2011 IEEE/RSJ International Conference on
Intelligent Robots and Systems
September 25-30, 2011. San Francisco, CA, USA

# ARGoS: a Modular, Multi-Engine Simulator for Heterogeneous Swarm Robotics

Carlo Pinciroli[†], Vito Trianni[†], Rehan O'Grady[†], Giovanni Pini[†], Arne Brutschy[†], Manuele Brambilla[†], Nithin Mathews[†], Eliseo Ferrante[†], Gianni Di Caro[‡], Frederick Ducatelle[‡], Timothy Stirling[§], Álvaro Gutiérrez[*], Luca Maria Gambardella[‡] and Marco Dorigo[†]

***Abstract*— We present ARGoS, a novel open source multi-robot simulator. The main design focus of ARGoS is the real-time simulation of large heterogeneous swarms of robots. Existing robot simulators obtain scalability by imposing limitations on their extensibility and on the accuracy of the robot models. By contrast, in ARGoS we pursue a deeply modular approach that allows the user both to easily add custom features and to allocate computational resources where needed by the experiment. A unique feature of ARGoS is the possibility to use multiple physics engines of different types and to assign them to different parts of the environment. Robots can migrate from one engine to another transparently. This feature enables entirely novel classes of optimizations to improve scalability and paves the way for a new approach to parallelism in robotics simulation. Results show that ARGoS can simulate about 10,000 simple wheeled robots 40% faster than real-time.**

## I. Introduction

In this paper we present ARGoS, a novel open source multi-robot simulator. ARGoS was developed within the EU-funded Swarmanoid project[1], which was dedicated to the study of tools and control strategies for heterogeneous swarms of robots. Simulation is central to the study of swarm robotics for several reasons. In general, simulation allows for cheaper and faster collection of experimental data, without the risk of damaging the (often expensive) real hardware platforms. In addition, simulated experiments can potentially involve quantity of robots that would be impossible to manufacture for reasons of cost. In the quest for an effective simulation tool for the Swarmanoid robots, we identified two critical requirements: *extensibility* (to support highly diverse robots) and *scalability* (to support a high number of robots). In this paper, we argue that existing simulator designs are not suitable for large heterogeneous swarms of robots. This is because designs focused on extensibility lack in scalability, while those focused on scalability lack in extensibility. We propose a novel simulator design that meets both requirements.

The result of our work is a multi-robot simulator called *ARGoS* (*Autonomous Robots Go Swarming*). Extensibility is ensured by ARGoS' highly modular architecture—robots, sensors, actuators, visualizations and physics engines are implemented as user-defined modules. Multiple implementations of each type of module are possible. The user can choose which modules to utilize in an experiment through an intuitive XML configuration file. To obtain scalability, the ARGoS architecture is multi-threaded and is designed to optimize CPU usage. Performance can be further enhanced by choosing appropriate modules. For instance, there are many possible models for each specific sensor or actuator, characterized by differences in accuracy and computational cost. Each model is implemented into an ARGoS module. By choosing the modules for an experiment, the user can allocate computational resources where necessary.

A unique feature of ARGoS is the fact that the simulated space can be partitioned into sub-spaces, each of which is managed by a different physics engine. Robots migrate seamlessly and transparently from sub-space to sub-space as they move in the environment. This feature of ARGoS enables a set of optimization opportunities (see Sec. IV) that significantly increase performance.

After the Swarmanoid project, ARGoS is now the official robot simulator of another EU-funded project, ASCENS[2]. ARGoS currently supports the Swarmanoid robots [1], [2], [3] and the e-puck [4]. ARGoS is open source and under continuous improvement[3]. It currently runs under Linux and Mac OS X.

The paper is organized as follows. In Sec. II, we discuss existing simulation designs with respect to extensibility and scalability. In Sec. III we describe the architecture of the ARGoS simulator. In Sec. IV we explain how multiple physics engines work together in ARGoS. In Sec. V we illustrate the parallelization of execution into multiple threads. In Sec. VI we report experimental scalability results. In Sec. VII we conclude the paper and indicate future research directions.

[†] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante and M. Dorigo are with IRIDIA, CoDE, Université Libre de Bruxelles, 50 Avenue F. Roosevelt, CP 194/6, 1050 Bruxelles, Belgium.

[‡] G. Di Caro, F. Ducatelle and L.M. Gambardella are with IDSIA, USI-SUPSI, Galleria 2, 6928 Manno-Lugano, Switzerland.

[§] T. Stirling is with LIS, École Polytechnique Fédérale de Lausanne, Station 11, CH-1015 Lausanne, Switzerland.

[*] Á. Gutiérrez is with ETSI Telecomunicación, Universidad Politécnica de Madrid, Avd. Complutense 30, 28040 Madrid, Spain.

[1]http://www.swarmanoid.org

[2]http://ascens-ist.eu/

[3]ARGoS can be downloaded at http://iridia.ulb.ac.be/argos/.

## II. Related Work

In the following, we describe the features of some existing multi-robot simulators with respect to the requirements for heterogeneous robot swarms: extensibility and scalability. The simulators we consider are all *physics-based*—robot bodies, sensors and actuators are simulated using physics models. Moreover, all simulators are *discrete-time*, which means that the execution proceeds synchronously in a constant step-wise fashion. A complete review of the state of the art in robot simulation is beyond the scope of this paper. We refer the interested reader to the survey of Kramer and Schultz [5].

### A. Extensibility

The design of a general and extensible simulator is a relatively recent achievement. Before the 2000s, CPU speed and RAM size on an average personal computer were insufficient to support extensible designs while ensuring acceptable simulation performance. In the last decade, a few simulators able to support different types of robots were developed. To date, the most widespread simulators of this class are Webots [6], USARSim [7] and Gazebo [8]. The engines of Webots and Gazebo are implemented with the well known open source 3D dynamics physics library ODE[4]. USARSim is based on Unreal Engine, a commercial 3D game engine released by Epic Games[5]. Although Gazebo and USARSim can support different kinds of robots, their architecture was not designed to allow the user to change the underlying models easily, thus limiting extensibility. Webots' architecture, on the other hand, provides a clean interface to the underlying ODE engine and override the way some forces are calculated. For example, Webots offers a fast 2D kinematics motion model for differential drive robots. However, extensibility is limited by the fact that it is not possible to change the implementation of sensors and actuators.

The recent multi-robot simulation framework MuRoSimF [9] tackles the issue of simulating robots of different kinds with a more general approach. In MuRoSimF, the devices forming a robot are arranged in a tree of nodes. Each node contains the code to simulate a model of a device. Each node can be further subdivided into sub-nodes to increase accuracy. This approach is very extensible and can support virtually any type of robot.

### B. Scalability

Scalability is an issue in swarm robotics systems due to the potentially high number of robots involved in an experiment. The simulators described in Sec. II-A are not designed to support large numbers of robots. The main concern in Webots, USARSim and Gazebo is accuracy, at the cost of performance. MuRoSimF is designed to support mechanically complex robots, such as humanoid robots. Typically, in swarm robotics, robots are designed to be mechanically simple, thus making MuRoSimF's computationally expensive structure unnecessary.

[4]http://www.ode.org/

[5]http://www.epicgames.com/

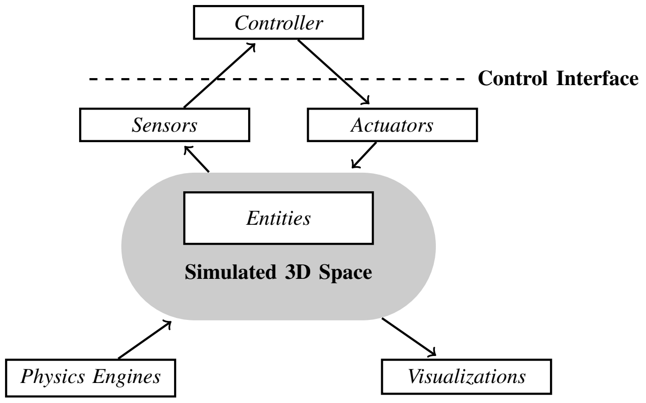


Fig. 1. The architecture of the ARGoS simulator.

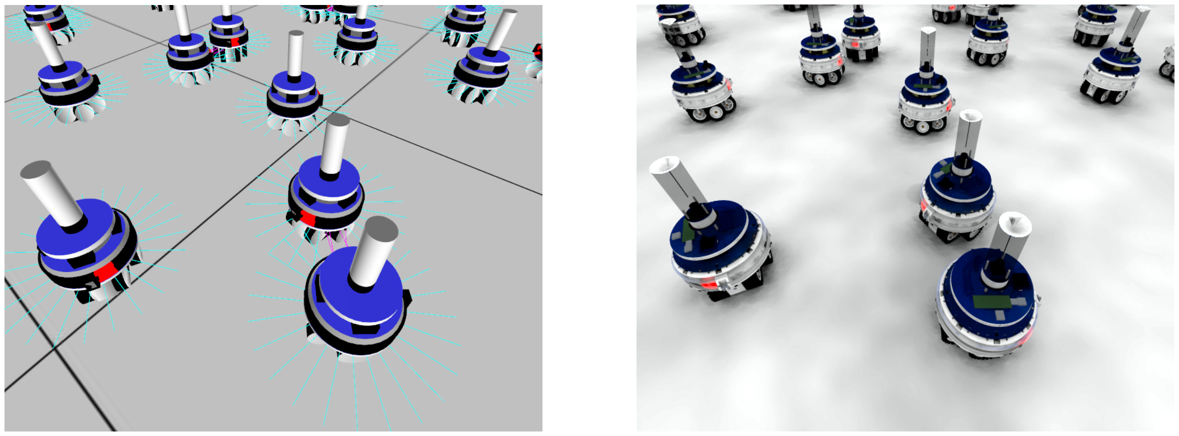
To the best of our knowledge, the only widespread simulator in the robotics community that tackles the issue of simulating thousands of robots in real-time is Stage [10]. However, this capability is obtained by imposing design and feature limitations. Stage is designed to support differential-drive robots modeled by 2D kinematics equations. Sensor and actuator models neglect noise. Stage excels at simulating navigation- and sensing-based experiments. However, due to the nature of the physics equations employed, realistic experiments involving robots gripping objects or self-assembling are not possible.

Combining scalability with extensibility is a non-trivial design challenge that has not yet been satisfactorily solved. In the following, we present the approach we followed in the design of ARGoS.

## III. The Architecture

The ARGoS architecture is depicted in Fig.1. The white boxes in the figure correspond to user-definable software modules.

*The simulated 3D space*. The core of the architecture is the *simulated 3D space*. It is a central repository containing all the relevant information about the state of the simulation. Such information is organized into basic items referred to as *entities*. ARGoS natively offers several types of entities and the user can define new types if necessary. Each type of entity stores information about a specific aspect of the simulation. For instance, a robot is typically represented in the simulated 3D space as a *composable* entity, that is, an entity that contains other entities. Entities that can compose a robot include the *controllable* entity, which stores a reference to an instance of the user-defined robot controller and its sensors and actuators, and the *embodied entity*, which stores spatial information about the robot and the way it occupies space (e.g., its position, orientation and 3D bounding box). Furthermore, entity types are organized in hierarchies. The embodied entity, for example, is an extension of the *positional* entity, which stores the position and orientation of an object in the 3D space. To enhance performance when



(a) (b)

Fig. 2. Screen-shots from different visualizations. (a) Qt-OpenGL; (b) POV-Ray.

accessing data in the simulated space, each type of entity is indexed in data structures optimized for access speed. For instance, positional entities and their extensions are indexed in several type-specific space hashes [11].

*Sensors and actuators*. Sensors are modules that read the state of the simulated 3D space. Exploiting the fact that simulated objects are composed of different entities, sensor modules need to access only specific kinds of entities to perform their calculations. For instance, a sensor module simulating a distance sensor needs to access information about embodied entities only. Analogously, actuator modules write into the components of a robot. For example, the LED actuator of a robot updates its *LED-equipped* entity component. This tight relationship between sensors/actuators and entity types has two beneficial effects: *(i)* sensors and actuators can be implemented in a generic and efficient way, taking into account specific components instead of the complete robot; *(ii)* new robots can be inserted reusing the already existing components, and all the sensors/actuators depending on those components work without modification.

*Physics engines*. Physics engines are modules that update the state of the embodied entities. As explained in more detail in Sec. IV, multiple engines of different types can be run in parallel during an experiment. Each physics engine is responsible for a subset of the embodied entities in the simulated space.

*Visualizations*. Visualization modules read the state of the simulated 3D space and output a representation of it. Currently, ARGoS offers three types of visualization: *(i)* an interactive graphical user interface based on Qt4[6] and OpenGL[7] (see Fig.2(a)), *(ii)* a high-quality rendering engine based on the well known ray-tracing software POV-Ray[8] (see Fig. 2(b)), and *(iii)* a text-based visualization designed for interaction with plotting programs such as GNUPlot[9].

*Controllers*. Robot controllers are modules interacting with the simulated space through sensors and actuators. As shown in Fig.1, the ARGoS architecture provides an abstract control interface to sensors and actuators. The control interface is the same for simulated and real robots, allowing users to develop code in simulation and seamlessly port their work to real robots.[10] ARGoS and the control interface are written in C++. However, it is possible to program the robots in other languages. The ASEBA scripting language [12] has already been integrated with ARGoS, and further language bindings (e.g., PROTO [13]) are under development.

## IV. Multiple Engines

In existing simulators such as Webots, Gazebo, USARSim and Stage, the physics engine *is* the simulated space. In ARGoS, simulated space and physics engine are distinct concepts. The link between the two concepts is realized by the embodied entities. Embodied entities are stored in the simulated space and their state is updated by a physics engine.

This novel design choice makes it possible to run multiple physics engines in parallel during an experiment. In practice, this is obtained by dividing the set of all the embodied entities into multiple subsets, and assigning to each subset a different physics engine. There are two ways to obtain suitable subsets of embodied entities. One way is to manually perform this division. For instance, in [14], [15], flying robots were assigned to a 3D dynamics engine and wheeled robots to a 2D kinematics engine. An alternative way to divide entities into subsets is by assigning non-overlapping bounded volumes of the space to different physics engines. For instance, in an indoor environment, each room and corridor can be assigned to a different physics engine. In the current implementation, the user space can be partitioned with volumes defined as arbitrarily sized prisms. The user can specify in the XML experiment configuration file what happens when a robot crosses each face of a prism. Two alternatives are possible: a face can be either a *wall* or a

[6]http://qt.nokia.com/
[7]http://www.opengl.org/
[8]http://www.povray.org/
[9]http://www.gnuplot.info/

[10]In practice, this is obtained by cross-compiling the code developed in simulation onto the real robot.

*gate*. A wall-type face is such that a robot cannot traverse it. A gate-type face is such that, when a robot traverses it, the robot migrates to another physics engine (set by the user in the configuration file). As a robot navigates the environment, its embodied entity component is updated by the physics engine corresponding to the volume in which it is located. The migration from a physics engine to another is completely transparent and performed by ARGoS automatically. The experiments presented in Sec.VI use this second division method.

To keep the state of the simulated 3D space consistent, we distinguish between *mobile* and *non-mobile* embodied entities. Embodied entities are mobile when their state (position, orientation and 3D bounding box) can change over time (e.g., robots and passive objects that can be pushed or gripped). To avoid conflicts between physics engines, mobile embodied entities can be associated to only one physics engine at a time. Conversely, embodied entities are non-mobile when their state is constant over time. Thus, they can be associated to multiple physics engines simultaneously. In this way, the structural elements of the environment (e.g., walls or columns) are shared across the physics engines, resulting in a consistent representation of the simulated 3D space.

It is important to notice that, although two robots updated by different physics engines do not physically interact, they can still communicate and sense each other (e.g., through proximity sensors or cameras). For example, consider ray-body intersection checking, which is a common method to calculate the readings of proximity sensors and cameras. In ARGoS, when a sensor casts a ray to check for intersecting bodies, it issues a query to the simulated space. In turn, the simulated space constructs a list of possible embodied entities that could intersect the ray. The list is constructed in an efficient way due to the optimized space hash that indexes the embodied entities. Each candidate embodied entity forwards the query for ray checking to the physics engine that is currently updating it. Thus, although the actual ray-body intersection is performed by the physics engine, for a sensor this is completely transparent, and two robots in different physics engines can sense each other.

The fact that robots updated by different engines do not physically interact could, in principle, lead to compenetration between two robots at opposite sides of the border between the two engines. It is up to the user to make sound choices to hinder the impact of this phenomenon. For instance, since proximity readings calculations work flawlessly across engines, an efficient obstacle avoidance routine would prevent compenetration from happening, keeping the simulation realistic. A further solution is partitioning wisely the space. For instance, flying robots could be assigned to a physics engine and wheeled robots to another. While flying robots are in the air, collision with wheeled robots can not happen. However, it is not possible to let robots self-assemble across engines. Self-assembly can only happen within an engine, and the assembled structure can subsequently navigate across engines.

**Alg. 1** Simplified pseudo-code of the main simulation loop of ARGoS. Each 'for all' loop corresponds to a phase of the main simulation loop. Each phase is parallelized as shown in Fig. 3.

```
 1: Initialize
 2: while experiment is not finished do
 3:     Visualize the simulated 3D space
 4:     for all robots do
 5:         Update sensor readings   }
 6:         Execute control step     }  sense+control
 7:     end for
 8:     for all robots do            }
 9:         Update robot status      }  act
10:     end for                      }
11:     for all physics engines do   }
12:         Update physics           }  physics
13:     end for                      }
14: end while
15: Visualize the simulated 3D space
16: Cleanup
```

Typically, physics engines perform their calculations in a local representation of the volume of space for which they are responsible. The results are then transformed into the representation of the simulated 3D space. This makes it possible to insert into ARGoS any kind of logic to update embodied entities. The user can easily add new application-specific physics engines whose local representation of the simulated 3D space is optimized for speed. At the time of writing, ARGoS natively offers four kinds of physics engines: *(i)* a 3D dynamics engine based on ODE, *(ii)* a custom 3D particle engine, *(iii)* a 2D dynamics engine based on the open source physics engine library Chipmunk[11], and *(iv)* a custom 2D kinematics engine.

The results reported in Sec.VI show that the simultaneous use of multiple physics engines has positive consequences on performance. Since embodied entities managed by different physics engines do not collide with each other, the engines must check collisions only among the embodied entities for which they are responsible. In addition, as explained in Sec. V, engines are executed in parallel threads, thus increasing CPU usage and decreasing run-time.

## V. Multiple Threads

To ensure efficient exploitation of computational resources, the main architecture of ARGoS is inherently multi-threaded. Multi-threading is embedded in the main simulation loop. During the execution of the main simulation loop, sensors and visualizations read the state of the simulated 3D space, while actuators and physics engines write into it (see Fig.1). The simulated space is thus a shared resource. Parallelizing the execution of the simulation loop could, in principle, create race conditions on the access of the simulated space. Solving race conditions with semaphores, though, is not optimal because of the high performance costs involved [16]. Thus, we designed the main loop and

[11]http://code.google.com/p/chipmunk-physics/

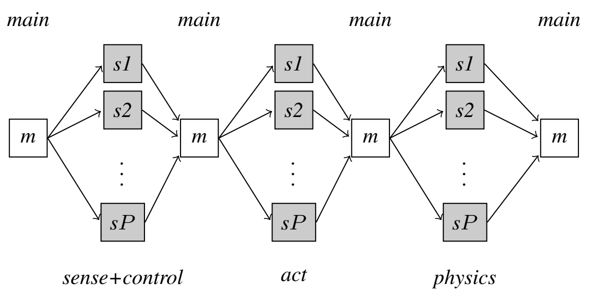


Fig. 3. The multi-threading schema of ARGoS is scatter-gather. The master thread (marked with ‘m’) coordinates the activity of the slave threads (marked with ‘s’). The *sense+control*, *act* and *physics* phases are performed by $P$ parallel threads. $P$ is defined by the user.

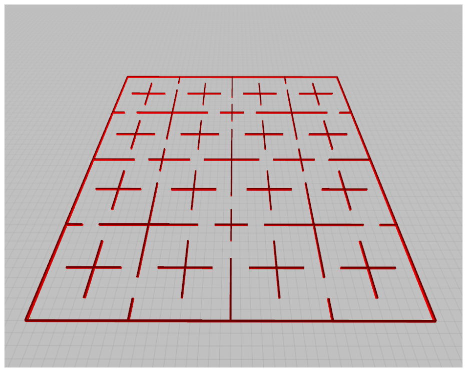


Fig. 4. A screen-shot from ARGoS showing the simulated arena created for experimental evaluation.

the simulated space so as to avoid race conditions. In this way, modules do not need to synchronize with each other or cope with resource access conflicts. As a consequence, developing new modules is easy, despite the parallel nature of the ARGoS architecture.

As can be seen from the pseudo-code reported in Alg.1, the main simulation loop is composed of three phases executed in sequence: *sense+control*, *act* and *physics*. These phases are parallelized following a scatter-gather paradigm. The three phases forming the main loop are coordinated by a master thread, marked with ‘m’ in Fig. 3, and executed by $P$ slave threads, marked by ‘s’. The number of slave threads $P$ is set by the user in the XML experiment configuration file. Each slave thread is initially idle, awaiting a signal from the master thread to proceed. When a phase is started by the master thread, the slave threads execute it and send a ‘finish’ signal back to the master thread upon completion of their part of the work.

The *sense+control* phase of the main simulation loop reads from the simulated space (lines 4–7 of Alg.1). The $C$ controllable entities stored in the simulated space are evenly distributed across the $P$ slave threads. Each thread loops through the $C/P$ controllable entities (if $C < P$, then $P - C$ threads are idle). For each controllable entity, first the sensors are executed to read the status of the simulated space and perform their calculations. Subsequently, the controller is executed. It uses the sensor readings to select the actions to perform. The actions are stored in the actuators associated to the controllable entity, but the simulated space is not updated yet (i.e., the actions are not executed). As the simulated space is only read from in this phase, race conditions are not possible.

In the two subsequent phases, the actions stored in the actuators are executed by updating the state of the entities in the simulated space. First, in the *act* phase, the actuators update the robot entity components linked to them, except for the embodied entities (lines 8–10). Analogously to the previous phase, the threads loop through $C/P$ controllable entities. Since each actuator is linked to a single robot entity component, even though actuators are executed in different threads, race conditions are not possible.

In the *physics* phase (lines 11–13) the physics engines update the mobile embodied entities in the simulated space. Denoting with $M$ the number of employed physics engines, each slave thread is responsible for $M/P$ physics engines. If $M < P$, then $P - M$ threads will be idle during this last phase. Race conditions are not possible since mobile embodied entities are assigned to only one physics engine at a time, and mobile embodied entities updated by different physics engines do not physically interact. In addition, physics engines do not need to synchronize with each other because their integration step is set to the same value.

## VI. Experimental Evaluation

In this section, we evaluate ARGoS’ scalability, focusing on the most distinctive feature of ARGoS—the possibility to run multiple physics engines in parallel. To highlight the performance benefits, we limit experiments to a single type of robot and multiple instances of one type of physics engine, and we do not use any visualization. For examples of experiments that utilize different types of physics engines and different kinds of robots, see [14], [15], [17], [18], [19], [20].

### A. Experimental Setup

To date, there is little work in assessing the performance of multi-robot simulators for thousands of robots. For this reason, in the literature no standard benchmark has been proposed. To the best of our knowledge, the only simulator whose scalability was studied for thousands of robots is Stage. In [10], Vaughan studies Stage’s performance in a very simple experiment in which robots disperse in an environment while avoiding collisions with obstacles. The rationale for this choice is that typically the performance bottleneck is in checking and solving collisions among the simulated objects. The robot controllers are intentionally kept simple and minimal to highlight the performance of the simulator, while performing a fairly meaningful task.

For our evaluation, we employ an experimental setup similar to Vaughan’s. Fig.4 depicts a screen-shot of the environment in which the robots disperse. It is a square whose sides are 40 m long. The space is structured into a set of connected rooms that loosely mimic the layout of a real indoor scenario. Analogously to the evaluation of

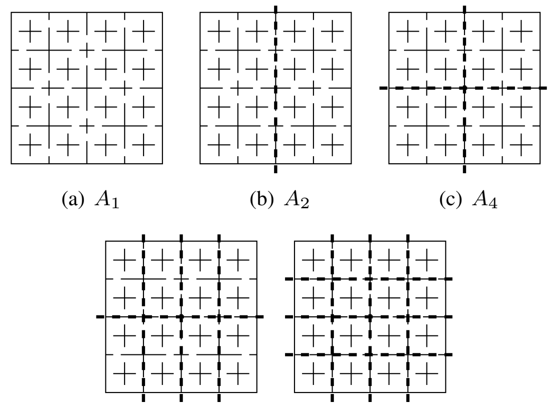

(a) $A_1$ (b) $A_2$ (c) $A_4$ (d) $A_8$ (e) $A_{16}$

Fig. 5. The different space partitionings ($A_1$ to $A_{16}$) of the environment used to evaluate ARGoS' performance. The bold dashed lines indicate the borders of each region. Each region is updated by a dedicated instance of a 2D dynamics physics engine.

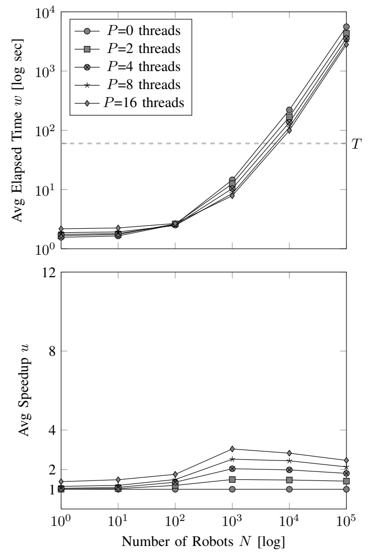


Fig. 6. Average wall clock time and speedup for a single physics engine ($A_1$). Each point corresponds to a set of 40 trials with a specific configuration $\langle N,P,A_1\rangle$. Each experiment simulates $T = 60$ s. Points under the dashed line in the upper plot mean that the simulations were faster than real time; above it, they were slower. Standard deviation is omitted because its value is so small that it would not be visible in the graph.

Stage, which was performed with a basic robot model, in our experiments we use the simplest robot available in ARGoS: the e-puck [4]. Each robot executes a simplified version of Howard *et al.*'s dispersal algorithm [21]. To keep the evaluation meaningful with respect to typical use cases, we run all the experiments with 2D dynamics physics engines, including collision checking and complete calculations of physical forces. We use the physics engine based on Chipmunk, a fast 2D physics library largely used in games and physics-based simulations.

We employ as performance measures two standard quantities. The first is the *wall clock time* ($w$), which corresponds to the elapsed real time between application start and end. To reduce noise, we run our experiments on dedicated machines in which the active processes were limited to only those required for a normal execution of the operating system. The second performance measure is the *speedup* ($u$). To calculate it, we first measure the total CPU time $c$ obtained by the process running the experiment. The difference between $w$ and $c$ is that the latter increases only when the process is actively using the CPU. The total CPU time $c$ is calculated as the sum of the CPU times obtained by the process on each core $c_i$: $c = \sum_i c_i$. The speedup is defined as $u = c/w$. In single-core CPUs or in single-threaded applications, $u \leq 1$. With multi-threaded applications on multi-core CPUs, the aim is to maximize $u$, $u \gg 1$.

We analyze the effect on $w$ and $u$ of different configurations of our experiment. In particular, we identify three factors that strongly influence the performance measures: *(i)* the number of robots $N$, *(ii)* the number of parallel slave threads $P$, and *(iii)* the way the environment is partitioned into multiple physics engines. Concerning the number of robots, we run experiments with $N = 10^i$, where $i \in [0,5]$. To test the effect of the number of threads $P$, we run our experiments on four machines with 16 cores[12], and let $P \in \{0,2,4,8,16\}$. When $P = 0$, the master thread executes everything without spawning the slave threads. Finally, we define five ways to partition the environment among multiple physics engines, differing from each other in how many engines are used and how they are distributed. We refer to a partitioning with the symbol $A_E$, where $E \in \{1,2,4,8,16\}$ is the number of physics engines employed. $E$ also corresponds to the number of regions in which the space is partitioned, i.e., each engine is responsible for a single region. The partitionings are depicted in Fig.5. For each experimental setting $\langle N, P, A_E\rangle$, we run 40 trials. The simulation time step is 100 ms long. Each trial simulates $T = 60$ s of virtual time, for a total of 600 time steps. In order to avoid artifacts in the measures of $w$ and $u$ due to initialization and cleanup of the experiments, the measures of wall clock time and speedup are taken only inside the main simulation loop.

[12]Each machine has two AMD Opteron Magny-Cours processors type 6128, each processor with 8 cores. The total size of the RAM is 16 GB.

## *B. Results with a Single Physics Engine*

Fig.6 shows the average wall clock time and speedup of 40 experiments in environment partitioning $A_1$ (a single physics engine updates all the robots) for different values of $N$ and $P$. The graphs show that more threads result in better performance when the number of robots is greater than 100. In particular, the lowest wall clock times are obtained when $P = 16$. Focusing on $N = 100{,}000$ and comparing the values of $w$ when using the maximum number of threads and when using no threads at all, we see that $w(P = 16)/w(P = 0) \approx 0.5$.

The aim of our analysis is to study scalability for large values of $N$. However, it is useful to explain why, when the number of robots is smaller than 100, the threads impact negatively on wall clock time. If we consider the time cost of managing multiple threads, we see that when the robots are few in number, the time taken to assign work to the threads is comparable to the time taken by a thread to perform the work. Thus, it is faster to let the master thread perform all the work. This result is also matched by the speedup values, which are only marginally better than single-threaded computation—when $N = 1$, $u(P = 2) \approx 1.01$ and $u(P = 16) \approx 1.39$.

Furthermore, regarding speedup, for all values of $P > 0$, $u$ is greater than 1. When $N = 1{,}000$, the highest speedup $u \approx 3.04$ occurs for $P = 16$. For larger values of $N$, the speedup decreases. This decrease occurs as only one physics engine is responsible for the update of all the robots. Therefore, when $P \geq 2$, only one thread runs the engine, while the other $P - 1$ must stay idle, not contributing to the measure of $c$ (however, the first two phases of the main simulation loop are still executed in parallel). Therefore, the more robots take part in the simulation, the more time the slave thread in charge for physics will spend working while the other threads stay idle—a situation analogous to a single-thread scenario, in which $w$ increases faster than $c$, thus resulting in a lower $u$.

## C. Results with Multiple Physics Engines

Using multiple physics engines has a beneficial impact on performance, as shown in Fig.7. For $A_2$ the behavior of $w$ and $u$ is still analogous to $A_1$—multiple threads are worth their time cost for $N > 100$ and $u$ presents a peak, this time for $N = 10{,}000$. Comparing the best wall clock times measured for $N = 10{,}000$ (which are obtained when $P = 16$), $w(A_2)/w(A_1) \approx 0.61$ and $w(A_2) \approx T$. Therefore, with only two engines, ARGoS can already simulate 10,000 robots in approximately real-time.

Using more engines improves both $w$ and $u$. Not surprisingly, when $N = 10{,}000$, the best values for wall clock time and speedup are reached for the highest number of space partitions ($A_{16}$) and for the highest number of threads employed ($P = 16$). In this configuration, the ratio between the measured wall clock time and the simulated virtual time $T$ is 0.6, which gives the remarkable result that a simulation of 10,000 robots can be performed 40% *faster than real-time*. For 100,000 robots, wall clock time is about $10T$, which is a reasonable value for many applications. Also, the speedup reaches its maximum value ($\approx 11.21$) when $N = 100{,}000$ and $P = 16$.

## D. Comparison with Stage

Stage's performance evaluation [10] was run on an Apple MacBook Pro, with a 2.33 GHz Intel Core 2 Duo processor and 2 GB RAM. For our evaluation, each core in the machines we employed provides comparable features: 2 GHz speed, 1 GB RAM per thread when $P = 16$.

Experiments conducted in a setup analogous to ours (no graphics, large environment with obstacles, simple robots) show that Stage can simulate about 1,000 robots in real-time. In comparison, when no threads are employed and a single physics engine is responsible for the entire environment, ARGoS simulates 1,000 robots 76% faster than real time. ARGoS performance is further enhanced by the use of threads. With 2 threads and a single physics engine, 1,000 robots are simulated 79% faster than real time. Increasing the number of threads to 16, 1,000 robots are simulated 87% faster than real time. When 16 physics engines are employed, 1,000 robots are simulated 91% faster than real time.

Moreover, it is worth remarking that, in our experiments, we employed a realistic 2D dynamics physics engine, whereas Stage is based on a simpler 2D kinematics physics engine.

## VII. Conclusions and Future Work

In this paper, we introduced ARGoS, a simulator designed for large heterogeneous swarms of robots. With respect to existing simulators, ARGoS offers a more *extensible* architecture that *(i)* enables the user to allocate accuracy (and therefore CPU resources) to the relevant parts of an experiment, and *(ii)* makes it easy to modify or add functionality in the form of modules, promoting exchange and cooperation among researchers. A unique feature of ARGoS is that *multiple physics engines* can be used at the same time, partitioning the space into independent sub-spaces. Each sub-space can have its own update rules, and these update rules can be optimized for the experiment at hand. Robots can migrate from a physics engine to another transparently. In addition, the *multi-threaded architecture* of ARGoS proves very *scalable*, showing low run-times and high speedup on multi-core CPUs. Results show that ARGoS can simulate 10,000 robots 40% faster than real-time, using multiple 2D dynamics physics engines.

Future work involves reaching real-time performance for swarms composed of hundreds of thousands of robots. Possible approaches may be: *(i)* employing a heterogeneous threading model performing the computation both on CPU and GPU [22] and *(ii)* modifying the multi-threaded architecture of ARGoS into a mixed multi-thread/multi-process architecture, in which physics engines and the simulated space are distributed across different machines in a network.

## Acknowledgments

This research was carried out in the framework of Swarmanoid, a project funded by the Future and Emerging Technologies programme (IST-FET) of the European Commission under grant IST-022888. This work was also partially supported by the ERC Advance Grant "E-SWARM: Engineering Swarm Intelligence Systems" (grant 246939), and by the EU project ASCENS (grant 257414). Manuele Brambilla acknowledges support from the Fund for Industrial and Agricultural Research FRIA-FNRS of Belgium's French Community. Arne Brutschy and Marco Dorigo acknowledge support from the Belgian F.R.S.-FNRS, of which they are Research Fellow and Research Director, respectively.

## References

[1] M. Bonani, V. Longchamp, S. Magnenat, P. Rétornaz, D. Burnier, G. Roulet, F. Vaussard, H. Bleuler, and F. Mondada, "The marXbot,

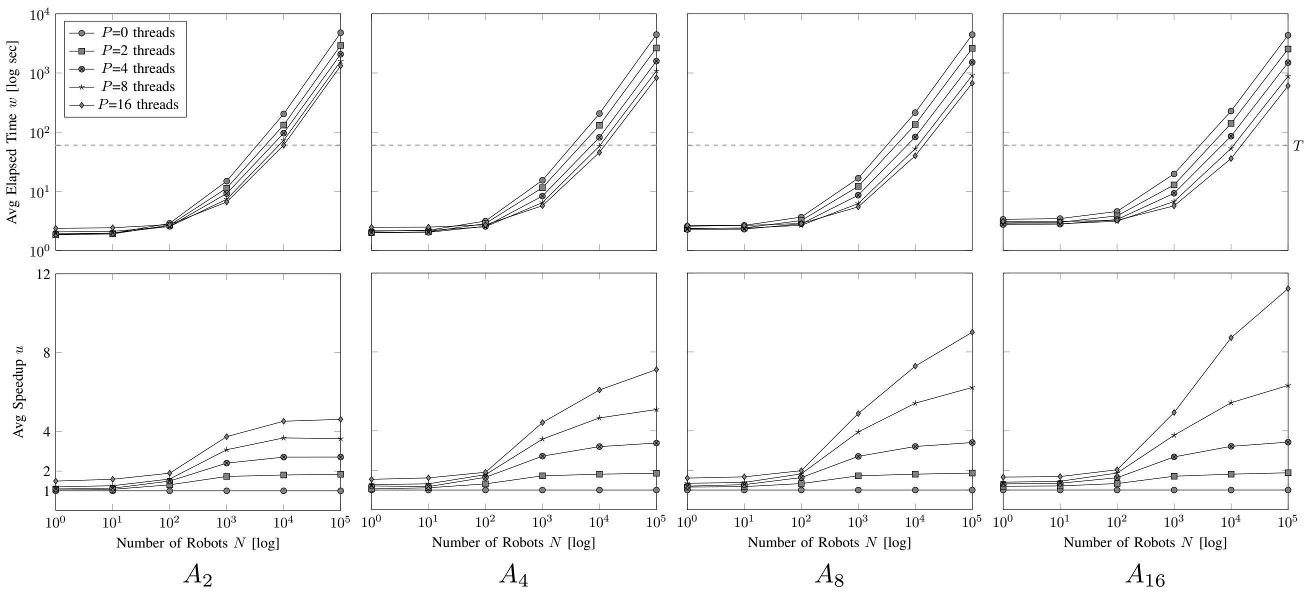


Fig. 7. Average wall clock time and speedup for partitionings $A_2$ to $A_{16}$. Each point corresponds to a set of 40 trials with a specific configuration $\langle N, P, A_E \rangle$. Each experiment simulates $T = 60$ s. Points under the dashed line in the upper plots mean that the simulations were faster than real time; above it, they were slower. Standard deviation is omitted because its value is so small that it would not be visible in the graph.

a miniature mobile robot opening new perspectives for the collective-robotic research," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Piscataway, NJ: IEEE Press, 2010, pp. 4187–4193.

[2] J. Roberts, T. Stirling, J. Zufferey, and D. Floreano, "Quadrotor using minimal sensing for autonomous indoor flight," in *European Micro Air Vehicle Conference and Flight Competition (EMAV)*, 2007, proceedings on CD-ROM.

[3] M. Bonani, S. Magnenat, P. Rétornaz, and F. Mondada, "The hand-bot, a robot design for simultaneous climbing and manipulation," in *Proceedings of the Second International Conference on Intelligent Robotics and Applications (ICIRA 2009)*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2009, vol. 5928, pp. 11–22.

[4] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli, "The e-puck, a robot designed for education in engineering," in *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*. Castelo Branco, Portugal: IPCB, 2009, vol. 1, pp. 59–65.

[5] J. Kramer and M. Schultz, "Development environments for autonomous mobile robots: a survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, 2007.

[6] O. Michel, "Cyberbotics Ltd. – Webots: Professional mobile robot simulation," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, pp. 39–42, March 2004.

[7] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "USARSim: a robot simulator for research and education," in *Proceedings of the IEEE Conference on Robotics and Automation (ICRA)*. Piscataway, NJ: IEEE Press, 2007, pp. 1400–1405.

[8] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Piscataway, NJ: IEEE Press, 2004, pp. 2149–2154.

[9] M. Friedman, "Simulation of autonomous robot teams with adaptable levels of abstraction," Ph.D. dissertation, Technische Universität Darmstadt, Germany, 2010.

[10] R. Vaughan, "Massively multi-robot simulation in Stage," *Swarm Intelligence*, vol. 2, no. 2, pp. 189–208, 2008.

[11] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross, "Optimized spatial hashing for collision detection of deformable objects," in *Proceedings of the Vision, Modeling, and Visualization Conference*. Heidelberg, Germany: Aka GmbH, 2003, pp. 47–54.

[12] S. Magnenat, P. Rétornaz, M. Bonani, V. Longchamp, and F. Mondada, "ASEBA: A modular architecture for event-based control of complex robots," *IEEE/ASME Transactions on Mechatronics*, vol. PP, no. 99, pp. 1–9, 2010.

[13] J. Bachrach, J. Beal, and J. McLurkin, "Composable continuous-space programs for robotic swarms," *Neural Computation & Applications*, vol. 19, pp. 825–847, 2010.

[14] N. Mathews, A. Christensen, E. Ferrante, R. O'Grady, and M. Dorigo, "Establishing spatially targeted communication in a heterogeneous robot swarm," in *Proceedings of 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010)*. Toronto, Canada: IFAAMAS, 2010, pp. 939–946.

[15] C. Pinciroli, R. O'Grady, A. Christensen, and M. Dorigo, "Self-organised recruitment in a heterogeneous swarm," in *The 14th International Conference on Advanced Robotics (ICAR 2009)*, 2009, p. 8, proceedings on CD-ROM, paper ID 176.

[16] A. S. Tanenbaum, *Modern Operating Systems*, 2nd ed. New Jersey, NJ: Prentice-Hall, 2001.

[17] G. Di Caro, F. Ducatelle, C. Pinciroli, and M. Dorigo, "Self-organised cooperation between robotic swarms," *Swarm Intelligence*, vol. 5, no. 2, pp. 73–96, 2011.

[18] F. Ducatelle, G. Di Caro, and L. Gambardella, "Cooperative self-organization in a heterogeneous swarm robotic system," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. New York, NY: ACM, 2010, proceedings on CD-ROM.

[19] M. A. Montes de Oca, E. Ferrante, N. Mathews, M. Birattari, and M. Dorigo, "Opinion dynamics for decentralized decision-making in a robot swarm," in *Proceedings of the Seventh International Conference on Swarm Intelligence (ANTS 2010)*, ser. LNCS 6234, M. Dorigo *et al.*, Eds. Berlin, Germany: Springer, 2010, pp. 251–262.

[20] E. Ferrante, M. Brambilla, M. Birattari, and M. Dorigo, "Socially-mediated negotiation for obstacle avoidance in collective transport," in *International Symposium on Distributed Autonomous Robotics Systems (DARS)*, ser. Advanced Robotics Series. Springer, 2010, in press.

[21] A. Howard, M. Matarić, and G. Sukhatme, "Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem," in *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems (DARS)*. New York: Springer, 2002, pp. 299–308.

[22] D. W. Holmes, J. R. Williams, and P. Tilke, "An events based algorithm for distributing concurrent tasks on multi-core architectures," *Computer Physics Communications*, vol. 181, no. 2, pp. 341–354, February 2010.