as the neighbor particle in the direction of the leader; while the *previous particle* is defined as the neighbor particle in the opposite direction.

Thus, this algorithm progresses by rotating the line of followers in an iterative, clockwise and bottom-to-up manner and the *Root* particle is in charge of this task; first, finding the next unoccupied node that should be filled and then, writing *Fill* on the corresponding port showing this selection.

After this action, if the *Root*'s immediate follower is not in the selected node, then it rotates the line of followers until it is placed in the direction pointed by *Fill*. Each line rotation is performed as follow:

The *Root* particle sends a *Rotate* message to its immediate follower neighbor in the line, then this message is forwarded until the last follower particle receives it. When this happens, the last follower moves one step clockwise around the next bounded neighbor and it checks if it is pointed by *Fill*. If it is pointed by *Fill*, it sends a *Complete* message to its next immediate neighbor. In other case, it sends an *Ack* message to its next immediate neighbor.

Any follower receiving an *Ack* message, moves one step clockwise around its next bounded neighbor and checks if it is pointed by *Fill*. If it is pointed by *Fill*, it writes *Fill* on the port that is opposite to the port marked by *Fill*, then it sends a *Rotate* message to its previous bonded neighbor in the line. In other case, it just forwards the *Ack* message to its next bonded neighbor.

Once the *Root* particle receives an *Ack* message, it replies back with a *Rotate* message. If a follower receives a *Complete* message, it just forwards it to its next bonded neighbor; but if it is the *Root* who receives a *Complete* message, then it understands that the rotation process finished and the line of followers is placed in the proper direction. See [21] for more details about the rotation process.

On the other hand, the process of forming the *Compact Ball* is started by the *Core Leader* particle, sending a *Root-Activation-0* message to its connected follower, and the receiver of this message becomes *First Root*, finds the next unoccupied node and then, rotates the line of followers as described before. Once this is done, it sends a *Root-Activation* message to the previous follower particle and becomes *Retired*.

The receiver of a *Root-Activation* message becomes *Root*, finds the next unoccupied node and then, if it is necessary, it rotates the line of followers as described before. See [21] for more details about process of forming the *Compact Ball*.

During the termination phase, the last *Retired* particle sends a *Termination* message to its retired neighbor and becomes *Terminated*. In general, every *Retired* particle receiving this message forwards it to their next *Retired* particle and becomes *Terminated*, and once the first *Retired* particle (initially, the *Core Leader* particle) receives it, it becomes *Terminated* and the algorithm finishes. A detailed description of this algorithm[1] is presented in Algorithm 3.1 and Figure 3.2 illustrates an example of a state implementation in C++ from the corresponding pseudocode.

[1] An implementation on C++ is available in: http://bit.ly/caterpillar-1

**Algorithm 3.1**: Caterpillar-1 - states and actions

**Core Leader:**
- Send *Root-Activation-0* to its bonded follower neighbor
- Become *Retired*

**First Root:**
- **if** it's the last particle
  - Send *Termination* to its next bonded neighbor
  - Become *Terminated*
- **else if** it hasn't chosen the next position to fill
  - Choose port 1 as next position to fill when the leader is the rightmost particle or choose port 4 when leader is the leftmost particle
  - Send *Rotate* to its previous bonded neighbor
- **else if** received a message
  - **if** received *Ack*
    - Send *Rotate* to its previous bonded neighbor
  - **if** received *Complete*
    - Send *Root-Activation* to its previous bonded neighbor
    - Become *Retired*

**Root:**
- **if** it's the last particle
  - Send *Termination* to its next bonded neighbor
  - Become *Terminated*
- **else if** it hasn't chosen the next position to fill
  - **if** it has 4 neighbors
    - Send *Root-Activation* to previous bonded neighbor
    - Become *Retired*
  - **else**
    - Choose the port before to the port pointing to the previous bonded neighbor as the next position to fill
    - Send *Rotate* to its previous bonded neighbor
- **else if** received a message
  - **if** received *Ack*
    - Send *Rotate* to its previous bonded neighbor
  - **if** received *Complete*
    - Send *Root-Activation* to its previous bonded neighbor
    - Become *Retired*

**Follower:**
- **if** receive *Root-Activation-0*
  - Become *First Root*
- **if** receive *Root-Activation*
  - Become *Root*
- **if** receive *Ack*
  - Move clockwise

```
        - if it is pointed by Fill
            - Write Fill in the opposite port
            - Send Rotate to its previous bonded neighbor
        - else
            - Forward Ack to its next bonded neighbor

    - if receive Complete and it's not the first particle
        - Forward Complete to its next bonded neighbor

    - if receive Rotate
        - if it's not the last particle
            - Forward Rotate to its previous bonded particle
        - else
            - Move clockwise
            - if it's pointed by Fill
                - Send Complete to its next bonded particle
            - else
                - Send Ack to its next bonded particle

Retired:
    - if receive Termination
        - if it's not the first particle
            - Forward Termination to its next bonded particle

        - Become Terminated
```

**Core Leader:**

- Send *Root-Activation-0* to its bonded follower neighbor

- Become *Retired*

(a)

```
void Particle::coreLeader (Particle * next, Particle * previous,
                           std::map<std::string, bool> * neighborhood) {
  /*
   * Algorithm for particles in the 'Core Leader' state
   *
   */

  previous->receivedMessages[0] = ROOT_ACTIVATION_0_MSG;

  _state = RETIRED_STATE;
  _sprite->setTexture(RETIRED_PARTICLE_IMG);
}
```

(b)

**Figure 3.2**: Core Leader state implementation for Caterpillar-1: (a) pseudocode (b) C++ implementation

### 3.1.2 Algorithm: Caterpillar-1M

The creation of the Caterpillar-1M algorithm is also part of this work and it was inspired by the algorithm described in the previous section. In this new version, each particle can be in five different states: *Leader*, *Root*, *Follower*, *Retired* and *Terminated*; each particle shares a memory with neighboring particles for

communication, there is no memory on the graph nodes and particles can send five type of messages: *Root Activation*, *Acknowledge*, *Complete*, *Termination* and a *small integer number*.

Similarly to the previous algorithm, this algorithm progresses by rotating the line of followers in an iterative, clockwise and bottom-to-up manner. Thus, keeping that in mind, the *next particle* of a given particle is defined as the neighbor particle in the direction of the leader; conversely, the *previous particle* of a given particle is defined as the neighbor particle in the opposite direction of the leader.

The algorithm is actually started when the leader particle is activated, sending a *Root Activation* message to its follower and then becoming *Retired*. Then, any follower receiving a *Root Activation* message becomes *Root*, but if it receives an *Acknowledge* or *Complete* message, then it forwards the message to the next particle and updates its local private memory if it is necessary.

Following this method, a particle can only be *Root* when it is already positioned in its final node and its task is to rotate the line of followers until the *previous* particle is in its final position; when this happens, it sends a *Root Activation* message to the *previous* particle and becomes *Retired*. While this is not the case, the *Root* particle either: waits for a *Complete* message or it computes the number of times it should rotate its line of followers based on the number of particle neighbors.

In the cases where it is necessary more than one rotation (e.g., the first root particle in the solving process), these are performed one at a time. In other words, the *Root* particle waits for a complete rotation of followers before starting a new one.

The number of rotations necessary is calculated once by the *Root* particle and then stored in its local private memory. Basically, this is the task performed during its first activation round as a *Root* and it is determined as follow: $numRotations = 4 - numNeighbors$.

As soon as this information is determined, the *Root* particle notifies to its *previous* particle that it should move one step clockwise. When a *Follower* receives this information, it uses its local private memory to store it and similarly, it notifies to its *previous* particle the number of clockwise steps that it should perform.

*Follower* particles can only rotate when they receive an *Acknowledge* or *Complete* message, updating its local private memory with the remaining steps to be in its final position. At least, it is the last follower particle, that rotates every time it is activated when it knows the remaining steps to be in its final position.

Once the last follower particle is located in its final position, it sends a *Complete* message to the *next* particle but, if a *Root* particle realizes that it is the last *Root* in

the algorithm (it does not have a *previous* particle), then it sends a *Termination* message and becomes *Terminated*.

Finally, if a *Retired* particle receives a *Termination* message, it forwards the message to the *next* particle and becomes *Terminated*. In a similar way as it was described for the previous algorithm, all clockwise rotations are performed with respect to the *next* particle and, if it is determined that a movement would break the line connectivity, then it is avoided. A detailed description of this algorithm[2] is presented in Algorithm 3.2 and Figure 3.3 illustrates an example of a state implementation in C++ from the corresponding pseudocode.

[2] An implementation on C++ is available in: http://bit.ly/caterpillar-1m

**Algorithm 3.2**: Caterpillar-1M - states and actions

**Core Leader:**
- Send *Root Activation* to previous particle
- Become *Retired*

**Follower:**
- **if** it's the last follower
  - **if** there is no rotations remaining
    - Send *Complete* to next particle
  - **else**
    - Move clockwise
    - Send *Acknowledge* to next particle
  - Update local private memory

- **if** receive an *Acknowledge* or *Complete* message:
  - Move clockwise
  - Forward message to next particle
  - Update local private memory
- **else if** receive a *Root Activation* message:
  - Become *Root*
- **else**
  - Process and save message in local private memory
  - **if** it's not the last particle:
    - Send number of rotations to previous particle

**Root:**
- **if** it's the last *Root* particle:
  - Send a *Termination* message to the next particle
  - Become *Terminated*

- **if** it doesn't know the number of rotations:
  - Compute the number of rotations and store it in local private memory
  - **if** it's necessary rotate the line of followers:
    - Send number of movements to previous follower
  - **else**
    - Send a *Root Activation* message to previous follower
    - Become *Retired*

- **if** receive a *Complete* message:
  - Update local private memory
  - **if** it's remaining rotations:
    - Send number of movements to previous follower
  - **else**
    - Send a *Root Activation* message to previous follower
    - Become *Retired*

**Retired:**
- **if** receive a *Termination* message:
  - **if** there is a next particle:
    - Forward message to next particle
  - Become *Terminated*

```
Retired:
   -  if receive a Termination message:
         -  if there is a next particle:
               -  Forward message to next particle
         -  Become Terminated
```

(a)

```
void Particle::retired (Particle * next, Particle * previous,
                        std::map<std::string, bool> * neighborhood) {
  /*
   * Algorithm for particles in the 'Retired' state
   *
   */

  if (_messages.empty())
    return;

  // read message
  auto msg = _messages.front();
  _messages.pop();

  if (msg == TERMINATION_MSG) {
    if (next) {
      next->receiveMessage(TERMINATION_MSG);
    }

    _state = TERMINATED_STATE;
    _sprite->setTexture(TERMINATED_PARTICLE_IMG);
  }
}
```

(b)

**Figure 3.3**: Core Leader state implementation for Caterpillar-1M: (a) pseudocode (b) C++ implementation

## 3.2 Experiments

The experiments performed in this work, consisted in the comparison of different settings for each algorithm independently, such as the number of particles in the system, simulation speed, leader position and type of scheduler. These experiments were performed five times using the same settings and then results were averaged to minimize any bias as a result of the use of random numbers or any random event in the machine where they were performed.

Additionally, each algorithm was compared against each other under the same circumstances and taking into consideration the same metrics: simulation time, number of particle activations, number of particle movements and number of messages exchanged between particles. The characteristics of the machine used for experiments are the followings:

- ➔ Personal computer
- ➔ Processor: Intel® Core™ i7-6700HQ CPU @ 2.60GHz × 8
- ➔ 16GB DDR4 RAM
- ➔ 250 GB SATA 6.0 Gb/s Internal Solid State Drive
- ➔ Intel® HD Graphics 530 (Skylake GT2)
- ➔ Ubuntu 18.04.1 LTS, 64-bit

# Chapter 4

## Experimental Results

This chapter shows the usage of the ProgPa simulation system to perform experiments and analysis with algorithms for programmable particles, concretely, the *Caterpillar-1* and *Caterpillar-1M* algorithms; providing different settings as it was described in the previous chapter.

Firstly, experiments consisted in comparing the influence of the number of particles in the system against the simulation time, the number of particle activations, the number of movements and the amount of message exchanged between particles. These comparisons were performed for both algorithms: *Caterpillar-1* and *Caterpillar-1M*, considering the location of the leader particle in the initial setting and three types of schedulers: Sequential Round-Robin (SRR-1), Sequential Random (SR) and Fully Synchronous (FSYNC).

Similarly, the same experiments were performed considering a slight change in the Sequential Round-Robin scheduler, called SRR-2 in the following sections, where the cyclic direction was changed to measure the influence on algorithms performance.

Appendix A contains the user manuals that can be used as a guide to reproduce these experiments and Appendix B represents an extension of this chapter, detailing all results collected from this phase.

## 4.1 Caterpillar-1

This section presents the results from the experiments performed specifically on the Caterpillar-1 algorithm.

### 4.1.1 Simulation Time

Table 4.1 compares the influence of the number of particles in system on the simulation time using the ProgPa Simulator with the *no show animation* setting, therefore, they represent the time needed by the algorithm to form the compact ball in the Curling Caterpillars problem.

**Table 4.1**: Caterpillar-1 - Simulation time without animation (sec)

| Num. of particles | SRR-1 | SRR-2 | SR | FSYNC |
|---|---|---|---|---|
| 10 | 1.02 | 1.02 | 1.02 | 1.01 |
| 20 | 1.03 | 1.04 | 1.06 | 1.03 |
| 30 | 1.07 | 1.06 | 1.10 | 1.05 |
| 40 | 1.15 | 1.16 | 1.20 | 1.08 |
| 50 | 1.30 | 1.28 | 1.38 | 1.11 |
| 100 | 3.98 | 4.01 | 4.84 | 1.58 |
| 200 | 33.63 | 33.65 | 44.54 | 6.77 |
| 300 | 136.64 | 136.62 | 183.32 | 23.52 |
| 400 | 371.89 | 371.87 | 494.66 | 61.59 |
| 500 | 857.36 | 857.39 | 1100.85 | 133.95 |
| 1000 | 9376.11 | 9376.12 | 12872.50 | 1539.27 |

From these results, it is clear that the Caterpillar-1 algorithm works better with a FSYNC scheduler and the slight change in the Sequential Round-Robin scheduler (SRR-2) does not have a significant influence in the algorithm performance. Figure 4.1 presents the simulation time trends as the number of particles increase in the system, showing a light exponential shape for sequential schedulers and the worst performance under the Sequential Random scheduler (SR).

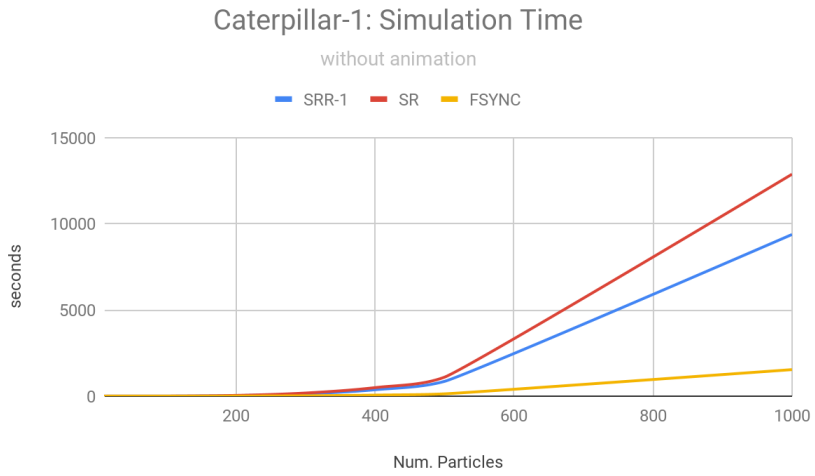



**Figure 4.1**: Caterpillar-1 - Simulation Time trend (without animation)

On the other hand, Table 4.2 presents similar results showing the influence of the speed setting while the animation is being shown during the simulation. Experiments consisted in using the original speed (1X), five times the original speed (5X) and ten times the original speed (10X). From these results, it is clear the following relation: $Simulation\ Time_{<s>X} \approx \frac{Simulation\ Time_{1X}}{<s>}$, where $< s >$ represents the speed setting chose by the user; for example, for $S = 5$,

$Simulation\ Time_{5X} \approx \frac{Simulation\ Time_{1X}}{5}$, similarly, for $S = 10$, $Simulation\ Time_{10X} \approx \frac{Simulation\ Time_{1X}}{10}$.

**Table 4.2**: Caterpillar-1 - Simulation time with animation (sec)

| Num. of particles | SRR-1 1X | SRR-1 5X | SRR-1 10X | SR 1X | SR 5X | SR 10X | FSYNC 1X | FSYNC 5X | FSYNC 10X |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 133.36 | 26.68 | 13.34 | 165.46 | 33.53 | 16.77 | 13.41 | 2.69 | 1.35 |
| 20 | 1747.73 | 349.55 | 174.79 | 2084.82 | 421.04 | 210.53 | 87.45 | 17.51 | 8.76 |
| 30 | 7648.27 | 1529.66 | 764.84 | 8988.68 | 1805.25 | 902.64 | 255.02 | 51.01 | 25.52 |
| 40 | 22546.59 | 4519.32 | 2160.59 | 25282.29 | 5049.55 | 2546.84 | 540.22 | 108.06 | 54.04 |
| 50 | 48159.13 | 9631.83 | 7437.12 | 56187.10 | 11236.12 | 5627.31 | 963.26 | 192.66 | 96.33 |

### 4.1.2 Particle Activations

The influence of the number of particles in system against the number of particle activations needed by the algorithm to solve the problem is presented in Table 4.3. These experiments showed the same results with and without the animation setting, returning a constant output in the five times running for all schedulers except for the Sequential Random scheduler (SR).

Additionally, it is clear that the algorithm needs one additional activation round when using the SRR-2 scheduler compared to the SRR-1 scheduler, following the relation: $num\_activations_{SRR-2} = num\_activations_{SRR-1} + num\_particles - 1$.

| **Table 4.3**: Caterpillar-1 - Number of Particle Activations | | | | |
|---|---|---|---|---|
| **Num. of particles** | **SRR-1** | **SRR-2** | **SR** | **FSYNC** |
| 10 | 1961 | 1970 | 2338.21 | 197 |
| 20 | 25701 | 25720 | 30366.03 | 1286 |
| 30 | 112471 | 112500 | 132388.12 | 3750 |
| 40 | 317721 | 317760 | 370648.04 | 7944 |
| 50 | 708201 | 708250 | 833342.19 | 14165 |
| 100 | 8402101 | 8402200 | 9800947.15 | 84022 |
| 200 | 98015401 | 98015600 | 114291256.03 | 490078 |
| 300 | 410367601 | 410367900 | 478839267.11 | 1367893 |
| 400 | 1131556401 | 1131556800 | 1318298697.10 | 2828892 |
| 500 | 2483226001 | 2483226500 | 2892253370.22 | 4966453 |
| 1000 | 28433203001 | 28433204000 | 33112003802.19 | 28433204 |

### 4.1.3 Particle Movements

Table 4.4 compares the influence of the number of particles in system against the particle movements needed to solve the problem and Figure 4.2 shows the trend of this influence. From these experiments, it is clear that the use of different types of schedulers does not impact the number of particle movements.

| **Table 4.4**: Caterpillar-1 - Number of Particle Movements | | | | |
|---|---|---|---|---|
| **Num. of particles** | **SRR-1** | **SRR-2** | **SR** | **FSYNC** |
| 10 | 150 | 150 | 150 | 150 |
| 20 | 1094 | 1094 | 1094 | 1094 |
| 30 | 3313 | 3313 | 3313 | 3313 |
| 40 | 7162 | 7162 | 7162 | 7162 |
| 50 | 12938 | 12938 | 12938 | 12938 |
| 100 | 79070 | 79070 | 79070 | 79070 |
| 200 | 470176 | 470176 | 470176 | 470176 |
| 300 | 1323041 | 1323041 | 1323041 | 1323041 |
| 400 | 2749090 | 2749090 | 2749090 | 2749090 |
| 500 | 4841701 | 4841701 | 4841701 | 4841701 |
| 1000 | 27933702 | 27933702 | 27933702 | 27933702 |

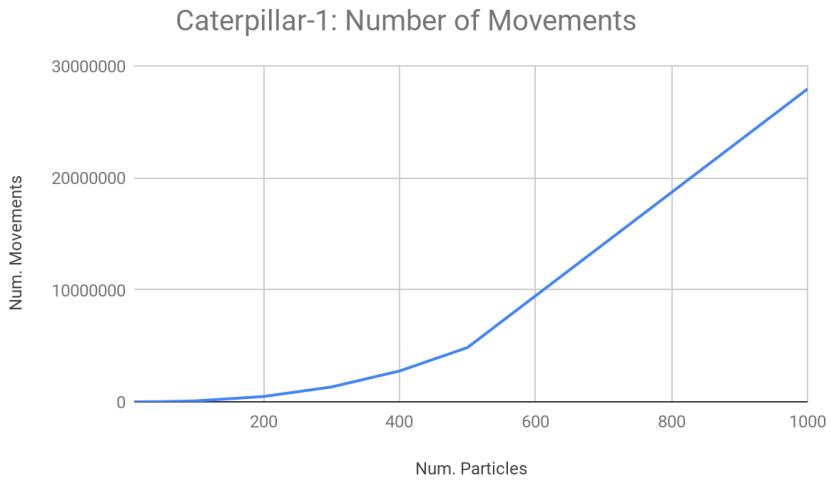



**Figure 4.2**: Caterpillar-1 - Number of movements trend

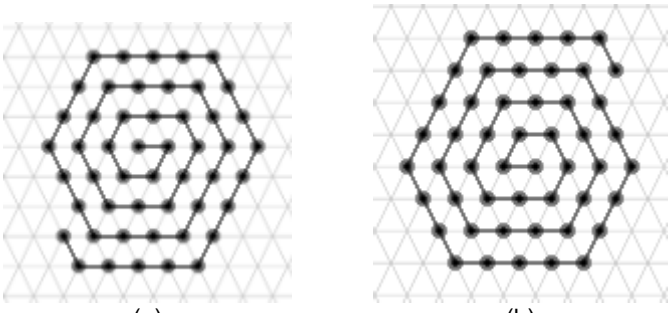
### 4.1.4 Messages

Similarly, Table 4.5 compares the influence of the number of particles in the system against the number of messages sent to solve the problem, showing that there is no influence of the type of scheduler as the last section.

| **Table 4.5**: Caterpillar-1 - Number of Messages | | | | |
|---|---|---|---|---|
| **Num. of particles** | **SRR-1** | **SRR-2** | **SR** | **FSYNC** |
| 10 | 407 | 407 | 407 | 407 |
| 20 | 2659 | 2659 | 2659 | 2659 |
| 30 | 7702 | 7702 | 7702 | 7702 |
| 40 | 16237 | 16237 | 16237 | 16237 |
| 50 | 28854 | 28854 | 28854 | 28854 |
| 100 | 169777 | 169777 | 169777 | 169777 |
| 200 | 985548 | 985548 | 985548 | 985548 |
| 300 | 2746086 | 2746086 | 2746086 | 2746086 |
| 400 | 5674001 | 5674001 | 5674001 | 5674001 |
| 500 | 9955911 | 9955911 | 9955911 | 9955911 |
| 1000 | 56933844 | 56933844 | 56933844 | 56933844 |

### 4.1.5 Leader Position

Experiments for this variable confirmed that the leader position does not influence in any of the other factors (e.g., simulation time, number of particle activations, movements and messages) given that root particles rotate the line of followers the same amount of times and following the same clockwise direction. Figure 4.3 illustrates the final compact ball for both leader positions: left and right.



(a) (b)

**Figure 4.3**: Caterpillar-1 - Compact Ball (a) left leader, (b) right leader

## 4.2 Caterpillar-1M

Similarly to the previous sections, this section presents the results from the experiments performed specifically on the Caterpillar-1M algorithm, following the same format presented before but excluding results for the Sequential Random scheduler (SR) given that experiments made evident that this algorithm does not solve the Curling Caterpillars problem using this kind of scheduler. Figure 4.4 shows some non-deterministic final shapes obtained as a result of using a SR scheduler.

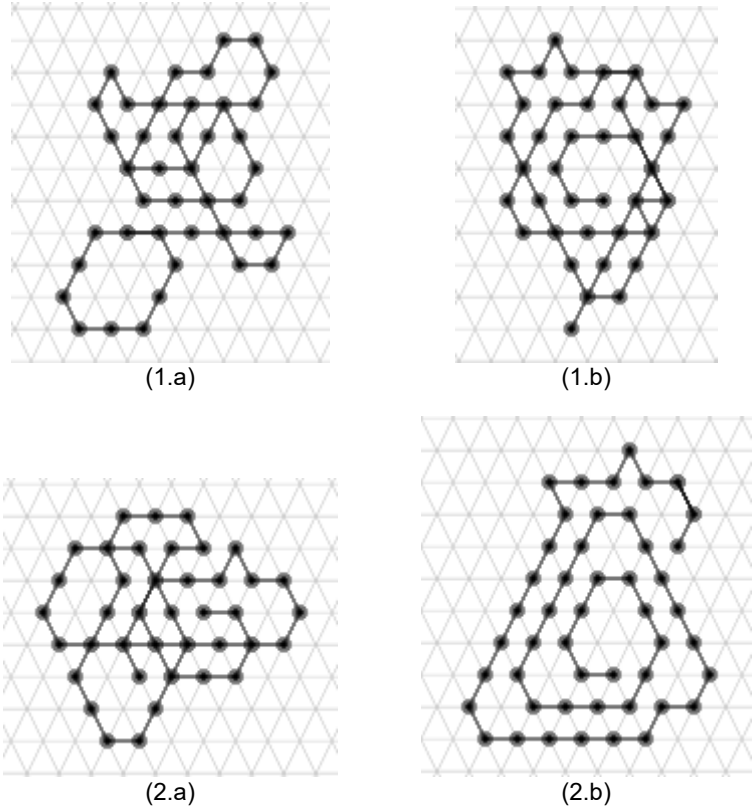



**Figure 4.4**: Caterpillar-1M - shapes under a SR scheduler (1.a), (2.a) left leader, (1.b), (2.b) right leader

### 4.2.1 Simulation Time

Table 4.6 compares the influence of the number of particles in system on the simulation time using the ProgPa Simulator with the *no show animation* setting. Again, similarly to the Caterpillar-1 algorithm, Caterpillar-1M works better with a FSYNC scheduler and the use of SRR-2 does not show a significant impact in the simulation time. Figure 4.5 presents the simulation time trends as the number of particles increase.

**Table 4.6**: Caterpillar-1M - Simulation time without animation (sec)

| Num. of particles | SRR-1 | SRR-2 | FSYNC |
|---|---|---|---|
| 10 | 1.02 | 1.02 | 1.02 |
| 20 | 1.03 | 1.02 | 1.02 |
| 30 | 1.04 | 1.04 | 1.03 |
| 40 | 1.05 | 1.05 | 1.04 |
| 50 | 1.1 | 1.09 | 1.07 |
| 100 | 1.34 | 1.35 | 1.2 |
| 200 | 2.76 | 2.77 | 2.07 |
| 300 | 5.96 | 5.94 | 3.96 |
| 400 | 11.32 | 11.30 | 7.18 |
| 500 | 19.12 | 19.13 | 11.79 |
| 1000 | 110.91 | 110.94 | 64.21 |

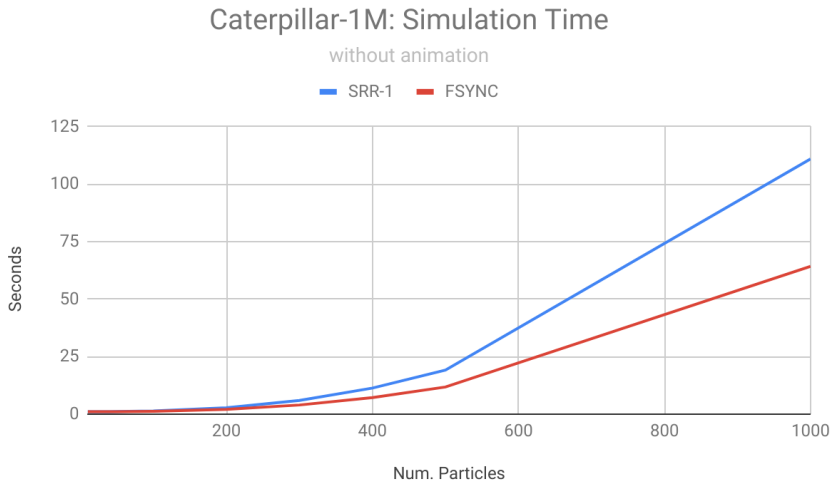



**Figure 4.5**: Caterpillar-1M - Simulation Time trend (without animation)

As the Caterpillar-1 algorithm, the influence of the speed setting while the animation is being shown during the simulation follows the relation: $Simulation\ Time_{<s>\ X} \approx \frac{Simulation\ Time_{1X}}{<s>}$, where $< s >$ represents the speed setting chose by the user. These results are presented in Appendix B.

### 4.2.2 Particle Activations

Table 4.7 shows how the number of particle activations is influenced by the number of particles in system and the use of different schedulers. Similarly to the Caterpillar-1 algorithm, experiments showed the same results with and without the animation setting but also showed that the algorithm needs one additional activation round when it is under a SRR-2 scheduler compared to the SRR-1 scheduler.

**Table 4.7**: Caterpillar-1M - Number of Particle Activations

| Num. of particles | SRR-1 | SRR-2 | FSYNC |
|---|---|---|---|
| 10 | 1081 | 1090 | 109 |
| 20 | 6801 | 6820 | 341 |
| 30 | 19621 | 19650 | 655 |
| 40 | 41241 | 41280 | 1032 |
| 50 | 73201 | 73250 | 1465 |
| 100 | 429201 | 429300 | 4293 |
| 200 | 2484801 | 2485000 | 12425 |
| 300 | 6915301 | 6915600 | 23052 |
| 400 | 14278001 | 14278400 | 35696 |
| 500 | 25039501 | 25040000 | 50080 |
| 1000 | 142973001 | 142974000 | 142974 |

### 4.2.3 Particle Movements

Experiments in this section compared the influence of the number of particles in system against the particle movements needed to solve the problem but they also showed the same results as the corresponding section in the Caterpillar-1 algorithm, indicating the no influence of the type of scheduler and the same amount of movements as the number of particle increase.

### 4.2.4 Messages

Table 4.8 presents how the number of messages necessary to solve the Curling Caterpillars problem increases proportionally to the number of particles in system, but also confirmed that there is no influence of the type of scheduler on the number of messages, similar to the Caterpillar-1 algorithm.

| **Table 4.8**: Caterpillar-1M - Number of Messages | | | |
|---|---|---|---|
| **Num. of particles** | **SRR-1** | **SRR-2** | **FSYNC** |
| 10 | 300 | 300 | 300 |
| 20 | 1586 | 1586 | 1586 |
| 30 | 4270 | 4270 | 4270 |
| 40 | 8680 | 8680 | 8680 |
| 50 | 15101 | 15101 | 15101 |
| 100 | 85460 | 85460 | 85460 |
| 200 | 488743 | 488743 | 488743 |
| 300 | 1357532 | 1357532 | 1357532 |
| 400 | 2802532 | 2802532 | 2802532 |
| 500 | 4916707 | 4916707 | 4916707 |
| 1000 | 28148001 | 28148001 | 28148001 |

### 4.2.5 Leader Position

Similar to the corresponding section for the Caterpillar-1 algorithm, experiments for this variable confirmed that the leader position does not influence in any of the factors studied in the previous sections.

## 4.3 Analysis

In general, all experiments showed that the number of messages and movements necessary to solve the problem depend strictly on the number of particles in the system, but they also showed that both algorithms have a better performance using a Fully Synchronous scheduler (FSYNC) regarding both, the simulation time and the number of particle activations; meaning that, the use of such scheduler help to decrease the amount of idle particles in the system,

contrary to the use of a Sequential Random scheduler, which represents the worst-case scenario for the Caterpillar-1 algorithm.

Similarly, experiments made evident that both algorithms are robust to the slight change performed on the Sequential Round-Robin scheduler regarding the simulation time, however, they showed a very slight influence in the number of activation particles necessary to solve the problem.

Table 4.9, Table 4.10 and Table 4.11 compare both algorithms in terms of simulation time, particle activations and messages necessary to form the Compact Ball; revealing that the Caterpillar-1M algorithm performs better than the Caterpillar-1 algorithm regarding all these metrics. Experiments with animations[3] showed this as a consequence of the nature of each algorithm: the Caterpillar-1 algorithm is defined as a sequential algorithm because particles should wait for confirmation of other particles before moving; while particles in the Caterpillar-1M algorithm move independently once they know where to move, explaining at the same time why it does not work with a Sequential Random scheduler.

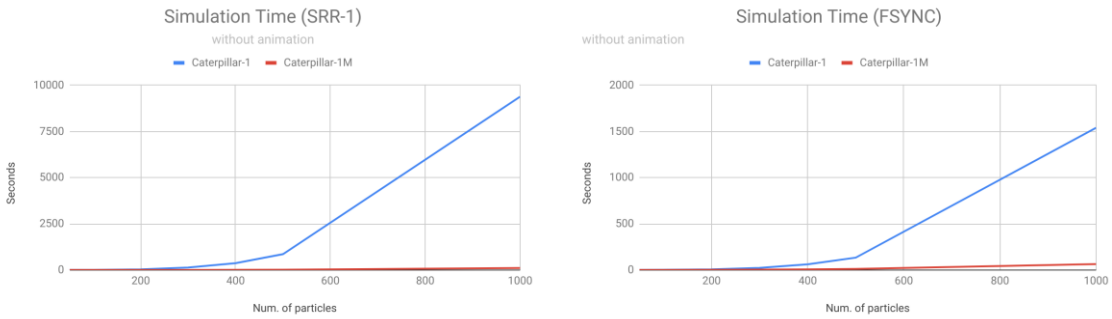
[3] Animations can be found in the following links. Caterpillar-1: http://bit.ly/caterpillar-1-srr-sim (SRR-1), Caterpillar-1M: http://bit.ly/caterpillar-1m-srr-sim (SRR-1)

| **Table 4.9:** Simulation Time (sec) | | | | |
|---|---|---|---|---|
| **Num. of particles** | **SRR-1** | | **FSYNC** | |
| | **Caterpillar-1** | **Caterpillar-1M** | **Caterpillar-1** | **Caterpillar-1M** |
| 10 | 1.02 | 1.02 | 1.01 | 1.02 |
| 20 | 1.03 | 1.03 | 1.03 | 1.02 |
| 30 | 1.07 | 1.04 | 1.05 | 1.03 |
| 40 | 1.15 | 1.05 | 1.08 | 1.04 |
| 50 | 1.3 | 1.1 | 1.11 | 1.07 |
| 100 | 3.98 | 1.34 | 1.58 | 1.2 |
| 200 | 33.63 | 2.76 | 6.77 | 2.07 |
| 300 | 136.64 | 5.96 | 23.52 | 3.96 |
| 400 | 371.89 | 11.32 | 61.59 | 7.18 |
| 500 | 857.36 | 19.12 | 133.95 | 11.79 |
| 1000 | 9376.11 | 110.91 | 1539.27 | 64.21 |

| **Table 4.10:** Number of particle activations | | | | |
|---|---|---|---|---|
| **Num. of particles** | **SRR-1** | | **FSYNC** | |
| | **Caterpillar-1** | **Caterpillar-1M** | **Caterpillar-1** | **Caterpillar-1M** |
| 10 | 1961 | 1081 | 197 | 109 |
| 20 | 25701 | 6801 | 1286 | 341 |
| 30 | 112471 | 19621 | 3750 | 655 |
| 40 | 317721 | 41241 | 7944 | 1032 |
| 50 | 708201 | 73201 | 14165 | 1465 |
| 100 | 8402101 | 429201 | 84022 | 4293 |
| 200 | 98015401 | 2484801 | 490078 | 12425 |
| 300 | 410367601 | 6915301 | 1367893 | 23052 |
| 400 | 1131556401 | 14278001 | 2828892 | 35696 |
| 500 | 2483226001 | 25039501 | 4966453 | 50080 |
| 1000 | 28433203001 | 142973001 | 28433204 | 142974 |

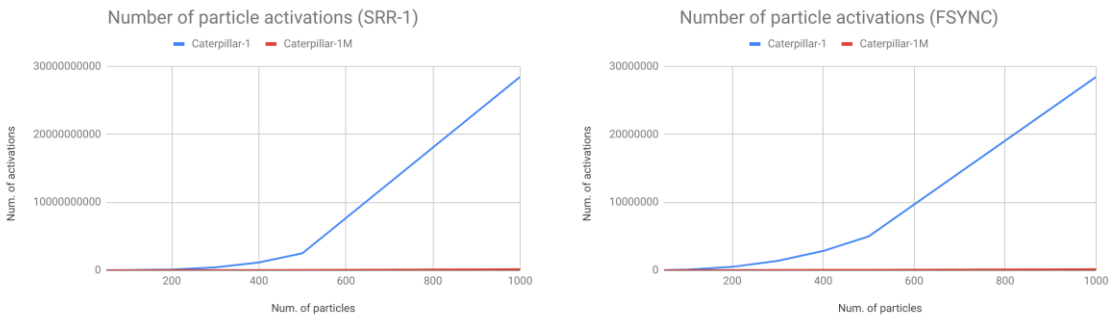
| **Table 4.11**: Number of messages | | |
|---|---|---|
| **Num. of particles** | **Caterpillar-1** | **Caterpillar-1M** |
| 10 | 407 | 300 |
| 20 | 2659 | 1586 |
| 30 | 7702 | 4270 |
| 40 | 16237 | 8680 |
| 50 | 28854 | 15101 |
| 100 | 169777 | 85460 |
| 200 | 985548 | 488743 |
| 300 | 2746086 | 1357532 |
| 400 | 5674001 | 2802532 |
| 500 | 9955911 | 4916707 |
| 1000 | 56933844 | 28148001 |

Figure 4.6, Figure 4.7 and Figure 4.8 show the trends for these results and depict how important are these differences when the number of particle increases considerably, particularly, considering the simulation time and the number of particle activations.




(a) (b)

**Figure 4.6**: Simulation time trends, (a) SRR-1 scheduler, (b) FSYNC scheduler




(a) (b)

**Figure 4.7**: Number of particle activations trends, (a) SRR-1 scheduler, (b) FSYNC scheduler

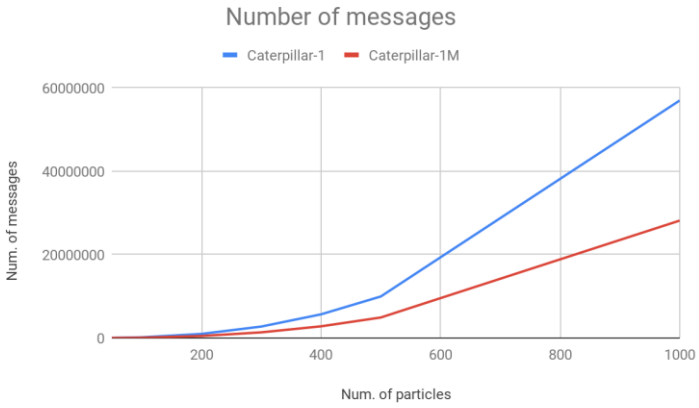



**Figure 4.8**: Number of messages trend

# Chapter 5

## Conclusions

In this thesis, I presented the first version of a simulator designed and implemented specifically for Programmable Particles, called the *ProgPa Simulator*. Its aim is to let researchers test algorithms for programmable particles under different situations and system configurations, finding points of weakness and showing graphically, that everything works as expected.

Given that most researches on programmable particles have been done from a pure theoretical point of view and a thorough search of the relevant literature yielded with no results of a similar tool, this field would greatly benefit from this new simulator known for being multiplatform, easily customizable and available to the public.

Concretely, the work performed on this thesis can be summarized as follow:

- ➔ Research on simulation tools in other fields such as robotics.
- ➔ Design and implementation of a new simulator for programmable particles.
- ➔ Design and implementation of a framework to extend functionalities in this simulator.

- ➔ Implementation and improvement of a new algorithm called *Caterpillar-1*, designed by Flocchini et al. [21] to form a Compact Ball and solve the Curling Caterpillars problem; representing a contribution to the experimental analysis phase of a draft paper.
- ➔ Design and implementation of a new algorithm called *Caterpillar-1M* to form a Compact Ball and solve the Curling Caterpillars problem.
- ➔ Usage of the *ProgPa Simulator* to perform experimental analysis and comparison of two algorithms for programmable particles.

This work made evident how useful is the *ProgPa Simulator* for the creation and validation of new algorithms for programmable particles, but it also confirmed that algorithms for programmable particles where entities have a higher grade of independence provide better results in terms of time, particle activations and communication, such as the case of the studied algorithms: *Caterpillar-1M* and *Caterpillar-1*.

This simulator includes features such as particle movement animations, animation speed configuration, zooming, animation recording and metrics report at the end of each simulation to ease the study and analysis process. Although results from this work were promising, there are still work and improvements that can be done in the future. Some of them are listed as follow:

- ➔ Implementation, integration and analysis of new algorithms. As an example, Flocchini et al. [21] designed other two algorithms called *Caterpillar-2* and

*Caterpillar-3* to solve the Curling Caterpillars problem forming similar shapes as in Figure 5.1.

- ➔ Implementation and integration of new schedulers.
- ➔ GUI improvements on the *ProgPa Simulator* in terms of user interface such as styles, colors and GUI elements, with the purpose of having a better look and feel, and improvement on users understanding of each element and functionality.
- ➔ Improvements in the integration process of new algorithms, making the simulator more scalable.

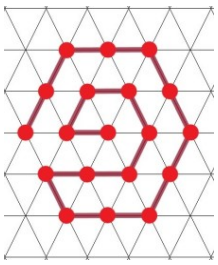


**Figure 5.1**: Caterpillar-2 algorithm shape [21]

Hopefully, the result of this thesis will be the base for future studies in the distributed computing field and new developments for programmable particles.

# References


[1] Aaron Staranowicz and Gian Luca Mariottini. *A Survey and Comparison of Commercial and Open-Source Robotic Simulator Software*. In Proceedings of the 4th International Conference on Pervasive Technologies Related to Assistive Environments, pages 56:1–8, 2011.

[2] Arnaud Casteigts. *JBotSim: a Tool for Fast Prototyping of Distributed Algorithms in Dynamic Networks*. In Proceedings of the 8nd International Conference on Simulation Tools and Techniques, 2015.

[3] *Cocos2d-x Framework*. Available at: https://cocos2d-x.org/. Last accessed: 2019 / 05 / 06.

[4] Chrysovalandis Agathangelou, Chryssis Georgiou, and Marios Mavronicolas. *A distributed algorithm for gathering many fat mobile robots in the plane*. In Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing, pages 250–259. ACM, 2013.

[5] Dan Swain. *glSwarm*. Available at: http://glswarm.sourceforge.net/. Last accessed: 2019 / 05 / 06.

[6] Dana Angluin, James Aspnes, Zoe Diamadi, Michael J. Fischer, and René Peralta. *Computation in networks of passively mobile finite-state sensors*. Distributed Computing, vol. 18, no. 4, pages 235–253, 2006.

[7] David Doty. *Theory of algorithmic self-assembly*. Communications of the ACM, vol. 55, no. 12, pages 78–88, 2012.

[8] Damien Woods. *Intrinsic universality and the computational power of self-assembly*. In Proceedings of the 6th of the Machines, Computations and Universality, pages 16–22, 2013.

[9] Dominique Dhoutaut, Benoit Piranda and Julien Bourgeois. *Efficient simulation of distributed sensing and control environments*. In Proceedings of the 2013 IEEE Int'l Conf. on Internet of Things, pages 452–459, IEEE, 2013.

[10] Giuseppe Antonio Di Luna, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro and Giovanni Viglietta. *A rupestrian algorithm*. 8th International Conference on Fun with Algorithms (FUN), LIPICS Vol. 49, 2016.

[11] Giuseppe Antonio Di Luna, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro and Giovanni Viglietta. *Line-Recovery by Programmable Particles*. In Proceedings of the 19th International Conference on Distributed Computing and Networking, pages 4:1–10, 2017.

[12] Giuseppe Antonio Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta and Yukiko Yamauchi. *Shape Formation by Programmable Particles*. In In Proceedings of the 21st International Conference on Principles of Distributed Systems, pages 31:1–16, 2017.

[13] Gregory Chirikjian. *Kinematics of a metamorphic robotic system*. In Proceedings of the 1994 International Conference on Robotics and Automation, volume 1, pages 449–455, 1994.

[14] Ho-Lin Chen, David Doty, Dhiraj Holden, Chris Thachuk, Damien Woods, and Chun-Tao Yang. *Fast algorithmic self-assembly of simple shapes using random agitation*. In Proceedings of the 20th International Meeting on DNA Computing and Molecular Programming, pages 20–36, 2014.

[15] Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The Amoebot Model. 2017. Available at: https://sops.engineering.asu.edu/sops/. Last accessed: 2019 / 05 / 06.

[16] Ke Li, Kyle Thomas, Claudio Torres, Louis Rossi, and Chien-Chung Shen. *Slime mold inspired path formation protocol for wireless sensor networks*. In Proceedings of the 7th International Conference on Swarm Intelligence, pages 299–311, 2010.

[17] Leonard M. Adleman. *Molecular computation of solutions to combinatorial problems*. Science, vol. 266, no. 11, pages 1021–1024, Nov. 1994.

[18] Matthew J. Patitz. *An introduction to tile-based self-assembly and a survey of recent results*. Natural Computing, vol. 13, no. 2, pages 195–224, 2014.

[19] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. *Programmable self-assembly in a thousand-robot swarm*. Science, vol. 345 no. 6198, pages 795–799, 2014.

[20] Othon Michail and Paul G. Spirakis. *Simple and efficient local codes for distributed stable network construction*. In Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, pages 76–85, 2014.

[21] Paola Flocchini, Aryo Jamshidpey, Nicola Santoro. *Programmable Particles: Curling Caterpillars*. Unpublished document.

[22] Paola Flocchini, David Ilcinkas, Andrzej Pelc, and Nicola Santoro. *Computing without communicating: Ring exploration by asynchronous oblivious robots*. Algorithmica, vol. 65, no. 3, pages 562–583, 2013.

[23] Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *Distributed Computing by Oblivious Mobile Robots*. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool, vol 3, pages 1–185, 2012.

[24] Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. *Arbitrary pattern formation by asynchronous, anonymous, oblivious robots*. Theoretical Computer Science, vol. 407, no. 1, pages 412–447, 2008.

[25] Tommaso Toffoli and Norman Margolus. *Programmable matter: concepts and realization*. Physica D: Nonlinear Phenomena, vol. 47, no. 1, pages 263–272, 1991.

[26] Valerio Volpi. Sycamore 2D / 3D Mobile robots simulation environment. Master thesis, Universitá di Pisa, 2013.

[27] Yaakov Benenson, Tamar Paz-Elizur, Rivka Adar, Ehud Keinan, Zvi Livneh and Ehud Shapiro. *Programmable and autonomous computing machine made of biomolecules*. Nature, vol. 414, no. 6862, pages 430–434, 2001.

[28] Zahra Derakhshandeh. *Algorithmic Foundations of Self-Organizing Programmable Matter*. PhD thesis, Arizona State University, 2017.

[29] Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler and Thim Strothmann. *Universal coating for programmable matter*. Theoretical Computer Science, vol. 671, pages 56-68, 2017.

[30] Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, Thim Strothmann. *Universal shape formation for programmable Matter*. In

Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, pages 289–299, 2016.

[31] Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida Bazzi, Andréa W. Richa, and Christian Scheideler. *Leader election and shape formation with self-organizing programmable matter*. In Proceedings of the 21st International Conference on DNA Computing and Molecular Programming, pages 117–132, 2015.

# Appendix A

## Manuals

This appendix describes important details to install and use the *ProgPa Simulator* in different environments.

### A.1 User Manual

As stated in previous sections, the *ProgPa Simulator* purpose is to let researchers write and test algorithms in the context of Programmable Particles. Three type of users has been identified for the *ProgPa Simulator*, and each of them will configure and use the simulator in different ways.

This manual corresponds to **Basic** users, who will perform experimental analysis on the simulator such as comparing the impact of different settings (e.g, scheduler, leader position, number of particles) in a given algorithm, which is already installed and implemented in the system.

This user does not need any specific knowledge on programming languages, but it needs a very basic knowledge on how to install and run a new software in a computer using a console, for example: the Windows command-line on Windows operating systems, the Terminal Console on Ubuntu or Linux-like operating systems, and the Mac OS X Command Line on Mac operating systems. The

instructions to install and run the simulator are stated in the *README* file located in the source code[4] and they mainly consist of four steps:

1. Follow instructions to install cocos2d-x, making sure to meet and read the prerequisites section.
2. Testing the working environment running the **cpp-tests** project located in the cocos2d-x package.
3. Download the ProgPa Simulator source code and create a symbolic link to the cocos2d-x package within the second progpa-simulator directory, called cocos2d. At the end, the directory structure should be similar to the presented in Figure A1.
4. Run the ProgPa Simulator project using either the cocos2d-x **run** command or an Integrated Development Environment (IDE). It will vary depending on your platform.

```
progpa-simulator/
├── LICENSE
├── README
└── progpa-simulator/
    ├── Classes/
    ├── Resources/
    ├── proj.android/
    ├── proj.ios_mac/
    ├── ...
    └── cocos2d -> your/cocos2d-x/installation
```

**Figure A1**: Directory structure example for the simulator source code

The following subsections presents these installations steps for cocos2d-x v3.17 in two different operating systems: Ubuntu and Windows.

[4] The *ProgPa Simulator* source code is available on: http://bit.ly/progpa-sim

## Ubuntu: v18.04.1 LTS, 64-bit

1. Installing cocos2d-x and prerequisites
    a. Python v2.7 installation. Command:

    ```
    sudo apt-get install python
    ```

    b. CMake v3.10 installation. Command:

    ```
    sudo apt-get install cmake
    ```

    c. Installing other dependencies. Command:

    ```
    sudo apt-get install g++ libgdk-pixbuf2.0-dev libx11-dev libxmu-dev libglu1-mesa-dev libgl2ps-dev libxi-dev libzip-dev libpng-dev libcurl4-gnutls-dev libfontconfig1-dev libsqlite3-dev libglew-dev libssl-dev libgtk-3-dev libglfw3 libglfw3-dev xorg-dev
    ```

    d. Following Linux Installation and Setup steps, downloading and extracting a self-contained .zip from the cocos2d-x website.
    e. Building cocos2d-x. Commands:

    ```
    cd cocos2d-x-root # this is your cocos2d-x location
    cd build
    mkdir linux-build # create directory
    cd linux-build
    cmake ../..
    ```

    f. Compiling cocos2d-x. Command:

```
make -j 4
```

2. Testing the working environment running the **cpp-tests** project located in the cocos2d-x package, especifically, located in the `build/linux-build/bin/Debug/cpp-tests` directory. Commands:

   ```
   cd cocos2d/build/linux-build/bin/Debug/cpp-tests
   ./cpp-tests
   ```

3. Download the ProgPa Simulator source code and create a symbolic link to the cocos2d-x library, called cocos2d. Command:

   ```
   ln -s /your/cocos2d-x/installation/ cocos2d
   ```

4. Run the ProgPa Simulator using the cocos2d-x **run** command:

   ```
   cocos run -s progpa-simulator -p linux
   ```

### Windows: v10, 64-bit

1. Installing cocos2d-x and prerequisites
   a. Python v2.7 installation
      i. Download Python v.2.7 from website
      ii. Execute and follow setup steps. Figure A2 depicts these steps.
      iii. Verify that your `Path` environment variable was updated successfully using the `Control Panel > System and Security > System > Advanced System Settings`. You should see two new values: `C:\Python27\` and

`C:\Python27\Scripts`. Figure A3 shows the expected result.

iv. Finally, open a Command Prompt windows and verify your python installation. Command:

```
python --version
```

b. Visual Studio (community) installation

i. Download VS (community) from website

ii. Execute and follow setup steps. Figure A4 shows these steps.

c. Cocos2d-x v3.17 installation:

i. Download source code from website and extract files

ii. Open a Command Prompt windows and go the the extracted files

iii. Run the `setup.py` file, skipping the `NDK_ROOT` and `ANDROID_SDK_ROOT` questions pressing enter

iv. Restart your terminal or computer to make added system variables take effect. You will see three new environment variables for your user: `COCOS_CONSOLE_ROOT`, `COCOS_X_ROOT` and `COCOS_TEMPLATE_ROOT`. Figure A5 shows these results.

2. Testing the working environment running the **cpp-tests** project located in the cocos2d-x library:

a. Go to the `build` directory and open the `cocos2d-win32.sln` file with Visual Studio

b. You might need to re-target projects depending on the Windows SDK versions installed in your system
c. Build and run the **cpp-tests** project

3. Download and configure the ProgPa Simulator source code
    a. Open a Command Prompt windows
    b. Go to the progpa-simulator directory (same level as Classes and Resources)
    c. Create a symbolic link called `cocos2d` to your cocos2d-x installation with the following command:

        ```
        mklink cocos2d your\cocos2d-x-3.17\installation
        ```

4. Run the ProgPa Simulator
    a. Go to the `progpa-simulator\proj.win32` directory
    b. Open the `progpa-simulator.sln` file using Visual Studio
    c. Install the missing tools to your environment if necessary
    d. Build and run the ProgPa Simulator project

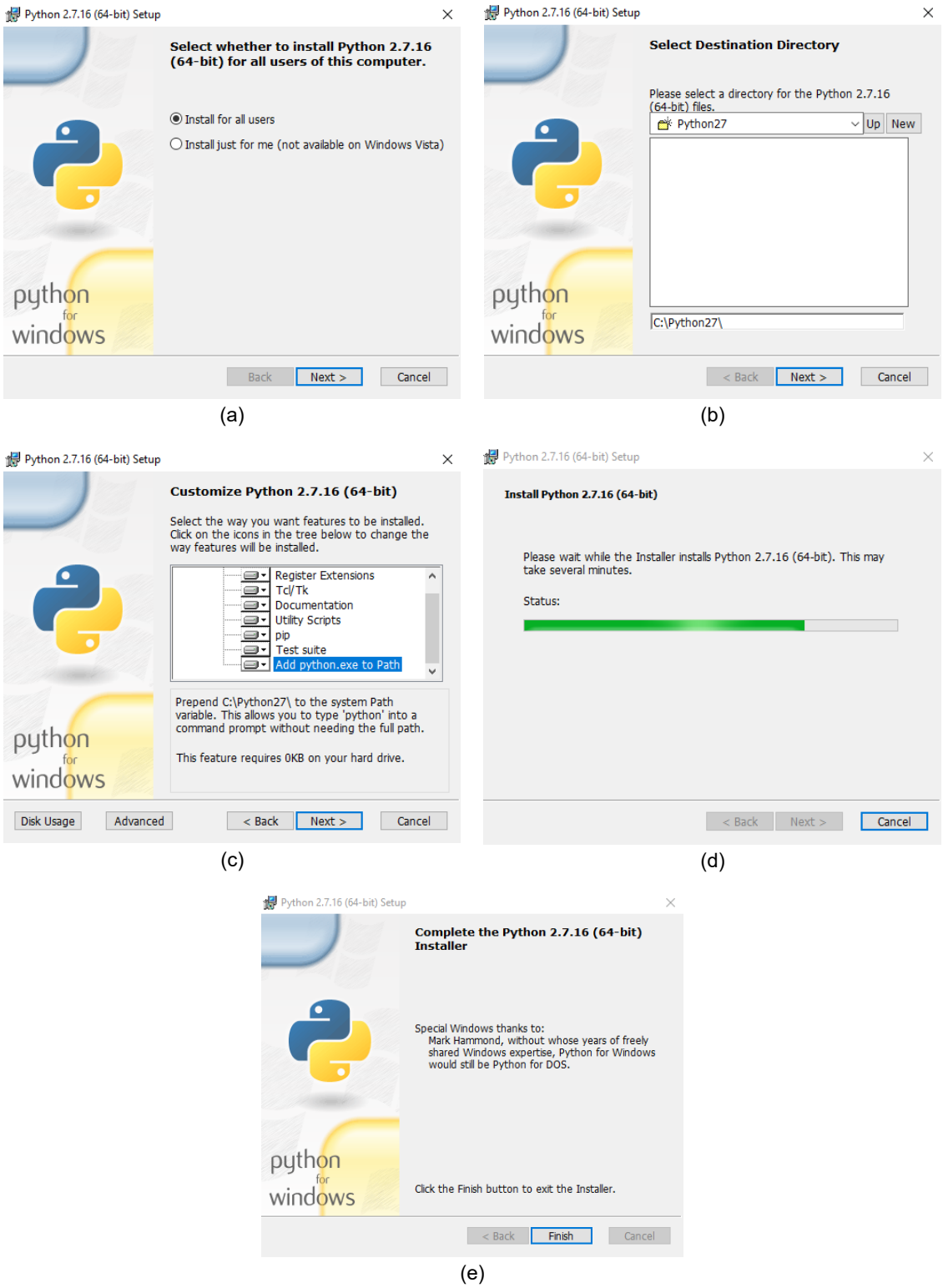



**Figure A2**: Windows - Python setup steps

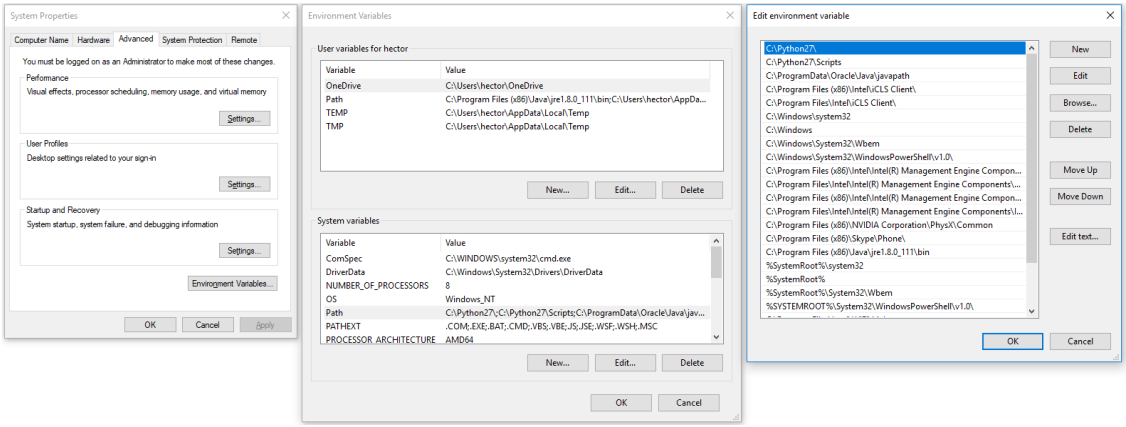
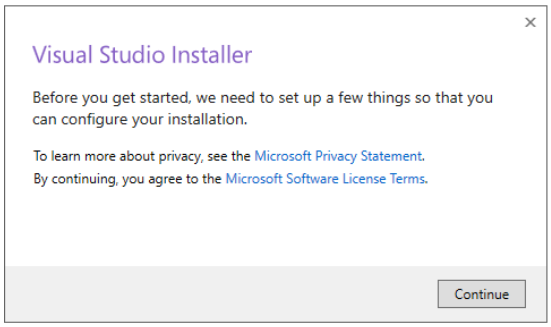
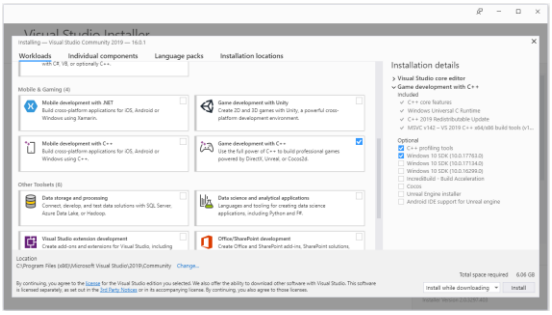


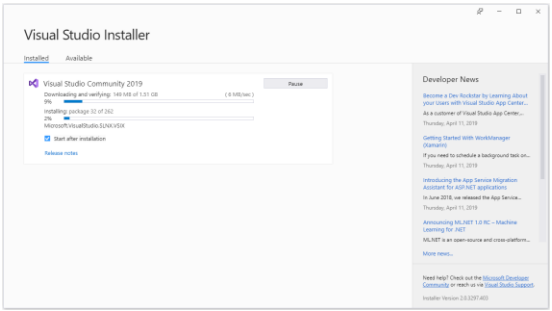

**Figure A3**: Windows - Python `Path` environment variable



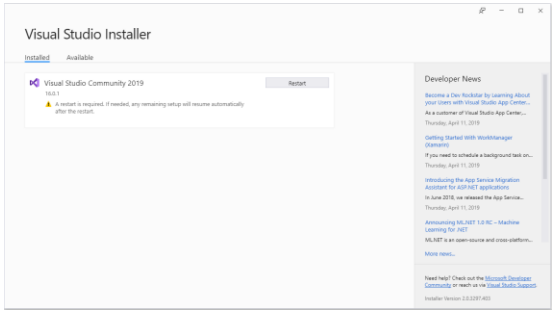

(a)



(b)




(c)




(d)

**Figure A4**: Windows - Visual Studio setup steps

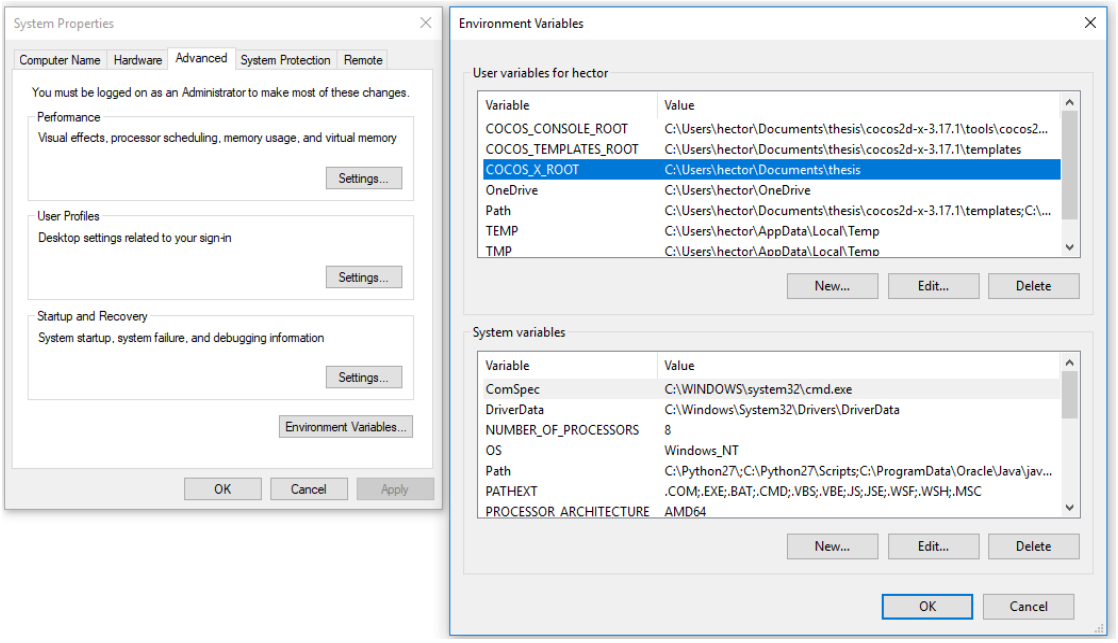



**Figure A5**: Windows - cocos2d-x environment variables

Once the simulator is properly installed, the user needs to interact with the GUI elements provided by the simulator: windows, buttons, text inputs, checkboxes, radio buttons and sliders. The three interacting steps were presented in Figure A6.

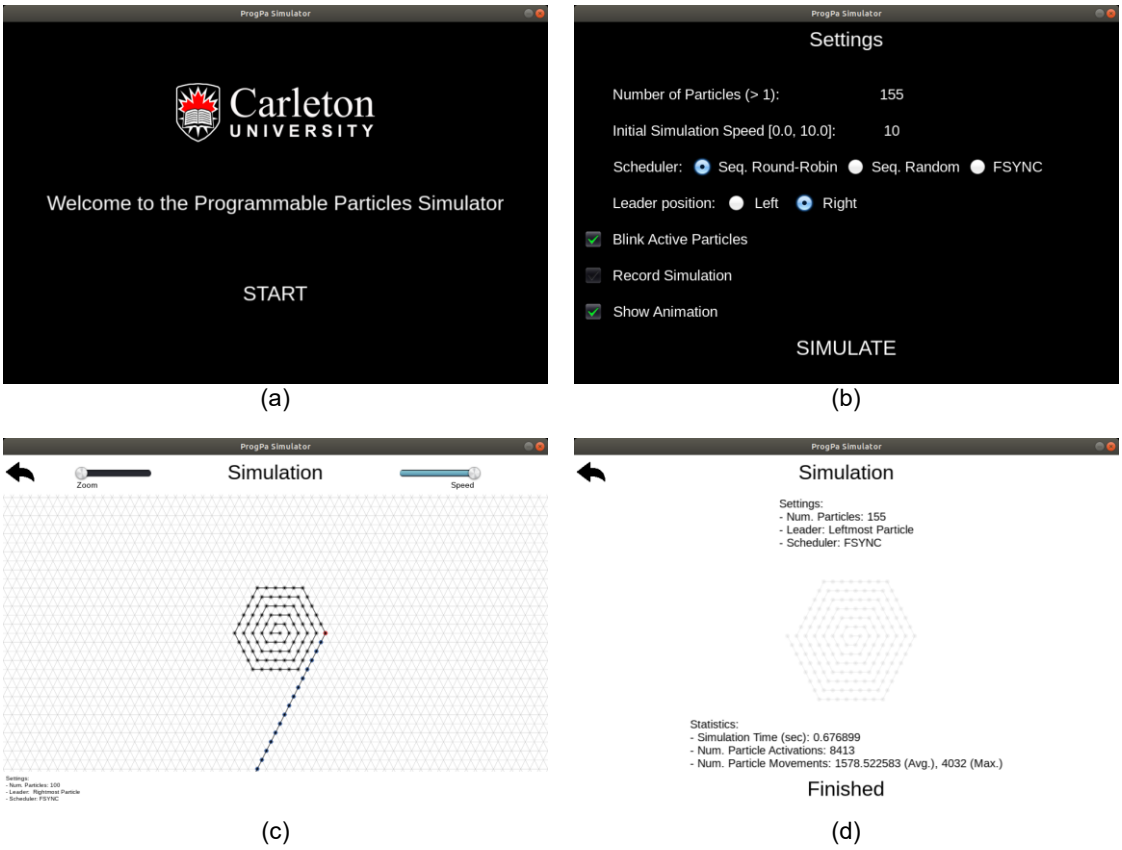



**Figure A6**: (a) Welcome Scene, (b) Configuration Scene, (c) Simulate Scene, (d) Statistic Scene

## A.2 System Manual

This manual corresponds to the remaining type of users: **Algorithm Developer** and **Simulator Developer** users, who are in charge of implementing and installing new algorithms in the system and implementing and installing new features in the simulator (e.g., new GUI elements, new interface, new schedulers, new particle representations, new animations, new maps), respectively.

Necessarily, these kind of users need programming skills, specifically, in the C++ programming language but also, the Simulator Developer users need a basic

knowledge of the Cocos2d framework [3], concretely, on the Cocos2d-x branch. Keeping this in mind, the following subsections provide guidelines for each type of user.

### Algorithm Developer User

This work also provides a framework to guide the process of implementing, installing and using a new algorithm. This process consists of four main steps:

1. Installing and running the simulator as it is described in the User Manual (or *README* file)
2. Defining the new algorithm interface in C++: this new interface should be written in a C++ header file (`.h`) with any name (e.g., `MyAlgorithm.h`), the interface should be named `Particle` and it should be derived from the `BaseParticle` class; defining two `Particle(...)` constructors, two mandatory methods: `isTerminated(...)` and `run(...)`; and finally, as many helper methods, state methods and internal attributes as necessary. Figure A7 shows a template for this interface.
3. Implementing the new algorithm in C++: this code should be written in a C++ code file (`.cpp`) with the same name provided before (e.g., `MyAlgorithm.cpp`), it should use the interface `Particle` defined in the previous step and it should implement every method defined in this interface. It could use any method or attribute inherited from the `BaseParticle` implementation. Figure A8 shows a template for this file.

4. Integrate, compile and use the new algorithm: this requires the modification of two existing files: the simulation scene header file (`BaseSimulationScene.h`), located in the `Classes` directory; and the list of files to compile and use (`CMakeLists.txt`), located in the second `progpa-simulator` directory.

   The first file requires to replace the name of the algorithm used during the simulation, the corresponding lines are the first 7 lines, where it is necessary, for example, to replace `#include "models/Caterpillar1Particle.h"` by `#include "models/MyAlgorithm.h"`.

   Similarly, the second file requires to replace the name of the files to use during the compilation process. The corresponding lines are around line 68 and line 82, where it is necessary two replacements: `Classes/models/Caterpillar1Particle.h` by `Classes/models/MyAlgorithm.h` and `Classes/models/Caterpillar1Particle.cpp` by `Classes/models/MyAlgorithm.cpp`.

Once these steps are completed, the simulator should be ready to run and use the new algorithm, for example, with the following on linux:

```
cocos run -s progpa-simulator -p linux
```

**Figure A7**: `TemplateParticle.h` - a template[5] for a new algorithm interfaces

```
#pragma once

#include "BaseParticle.h"
#include "Helpers.h"            /* if needed */

/*
 * Sprite / Image definitions for each state
 *
 * Example:
 *
 * static const std::string FOLLOWER_PARTICLE_IMG = "blue_particle.png";
 */


// Add your code here


/*
 * Definition of particle states
 *
 * Example:
 *
 * static const short int FOLLOWER_STATE = 0;
 */


// Add your code here


/*
 * Definition of messages between particles
 *
 * Example:
 *
 * static const short int ROOT_ACTIVATION_MSG = 10;
 */


// Add your code here
```

[5] `TemplateParticle.h` is available in the source code

```
/*
 * Particle Class definition
 */


class Particle : public BaseParticle {
 public:
  /* Constructors */
  Particle();
  Particle(float x, float y, bool isLeader, bool leaderIsRightmost);

  /* Any specific helper functions needed */
  bool isTerminated() const;    /* Mandatory, used by the Simulation Scene */
  int myHelper() const;         /* Optional */

  /*
   * Algorithm methods to be executed by particles according to its state.
   */

  /* Mandatory, used by the Simulation Scene */
  void run(Particle * next, Particle * previous,
           std::map<std::string, bool> * neighborhood,
           unsigned int * messages);

  /* Optional */
  void yourState1(Particle * next, Particle * previous,
                  std::map<std::string, bool> * neighborhood,
                  unsigned int * messages);

  /* Optional */
  void yourState2(Particle * next, Particle * previous,
                  std::map<std::string, bool> * neighborhood,
                  unsigned int * messages);
  /*
   * Optional.
   *
   * Specific attributes used by your algorithm for communication or
   * working properly.
   *
   * Example:
   *
   * - Shared / Private memories
   * - Internal variables
   */
};
```

**Figure A8**: `TemplateParticle.cpp` - a template[6] for a new algorithm implementation

```cpp
#include "TemplateParticle.h"


/* ------------
 * CONSTRUCTORS
 * ------------
 */

Particle::Particle () {}

Particle::Particle (float x, float y, bool isLeader, bool leaderIsRightmost) {
  _leaderIsRightmost = leaderIsRightmost;

  if (isLeader) {
    _sprite = cocos2d::Sprite::create(state1Img);
    _state = state1;
  } else {
    _sprite = cocos2d::Sprite::create(state2Img);
    _state = state2;
  }

  _sprite->setAnchorPoint(cocos2d::Vec2(0.5, 0.5));
  _sprite->setPosition(cocos2d::Vec2(x, y));
}

/* ----------------
 * HELPER FUNCTIONS
 * ----------------
 */


// Add your code here
```

[6] `TemplateParticle.cpp` is available in the source code

```
/* -----------------
 * ALGORITHM METHODS
 * -----------------
 */

void Particle::state1 (Particle * next, Particle * previous,
                       std::map<std::string, bool> * neighborhood,
                       unsigned int * messages) {
  /*
   * Algorithm for particles in the 'state1' state
   *
   */

  // Add your code here
}


// Add more states implementation here


void Particle::run (Particle * next, Particle * previous,
                    std::map<std::string, bool> * neighborhood,
                    unsigned int * messages) {
  if (_state == state1) {
    state1(next, previous, neighborhood, messages);
  } else if (_state == state2) {
    state2(next, previous, neighborhood, messages);
  } else {
    CCLOG("Invalid state...");
  }
}
```

## Simulator Developer User

This subsection describes the process of implementing and installing new features on the simulator, including but not limited to: a new GUI elements, new interface, new schedulers, new particle representations, new animations and new graphs (map).

Similarly to the previous case, this work also provides a framework to implement these kind of changes. For example, implementing a new scheduler requires the following four steps:

1. Installing and running the simulator as it is described in the User Manual (or *README* file)
2. Defining the new scheduler interface in C++: this new interface should be written in the `Schedulers.h` file within the `models` directory. It could have any name (e.g., `MyScheduler`) and it should be derived from the `BaseScheduler` class; defining a constructor: `MyScheduler(...)`, a mandatory method: `getParticleIndexes(...)`; and finally, as many helper methods and internal attributes as necessary. Figure A9 shows an example for this interface.

   Additionally, it is necessary to assign an identifier variable for this new scheduler at the top of the `Schedulers.h` file.
3. Implementing the new scheduler in C++: this code should be written in the `Schedulers.cpp` file within the `models` directory. It should use the interface defined in the previous step (e.g., `MyScheduler`) and it should implement every method defined in this interface. It could use any method or attribute inherited from the `BaseScheduler` implementation. Figure A10 shows an example of such implementations.
4. Integrate and use the new scheduler: this requires the modification of two existing files: the configuration scene implementation file

(`ConfigurationScene.cpp`) and the simulation scene implementation file (`SimulationScene.cpp`); both located in the `Classes` directory.

While the first file requires to update the `_addSchedulerUI` method to show the user a new scheduler option through the GUI, using a new radio button element; the second file requires to update the code portion that parses the scheduler identifiers and creates the corresponding instance in the `init` method, but also requires to update the `_addSettingsUI` method to consider and print the new scheduler name on the settings section in the screen.

In the same way, any change in one of the three available scenes should be done in the corresponding `.cpp` file. For example, a change in the welcome scene should be performed in the `WelcomeScene.cpp` file. However, changes corresponding to the images in the simulator, for example, the *Carleton University* logo should be done in the `Resources` directory and the scene in which it appears (e.g., `WelcomeScene.cpp`).

**Figure A9**: Example of a Scheduler Interface definition

```
/* ---------------------------
 * Sequential Random Scheduler
 * ---------------------------
 */

class SequentialRandomScheduler : public BaseScheduler {
public:
  /* Constructors */
  SequentialRandomScheduler(int numParticles);

  /* Getters & Setters */
  std::vector<int> getParticleIndexes(unsigned long long int tickRound);
private:
  std::vector<int> indexes;
};
```

**Figure A10**: Example of a Scheduler Interface implementation

```
/* ---------------------------
 * Sequential Random Scheduler
 * ---------------------------
 */

SequentialRandomScheduler::SequentialRandomScheduler (int numParticles) : BaseScheduler(numParticles) {
  for (int i = 0; i < numParticles; i++) {
    this->indexes.push_back(i);
  }
}

std::vector<int> SequentialRandomScheduler::getParticleIndexes (unsigned long long int tickRound) {
  std::vector<int> result;
  int currentIndex = tickRound % this->numParticles;

  if (currentIndex == 0) {
    std::random_shuffle(this->indexes.begin(), this->indexes.end());
  }

  result.push_back(this->indexes[currentIndex]);

  return result;
}
```

# Appendix B

## Experimental Results

This appendix shows detailed results from the experimental phase. They are mainly divided by algorithm: Caterpillar-1 and Caterpillar-1M, and then by scheduler Sequential Round-Robin (SRR-1), Sequential Round-Robin variant (SRR-2), Sequential Random (SR) and Fully Synchronous (FSYNC).

### B.1 Caterpillar-1

Tables from B1 to B10 resume the influence of the number of particles in the system against the simulation time, the number of particle activations, the average, maximum and total number of movements between all particles and finally, the maximum and the total number of messages necessary to solve the problem.

| **Table B1**: Caterpillar-1 - SRR-1 Scheduler | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Num. Particles** | **Simulation Time** | **Num. Particle Activations** | **Avg. Num. Movements** | **Max. Num. Movements** | **Total Movements** | **Max. Num. Messages** | **Total Messages** |
| 10 | 1.02 | 1961 | 15 | 41 | 150 | 84 | 407 |
| 20 | 1.03 | 25701 | 54.7 | 145 | 1094 | 292 | 2659 |
| 30 | 1.07 | 112471 | 110.43 | 290 | 3313 | 582 | 7702 |
| 40 | 1.15 | 317721 | 179.05 | 467 | 7162 | 936 | 16237 |
| 50 | 1.3 | 708201 | 258.76 | 672 | 12938 | 1346 | 28854 |
| 100 | 3.98 | 8402101 | 790.7 | 2031 | 79070 | 4064 | 169777 |
| 200 | 33.63 | 98015401 | 2350.87 | 5990 | 470176 | 11982 | 985548 |
| 300 | 136.64 | 410367601 | 4410.13 | 11198 | 1323041 | 22398 | 2746086 |
| 400 | 371.89 | 1131556401 | 6872.72 | 17415 | 2749090 | 34832 | 5674001 |
| 500 | 857.36 | 2483226001 | 9683.4 | 24503 | 4841701 | 49008 | 9955911 |
| 1000 | 9376.11 | 28433203001 | 27933.7 | 70434 | 27933702 | 140870 | 56933844 |

| **Table B2**: Caterpillar-1 - SRR-2 Scheduler | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Num. Particles** | **Simulation Time** | **Num. Particle Activations** | **Avg. Num. Movements** | **Max. Num. Movements** | **Total Movements** | **Max. Num. Messages** | **Total Messages** |
| 10 | 1.02 | 1970 | 15 | 41 | 150 | 84 | 407 |
| 20 | 1.04 | 25720 | 54.7 | 145 | 1094 | 292 | 2659 |
| 30 | 1.06 | 112500 | 110.43 | 290 | 3313 | 582 | 7702 |
| 40 | 1.16 | 317760 | 179.05 | 467 | 7162 | 936 | 16237 |
| 50 | 1.28 | 708250 | 258.76 | 672 | 12938 | 1346 | 28854 |
| 100 | 4.01 | 8402200 | 790.7 | 2031 | 79070 | 4064 | 169777 |
| 200 | 33.65 | 98015600 | 2350.87 | 5990 | 470176 | 11982 | 985548 |
| 300 | 136.62 | 410367900 | 4410.13 | 11198 | 1323041 | 22398 | 2746086 |
| 400 | 371.87 | 1131556800 | 6872.72 | 17415 | 2749090 | 34832 | 5674001 |
| 500 | 857.39 | 2483226500 | 9683.4 | 24503 | 4841701 | 49008 | 9955911 |
| 1000 | 9376.12 | 28433204000 | 27933.7 | 70434 | 27933702 | 140870 | 56933844 |

| **Table B3**: Caterpillar-1 - SR Scheduler | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Num. Particles** | **Simulation Time** | **Num. Particle Activations** | **Avg. Num. Movements** | **Max. Num. Movements** | **Total Movements** | **Max. Num. Messages** | **Total Messages** |
| 10 | 1.02 | 2338 | 15 | 41 | 150 | 84 | 407 |
| 20 | 1.06 | 30366 | 54.7 | 145 | 1094 | 292 | 2659 |
| 30 | 1.1 | 132388 | 110.43 | 290 | 3313 | 582 | 7702 |
| 40 | 1.2 | 370648 | 179.05 | 467 | 7162 | 936 | 16237 |
| 50 | 1.38 | 833342 | 258.76 | 672 | 12938 | 1346 | 28854 |
| 100 | 4.84 | 9800947 | 790.7 | 2031 | 79070 | 4064 | 169777 |
| 200 | 44.54 | 114291256 | 2350.87 | 5990 | 470176 | 11982 | 985548 |
| 300 | 183.32 | 478839267 | 4410.13 | 11198 | 1323041 | 22398 | 2746086 |
| 400 | 494.66 | 1318298697 | 6872.72 | 17415 | 2749090 | 34832 | 5674001 |
| 500 | 1100.85 | 2892253370 | 9683.4 | 24503 | 4841701 | 49008 | 9955911 |
| 1000 | 12872.5 | 33112003802 | 27933.7 | 70434 | 27933702 | 140870 | 56933844 |

| **Table B4**: Caterpillar-1 - FSYNC Scheduler | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Num. Particles** | **Simulation Time** | **Num. Particle Activations** | **Avg. Num. Movements** | **Max. Num. Movements** | **Total Movements** | **Max. Num. Messages** | **Total Messages** |
| 10 | 1.01 | 197 | 15 | 41 | 150 | 84 | 407 |
| 20 | 1.03 | 1286 | 54.7 | 145 | 1094 | 292 | 2659 |
| 30 | 1.05 | 3750 | 110.43 | 290 | 3313 | 582 | 7702 |
| 40 | 1.08 | 7944 | 179.05 | 467 | 7162 | 936 | 16237 |
| 50 | 1.11 | 14165 | 258.76 | 672 | 12938 | 1346 | 28854 |
| 100 | 1.58 | 84022 | 790.7 | 2031 | 79070 | 4064 | 169777 |
| 200 | 6.77 | 490078 | 2350.87 | 5990 | 470176 | 11982 | 985548 |
| 300 | 23.52 | 1367893 | 4410.13 | 11198 | 1323041 | 22398 | 2746086 |
| 400 | 61.59 | 2828892 | 6872.72 | 17415 | 2749090 | 34832 | 5674001 |
| 500 | 133.95 | 4966453 | 9683.4 | 24503 | 4841701 | 49008 | 9955911 |
| 1000 | 1539.27 | 28433204 | 27933.7 | 70434 | 27933702 | 140870 | 56933844 |

| **Table B5**: Caterpillar-1 - Simulation Time (animation). SRR-1 Scheduler | | | |
|---|---|---|---|
| **Num. Particles** | **1X** | **5X** | **10X** |
| 10 | 133.36 | 26.68 | 13.34 |
| 20 | 1747.73 | 349.55 | 174.79 |
| 30 | 7648.27 | 1529.66 | 764.84 |
| 40 | 22546.59 | 4519.32 | 2160.59 |
| 50 | 48159.13 | 9631.83 | 7437.12 |
| 100 | 571359.86 | 114272.02 | 57136.02 |

| **Table B6**: Caterpillar-1 - Simulation Time (animation). SR Scheduler | | | |
|---|---|---|---|
| **Num. Particles** | **1X** | **5X** | **10X** |
| 10 | 165.46 | 33.53 | 16.77 |
| 20 | 2084.82 | 421.04 | 210.53 |
| 30 | 8988.68 | 1805.25 | 902.64 |
| 40 | 25282.29 | 5049.55 | 2546.84 |
| 50 | 56187.1 | 11236.12 | 5627.31 |
| 100 | 667317.77 | 133463.61 | 66785.71 |

| **Table B7**: Caterpillar-1 - Simulation Time (animation). FSYNC Scheduler | | | |
|---|---|---|---|
| **Num. Particles** | **1X** | **5X** | **10X** |
| 10 | 13.41 | 2.69 | 1.35 |
| 20 | 87.45 | 17.51 | 8.76 |
| 30 | 255.02 | 51.01 | 25.52 |
| 40 | 540.22 | 108.06 | 54.04 |
| 50 | 963.26 | 192.66 | 96.33 |
| 100 | 5713.79 | 1142.76 | 577.28 |

| **Table B8**: Caterpillar-1 - Simulation Time Relation (animation). SRR-1 Scheduler | | |
|---|---|---|
| **1X / 5X** | **1X / 10X** | **5X / 10X** |
| 4.99850075 | 9.997001499 | 2.000000000 |
| 4.999942784 | 9.999027404 | 1.999828365 |
| 4.999980388 | 9.99983003 | 1.999973851 |
| 4.988934176 | 10.4353857 | 2.091706432 |
| 4.999997924 | 6.475507992 | 1.295102136 |
| 4.9999979 | 9.999994049 | 1.99999965 |

| **Table B9**: Caterpillar-1 - Simulation Time Relation (animation). SR Scheduler | | |
|---|---|---|
| **1X / 5X** | **1X / 10X** | **5X / 10X** |
| 4.934685356 | 9.866428145 | 1.999403697 |
| 4.951596048 | 9.902721702 | 1.999905002 |
| 4.979188478 | 9.958211469 | 1.999966764 |
| 5.006840213 | 9.926925131 | 1.982672645 |
| 5.000578492 | 9.984717387 | 1.996712461 |
| 4.999997902 | 9.99192447 | 1.998385733 |

| **Table B10**: Caterpillar-1 - Simulation Time Relation (animation). FSYNC Scheduler | | |
|---|---|---|
| **1X / 5X** | **1X / 10X** | **5X / 10X** |
| 4.985130112 | 9.933333333 | 1.992592593 |
| 4.994288978 | 9.982876712 | 1.998858447 |
| 4.99941188 | 9.992946708 | 1.998824451 |
| 4.999259671 | 9.996669134 | 1.999629904 |
| 4.99979238 | 9.999584761 | 2.000000000 |
| 4.999991249 | 9.897779241 | 1.979559313 |

## B.2 Caterpillar-1M

Similarly, tables from B11 to B17 resume the result obtained for the new proposed algorithm *Caterpillar-1M*, using a similar format than the previous section but excluding results for the SR Scheduler, given that this algorithm does not work with it.

**Table B11**: Caterpillar-1M - SRR-1 Scheduler

| Num. Particles | Simulation Time | Num. Particle Activations | Avg. Num. Movements | Max. Num. Movements | Total Movements | Max. Num. Messages | Total Messages |
|---|---|---|---|---|---|---|---|
| 10 | 1.02 | 1081 | 15 | 41 | 150 | 100 | 300 |
| 20 | 1.03 | 6801 | 54.7 | 145 | 1094 | 322 | 1586 |
| 30 | 1.04 | 19621 | 110.43 | 290 | 3313 | 626 | 4270 |
| 40 | 1.05 | 41241 | 179.05 | 467 | 7162 | 993 | 8680 |
| 50 | 1.1 | 73201 | 258.76 | 672 | 12938 | 1416 | 15101 |
| 100 | 1.34 | 429201 | 790.7 | 2031 | 79070 | 4194 | 85460 |
| 200 | 2.76 | 2484801 | 2350.87 | 5990 | 470176 | 12226 | 488743 |
| 300 | 5.96 | 6915301 | 4410.13 | 11198 | 1323041 | 22753 | 1357532 |
| 400 | 11.32 | 14278001 | 6872.72 | 17415 | 2749090 | 35297 | 2802532 |
| 500 | 19.12 | 25039501 | 9683.4 | 24503 | 4841701 | 49581 | 4916707 |
| 1000 | 110.91 | 142973001 | 27933.7 | 70434 | 27933702 | 141975 | 28148001 |
| 5000 | 6243.48 | 8088960001 | 320284.09 | 803776 | 1601420546 | 1612794 | 1603846871 |

| **Table B12**: Caterpillar-1M - SRR-2 Scheduler | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Num. Particles** | **Simulation Time** | **Num. Particle Activations** | **Avg. Num. Movements** | **Max. Num. Movements** | **Total Movements** | **Max. Num. Messages** | **Total Messages** |
| 10 | 1.02 | 1090 | 15 | 41 | 150 | 100 | 300 |
| 20 | 1.02 | 6820 | 54.7 | 145 | 1094 | 322 | 1586 |
| 30 | 1.04 | 19650 | 110.43 | 290 | 3313 | 626 | 4270 |
| 40 | 1.05 | 41280 | 179.05 | 467 | 7162 | 993 | 8680 |
| 50 | 1.09 | 73250 | 258.76 | 672 | 12938 | 1416 | 15101 |
| 100 | 1.35 | 429300 | 790.7 | 2031 | 79070 | 4194 | 85460 |
| 200 | 2.77 | 2485000 | 2350.87 | 5990 | 470176 | 12226 | 488743 |
| 300 | 5.94 | 6915600 | 4410.13 | 11198 | 1323041 | 22753 | 1357532 |
| 400 | 11.3 | 14278400 | 6872.72 | 17415 | 2749090 | 35297 | 2802532 |
| 500 | 19.13 | 25040000 | 9683.4 | 24503 | 4841701 | 49581 | 4916707 |
| 1000 | 110.94 | 142974000 | 27933.7 | 70434 | 27933702 | 141975 | 28148001 |

| **Table B13**: Caterpillar-1M - FSYNC Scheduler | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Num. Particles** | **Simulation Time** | **Num. Particle Activations** | **Avg. Num. Movements** | **Max. Num. Movements** | **Total Movements** | **Max. Num. Messages** | **Total Messages** |
| 10 | 1.02 | 109 | 15 | 41 | 150 | 100 | 300 |
| 20 | 1.02 | 341 | 54.7 | 145 | 1094 | 322 | 1586 |
| 30 | 1.03 | 655 | 110.43 | 290 | 3313 | 626 | 4270 |
| 40 | 1.04 | 1032 | 179.05 | 467 | 7162 | 993 | 8680 |
| 50 | 1.07 | 1465 | 258.76 | 672 | 12938 | 1416 | 15101 |
| 100 | 1.2 | 4293 | 790.7 | 2031 | 79070 | 4194 | 85460 |
| 200 | 2.07 | 12425 | 2350.87 | 5990 | 470176 | 12226 | 488743 |
| 300 | 3.96 | 23052 | 4410.13 | 11198 | 1323041 | 22753 | 1357532 |
| 400 | 7.18 | 35696 | 6872.72 | 17415 | 2749090 | 35297 | 2802532 |
| 500 | 11.79 | 50080 | 9683.4 | 24503 | 4841701 | 49581 | 4916707 |
| 1000 | 64.21 | 142974 | 27933.7 | 70434 | 27933702 | 141975 | 28148001 |
| 5000 | 4416.79 | 1617793 | 320284.09 | 803776 | 1601420546 | 1612794 | 1603846871 |

| **Table B14**: Caterpillar-1 - Simulation Time (animation). SRR-1 Scheduler | | | |
|---|---|---|---|
| **Num. Particles** | **1X** | **5X** | **10X** |
| 10 | 73.59 | 14.71 | 7.36 |
| 20 | 462.59 | 92.51 | 46.26 |
| 30 | 1334.29 | 266.85 | 133.43 |
| 40 | 2804.59 | 560.91 | 280.46 |
| 50 | 4977.99 | 995.59 | 497.8 |
| 100 | 29186.68 | 5837.33 | 2918.67 |
| 200 | 168971.59 | 33794.33 | 16897.17 |
| 300 | 490352.56 | 98070.55 | 49036.09 |

| **Table B15**: Caterpillar-1 - Simulation Time (animation). FSYNC Scheduler | | | |
|---|---|---|---|
| **Num. Particles** | **1X** | **5X** | **10X** |
| 10 | 7.49 | 1.49 | 0.75 |
| 20 | 23.39 | 4.67 | 2.34 |
| 30 | 44.59 | 8.91 | 4.46 |
| 40 | 70.39 | 14.07 | 7.04 |
| 50 | 99.79 | 19.95 | 9.98 |
| 100 | 291.99 | 58.39 | 29.2 |
| 200 | 1994.09 | 398.81 | 199.41 |
| 300 | 6137.69 | 1227.53 | 613.77 |

| **Table B16**: Caterpillar-1 - Simulation Time relation (animation). SRR-1 Scheduler | | |
|---|---|---|
| **1X / 5X** | **1X / 10X** | **5X / 10X** |
| 5.002719239 | 9.998641304 | 1.998641304 |
| 5.000432386 | 9.999783831 | 1.999783831 |
| 5.000149897 | 9.999925054 | 1.999925054 |
| 5.000071313 | 9.999964344 | 1.999964344 |
| 5.000040177 | 9.999979912 | 1.999979912 |
| 5.000005139 | 9.999993148 | 1.999996574 |
| 4.999998225 | 9.99999349 | 1.999999408 |
| 4.999998063 | 9.999829921 | 1.999966759 |

| **Table B17**: Caterpillar-1 - Simulation Time relation (animation). SR Scheduler | | |
|---|---|---|
| **1X / 5X** | **1X / 10X** | **5X / 10X** |
| 5.026845638 | 9.986666667 | 1.986666667 |
| 5.00856531 | 9.995726496 | 1.995726496 |
| 5.004489338 | 9.997757848 | 1.997757848 |
| 5.002842928 | 9.998579545 | 1.998579545 |
| 5.002005013 | 9.998997996 | 1.998997996 |
| 5.000685049 | 9.999657534 | 1.999657534 |
| 5.000100298 | 9.999949852 | 1.999949852 |
| 5.000032586 | 9.999983707 | 1.999983707 |