ARGoS: a Pluggable, Multi-Physics Engine Simulator for Heterogeneous Swarm Robotics

C.Pinciroli, V.Trianni, R.O'Grady, G.Pini, A.Brutschy, M.Brambilla, N.Mathews, E.Ferrante, G.Di Caro, F.Ducatelle, T.Stirling, Á.Gutiérrez, L.M.Gambardella and M.Dorigo

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2010-026

December 2010
Last revision: February 2011

IRIDIA – Technical Report Series
ISSN 1781-3794

Published by:

IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2010-026

Revision history:

TR/IRIDIA/2010-026.001	December 2010
TR/IRIDIA/2010-026.002	February 2011

ARGoS: a Pluggable, Multi-Physics 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, Frederic Ducatelle,
Timothy Stirling, Álvaro Gutiérrez,
Luca Maria Gambardella and Marco Dorigo

February 17, 2011

Abstract

We present a novel robot simulator called ARGoS. The main focus of ARGoS is the real-time simulation of massive heterogeneous swarms of robots. In contrast to existing robot simulators, which obtain scalability by imposing limitations to the extent and accuracy of the robot models, in ARGoS we pursue a deeply modular approach that allows the user both to add custom features easily and to allocate computational resources where needed by the experiment at hand. In this respect, a unique feature of ARGoS is the possibility to use multiple physics engines for different parts of the environment. The physics engines can be of different kinds and robots can migrate from one to another in a transparent way. This feature enables a whole new set of optimizations to improve scalability and paves the way for a new approach to parallelism in robotic simulation. Results show that ARGoS can simulate about 10,000 wheeled robots with full dynamics in real-time. We discuss possible improvements to the architecture of ARGoS to achieve simulations of millions of entities in real-time.

1 Introduction

Heterogeneous robotic systems are attracting increasing attention in the research community, as witnessed by the pioneering projects FRONTS[1] and Swarmanoid[2]. The robotic platforms studied in these projects share three common aspects: *(a)* a potentially large number of devices involved, *(b)* a high grade of diversity among them, both at the physical and at the functional level, and *(c)* the prospect of a wide impact in the research community, making these platforms an enabler for a variety of innovative studies in the near future.

The growth of these new systems is mirrored by the parallel evolution of novel software tools to design, develop and study them. For instance, Bachrach

[1] http://fronts.cti.gr/
[2] http://www.swarmanoid.org

et al. have recently proposed a new programming paradigm, PROTO [1], that allows a developer to abstract away from the specific actions and hardware capabilities of individual devices in a heterogeneous swarm, and consider the system as a unique, time- and space-aware entity.

In this paper, we argue that the distinctive features of heterogeneous robotic systems *(a)-(c)* stress the limits in the scalability and flexibility of existing robot simulators. We present a novel simulator named *ARGoS (Autonomous Robots Go Swarming)* explicitly designed to address and solve the new issues arising from the study of this new kind of robotic systems.

In the quest for an effective design, a first important aspect to consider is that the study of heterogeneous robotic systems started recently and it is still in a very early stage. It is impossible to foresee what kind of devices will be developed in the next future. This calls for a particular flexibility requirement—the support for robots with completely different features, both at the functional and at the physical level. For instance, the Swarmanoid project studied the coordination of three kinds of robots: wheeled robots with self-assembly capabilities (*foot-bots* [3]), flying robots (*eye-bots* [21]) and climbing robots with manipulation capabilities (*hand-bots* [4]). Allowing capable users to add new devices (robots, sensors, actuators) in a comfortable manner, without the need to modify the core of the architecture, is a non-trivial design challenge.

This kind of flexibility must coexist with more common needs. In research, experiments may vary dramatically across different works and different people. Even though some features, such as robot motion, are almost always necessary, many other features are linked to the type of experiment at hand. For instance, the quantities on which statistics must be calculated depend on the experiment. Also, if the environment presents custom dynamics, such as objects being added or removed as a result of the actions of the robots, these mechanisms need to be implemented in the simulator. The need for specific and often divergent features could make the design of a generic simulator intractably complex. Even worse, adding features usually renders the learning curve of a tool much steeper, hindering usability and maintainability, a phenomenon often referred to as *feature creep*.

In addition, the very nature of the study of heterogeneous swarms calls for high degrees of scalability for increasing numbers of individuals. To the best of our knowledge, and as discussed in more detail in Section 2, the only general-purpose simulator that explicitly addresses scalability for at least 1,000 robots is Stage [24]. However, this result is obtained imposing assumptions and severe limitations on the simulated models and the extent of their modification.

Marrying a high deal of flexibility with extreme scalability was another big challenge in the design of ARGoS. As it will be explained in more detail, flexibility is granted by the deeply modular architecture of ARGoS in which all the main components are plug-ins—including robots, sensors, actuators and visualizations. A distinctive feature of ARGoS is the fact that multiple physics engines can be run in parallel during an experiment. Physics engine are themselves plug-ins, so experienced users can add new ones if necessary. The careful choice of which plug-ins to use for an experiment is one of the means to obtain scalability. In addition, the architecture of ARGoS was designed to be inherently multi-core. As the reported results show, ARGoS profits of multi-core architectures to cut run-time and exploit CPU resources efficiently.

In the rest of the paper, we describe the main principles and choices that drove us in the design and implementation of ARGoS. In Section 2, we describe existing simulators and related work. Subsequently, we illustrate the main features of ARGoS in Section 3. The focus of Section 4 is on the design of the simulator. The experimental evaluation of its scalability is covered by Section 5. We discuss possible improvements to achieve the goal of simulating millions of robots in Section 6. The paper is concluded by Section 7.

2 Related Work

In the following, we discuss the features of existing multi-robot simulators. We limit the discussion to those simulators that are widely used in the research community and provide some deal of generality. A complete review of the state of the art in robot simulation is beyond the scope of this paper. We suggest the interested reader to refer to the survey of Kramer and Schultz [17].

Stage/Gazebo [10, 16, 8] [3] The aim of the Player Project is to produce free software to enable research in robot and sensor systems. Software code is developed by an international team of robotics researchers and used in many laboratories around the world. Player is a robot server that provides full access and control of a robotic platform and of all its sensors and actuators. Stage is a scalable simulator that is interfaced to Player. It simulates a population of mobile robots moving and sensing in a two-dimensional bitmapped environment. Physics is simulated in a purely kinematic fashion and noise is explicitly neglected. These design choices make it possible to simulate thousands of robots in real-time on an average laptop. Various sensor models are provided, including sonar, scanning laser rangefinder, pan-tilt-zoom camera with color blob detection and odometry. Stage devices present a standard Player interface so few or no changes are required to move between simulation and hardware. In November 2010, a new version of Stage was released in which, among other features, support for multi-thread simulations was added. Gazebo is a multi-robot simulator that extends Stage's capabilities for 3D outdoor environments. It generates both realistic sensor feedback and physically plausible interactions between objects using ODE's libraries for the simulation of rigid-body physics. Gazebo presents a standard Player interface in addition to its own native interface. Controllers written for the Stage simulator can generally be used with Gazebo without modification (and vice-versa).

USARSim [6] [4] Urban Search and Rescue Simulation (USARSim) is a high fidelity multi-robot simulator that was originally developed in the context of the search and rescue (SAR) research activities of the Robocup contest and which is now becoming one of the most complete general-purpose tools for robotics in research and educations. Its development is driven by a large community of researchers. It builds upon a widely used and affordable commercial game engine, the Unreal Engine 2.0, produced by Epic Games, which provides good accuracy of physics simulation, a number of geometrical and physical models, acceptable computational speed,

[3] `http://playerstage.sourceforge.net`
[4] `http://usarsim.sourceforge.net`

and some deal of flexibility. Robots are customizable and can be controlled by a client program through a TCP socket connection. USARSim provides a large collection of robot models, including wheeled, legged, flying, and underwater ones, and of fully configurable sensors and actuators with associated noise models. Quantitative evaluations show a close correspondence between results obtained within USARSim and with the corresponding real world system or sensor.

Webots [18] [5] Webots is a commercial robotic simulator developed by Cyberbotics Ltd. It provides an ODE-based accurate physics simulation and several models of real robots such as Sony Aibo, Khepera, or Pioneer2. The robots and the environment are described using the VRML standard for graphical models, extended by nodes for the Webots elements, sensors, and physical attributes. Mobile robots with any physical characteristics can be designed, including flying, wheeled, and legged ones. Controllers can be programmed in C++ or Java and connected to third party software through a TCP/IP interface. An extensive library of tunable sensors and actuators is provided, including distance and global positioning sensors, compass, cameras, radio transmitters, incremental encoders, etc. A powerful graphical visualization is realized with the use of OpenGL libraries. Webots serves a large community of users across the globe and undergoes continual updating.

ÜberSim [5] [6] The ÜberSim simulator is developed at Carnegie Mellon with the intent to create an open source high-fidelity simulator for dynamic robot soccer scenarios that enables rapid development of control systems that can be transferred to real robots with a minimum of overhead. ÜberSim makes use of ODE to provide realistic dynamics including motions and physical interactions. It is targeted towards providing parametrized robot classes that are easy to extend and reconfigure. Exploiting ODE's capabilities, it provides the definition and use of robot shapes and actuators which are generic enough to simulate a wide range of robot types. Custom robots can be modeled by programming their structure in C classes. ÜberSim has a client/server architecture, where clients communicate with the server over TCP sockets.

Breve [14] [7] Breve is a simulation package designed for realistic simulations of large distributed and artificial life systems in continuous 3D worlds with continuous time. Simulations are written by defining the behaviors and interactions of agents using a simple object-oriented programming language called *Steve* or in Python. Breve makes use of the ODE physics engine libraries to provide facilities for rigid body simulation, collision detection/response, and articulated body simulation. It is intended to permit the rapid construction of complex multi-agent simulations in realistic physical environments. Breve includes an OpenGL display engine that allows observers to manipulate the perspective in the 3D world and view the agents from any location and angle. Users can interact at run-

[5] `http://www.cyberbotics.com`
[6] `http://www.cs.cmu.edu/~robosoccer/ubersim`
[7] `http://www.spiderland.org`

time with the simulation using a web interface. Multiple simulations can interact and exchange individuals over the network.

TODO: Discuss the limits of existing simulators and relate them to heterogeneous swarms of robots.

3 Main Features

In this section, we highlight the main features of ARGoS, that will be more technically discussed in the rest of the paper.

3.1 The Simulator

ARGoS is a *physics-based* simulator. This means the agent bodies are simulated through physics models. This kind of simulation is often referred to as *embodied*. Moreover, ARGoS is a *discrete-time* simulator, which means that simulation proceeds synchronously in a constant step-wise fashion. The need for scalability and flexibility puts forwards a set of novel problems, that, as discussed in Section 1, stress the limits of existing simulator designs. We solved these issues through a set of novel design choices.

3.1.1 Multiple Physics Engines

ARGoS gives one the possibility to run multiple physics engines in parallel. This feature is obtained by decoupling the simulated space from the physics rules that update it. A physics engine is, in other words, a *view* of the simulated space. For instance, the global 3D space could be decoupled into a 2D space corresponding to the floor to simulate simple wheeled robots and a 3D engine with avionics equations to simulate flying robots. Alternatively, different portions of the environment can be managed by different physics engines. For example, in an environment composed of different rooms, some rooms can be simulated by 3D physics engines and others by 2D engines, depending on the tasks to be performed by the robots.

In ARGoS, we assume that two robots managed by different physics engines cannot collide with each other. However, partitioning the physical space in this way does not influence other kinds of interactions between the robots, such as communication. Furthermore, all robots can still perceive each other, for instance through proximity sensors and cameras. The partitioning is only at the level of which physics rules are used to update the simulated objects.

Another important aspect is that entities are not confined into a physics engine for the entirety of an experiment. It is possible, if needed, to move entities from one physics engine to another. This makes it possible, for example, to simulate at maximum speed simple navigation in a corridor, while ensuring the needed accuracy to those robots executing a more complex task inside a room. When robots enter or exit the room, they switch physics engine. Once again, at the level of the robot this is completely transparent—ARGoS performs the transfer internally.

The use of multiple physics engines presents a number of positive consequences. Existing, well established physics engine libraries, such as Chipmunk[8], Bullet[9] and ODE[10], can be easily integrated into ARGoS. Custom physics engines optimized for particular applications, such as particle engines or even grid-based worlds, are easy to add too. The choice of the right physics engine is left to the user as a result of considerations about where accuracy is actually needed.

Moreover, since entities in different physics engines do not collide with each other, the engines must check collisions only among the entities they manage. In a simulation with thousands of agents, if engines are carefully used the speedup can be relevant, as shown in Section 5. In this way, even simple, custom physics engines with poor scalability may be used in concert with others, resulting in little impact on the overall performance.

Finally, the engines run in parallel, as explained in Section 3.1.4. In this way, even though an individual engine is not inherently multi-thread, modern multi-core architectures can be exploited efficiently.

3.1.2 Everything is a Plug-in

ARGoS supports a collaborative approach to development thanks to its modular architecture. New modules can be added in an easy way, due to the fact that the interfaces to implement are intentionally kept simple. Modules can be implemented by different people in parallel, and be made readily available to other users. In ARGoS, modules are implemented as plug-ins and basically *everything is a plug-in*. For instance, besides physics engines, also robot controllers and visualizations are plug-ins.

Robot sensors and actuators are particular plug-ins. From the control point of view, the model of a robot is mainly encoded into how sensors and actuators interact with the simulated environment. Different experiments typically require different models, and, even in a specific experiment, it is often easier to start with a basic sensor/actuator model (maybe noiseless) to kick-start the work, and then pass on to more complex (and realistic) models. For these reasons, developers can define multiple implementations of the same robot sensor or actuator, and users can choose the one that suits best the experiment at hand. This is another way (in addition to multiple physics engines) to tune modeling accuracy where needed, while ensuring maximum performance.

Also the simulated entities that populate the global space are plug-ins of a special kind, in that they can be built starting from simpler entity components, which are entities too. This feature allows developers to insert new robots in a clean and easy way, reusing the basic functionality offered by ARGoS. Moreover, in some parts of ARGoS, such as those concerning physics and communication, the fact that robots are nothing but a set of components made it easier to write generic, clean and optimized code. An instance of this set of optimizations is described in Section 4.3.

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

[9]http://code.google.com/p/bullet/

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

3.1.3 The Loop Functions

To cope with the explosion of very specific and short-lived features (feature creep) described in Section 1, in ARGoS we followed the common approach of providing function hooks in strategical points of the simulation loop. The hooked functions are user-defined and in ARGoS they are called *loop functions*. There are hooks to customize the initialization and the end of an experiment, as well as before and after each simulation step. It is also possible to define custom end conditions for an experiment.

Loop functions allow one to access and modify the entire simulation. In this way, the user can collect figures and statistics, and store complex data for later analysis. It is also possible to interfere with the simulation, by moving, adding or removing entities in the environment, or changing their internal state.

Finally, the loop functions are the place where many new features are prototyped before being promoted to the main code.

3.1.4 Multi-Thread Architecture

Scalability is a major issue in simulators, even more so in swarm robotics applications. Thus, it is of vital importance to exploit the resources of modern multi-core CPU architectures in the best way possible.

Unfortunately, most of the wide spread libraries relevant for simulation are not currently multi-thread, although a significant body of research is currently dedicated to the design of parallel physics engines [12]. For instance, most freely available physics engine libraries (such as Chipmunk, ODE and Bullet) are still single-thread.

To overcome this issue and ensure a more efficient exploitation of computing resources, the main architecture of ARGoS is inherently multi-thread. The main simulation loop (which involves the simulation of robot sensors and actuators, as well as control code of each robot) are executed in parallel threads. Such threads are also responsible for the physics engines. Interestingly, as discussed more in detail in Section 4.2, multi-threading does not affect plug-in development, since the architecture was carefully designed to avoid race conditions by *(i)* tightly compartmentalizing reading and writing phases and *(ii)* carefully partitioning shared resources. Experiments show that threads improve performance significantly and that ARGoS exploits computational resources efficiently (see Section 5).

3.2 Behavioral Composition

Besides modularity, another feature of ARGoS that enables cooperative development is the fact that robot control code is composable. In fact, control code can be decomposed into modules called *behaviors*. ARGoS offers means to code behaviors individually and then to put them together [9]. In this way, robot control code is reusable. Researchers can concentrate on their part of the work while exploiting the effort of other people.

In addition to this “native” set of facilities for behavioral decomposition, a more low-level access to the robot interface is possible. In this way, a user can

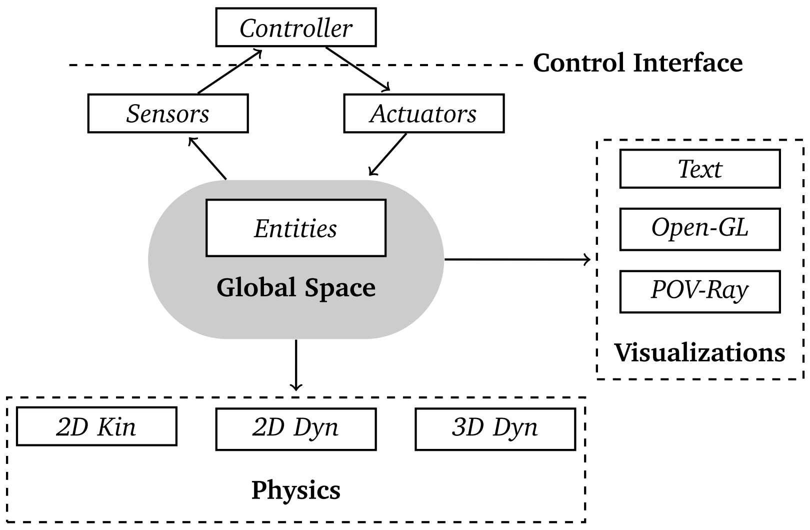


Figure 1: The architecture of the ARGoS simulator.

integrate more sophisticated software architectures, such as ROS[11], YARP[12], OROCOS[13], Carmen[14], Orca[15], or MOOS[16].

3.3 Transfer to Real Robots

In addition to simulation capabilities, ARGoS offers to the user all the needed tools in the development cycle of robot control code, from design to validation on real robots.

For instance, code developed in simulation can be easily ported to real robots. A robot controller, in fact, is implemented using an interface called *common control interface* that abstracts the details of the underlying device, be it simulated or real. For a user, therefore, there is no difference between coding for simulation or reality.

Along with this abstraction, cross compilation is ensured by supporting the most common compilation tool-chains.

Ongoing work in this matter is concentrated into adding support for more high-level languages for robot code control, such as PROTO [1] and Herbal [7].

4 The Simulator

In the following, we will describe in detail the design of the most relevant parts of the architecture of the ARGoS simulator.

[11] http://www.ros.org/wiki/ROS/Introduction
[12] http://eris.liralab.it/yarp/
[13] http://www.orocos.org/
[14] http://carmen.sourceforge.net/
[15] http://orca-robotics.sourceforge.net/
[16] http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php

4.1 Modular Architecture

The architecture of ARGoS is the result of the synthesis of different, often diverging requirements. User needs such as flexibility (for instance, the possibility to tune the accuracy of the models) and scalability for increasing number of simulated entities must coexist with ease of use and more prosaic engineering principles: simplicity, maintainability, symmetry, standardization, which in turn make the addition of new extensions easier for developers. We opted for a deeply modular architecture, in which all the parts that users may want to override are plug-ins. A pictorial representation of the ARGoS architecture is reported in Figure 1.

The glue that keeps everything together is a set of basic interfaces and a global simulated space. The global space contains all the simulated entities and stores their global position and orientation, along with other information not related to physics, such as the robots' current LED colors.

The position and orientation of the entities is updated by the physics engines. To this aim, physics engines contain all the relevant information to update the entities under their responsibility, such as position, orientation, speed, applied forces, and so on. It is important to notice that, as discussed in Section 3.1.1, physics engines correspond to a view of the global space. A two dimensional physics engine located on the ground, for example, could map its xy plane to a portion of the global xy plane. Similarly, if the two dimensional physics engine was located along a wall, its xy plane would correspond to a portion of the xz plane of the global space. Therefore, after a physics engine has updated an entity in its view of the space, a geometrical transformation is applied to transform the positional information into the representation of the global space.

The soundness of the status of the global space is guaranteed by a number of design choices:

1. ARGoS is a discrete-time simulator, meaning that the status of the global space is updated synchronously by steps of τ simulated milliseconds. At each time step, all physics engines are called to integrate the status of the entities by τ. In this way, the global time is kept synchronized and the status of all the entities evolves at the same speed, even though physics engines are executed in parallel threads.

2. Mobile entities can belong to only one physics engine at a time. An entity is mobile when its position and orientation can change over time as a result of an active choice (i.e., a robot and its controller) or as a result of the physical interaction with other entities (i.e., an object being pulled or pushed).

3. Mobile entities in different physics engines do not physically interact with each other.

4. Immobile entities can belong to any number of physics engines. In this way, walls and obstacles can be shared, leading to a consistent view of the environment by all the engines.

Robot controllers interact with the environment through simulated sensors and actuators. Sensors read from the state of the global space. It is for this reason that communication and sensing can happen independently of how the

Algorithm 1 The simplified main loop of the ARGoS simulator.

```
1:  while experiment is not finished do
2:    for all robots do
3:      Update sensor readings      }  read from global space
4:      Execute control step        }
5:    end for
6:    for all robots do             }
7:      Update robot status         }  write into global space
8:    end for                       }
9:    for all physics engines do    }
10:     Update physics              }  write into global space
11:   end for                       }
12: end while
```

entities are partitioned into physics engines. Actuators update a robot's state. For instance, a robot's LED actuator updates the portion of the robot state that stores the current LED colors.

Visualizations access the state of the global space and output a representation of it for the user. At the time of writing, the only interactive visualization available is based on Qt4[17] and OpenGL and allows the user to manage the simulated environment in an intuitive way. Two additional non-interactive visualizations are present: *(i)* a high quality visualization based on the well known ray-tracing software POV-Ray[18] and *(ii)* a text-based visualization mainly designed for interaction with scripts and plotting programs such as Gnuplot[19].**TODO: show screenshots**

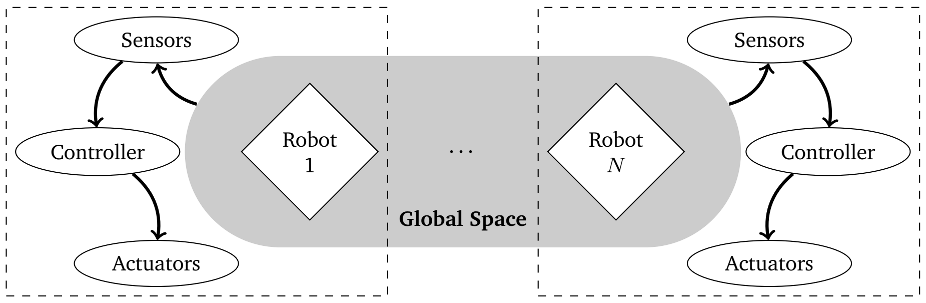
4.2 Multi-Thread Architecture

The architecture of ARGoS was designed to be inherently multi-thread, to exploit more efficiently the resources of modern multi-core CPUs, even though the libraries used in the plug-ins are not multi-thread. Multi-threading is embedded in the simulation loop. To understand how it works, we start by introducing the main concepts on information flow in the simulation loop. Subsequently, we explain how the computation is partitioned into multiple threads.

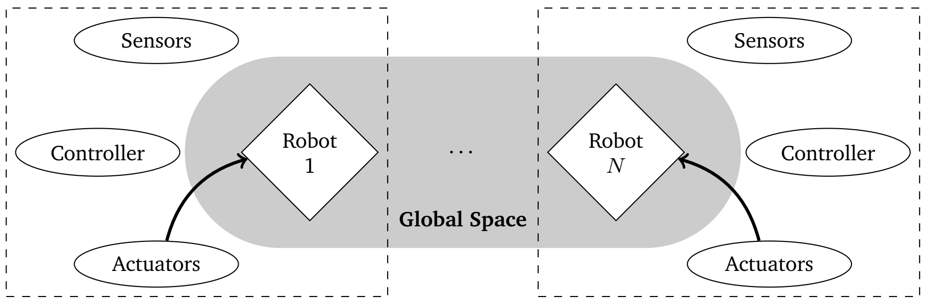
During the execution of the simulation loop, multiple threads access the global space for reading or writing. Thus, the global space is a shared resource on which, in principle, race conditions may occur. Solving race conditions with semaphores, though, is not optimal because of the high performance costs involved. In the main loop of ARGoS, therefore, we removed race conditions altogether. As it can be seen from the pseudo-code reported in Algorithm 1, access to the global space for reading and writing happens in different moments. A diagrammatic representation of the simulation loop is also reported in Figure 2.

For each cycle of the simulation, the first phase is read-only (lines 2–5 and Figure 2(a)). In this phase, all sensor readings are stored in the robot sensor objects. Then, the controllers read from the sensors to select the actions to per-

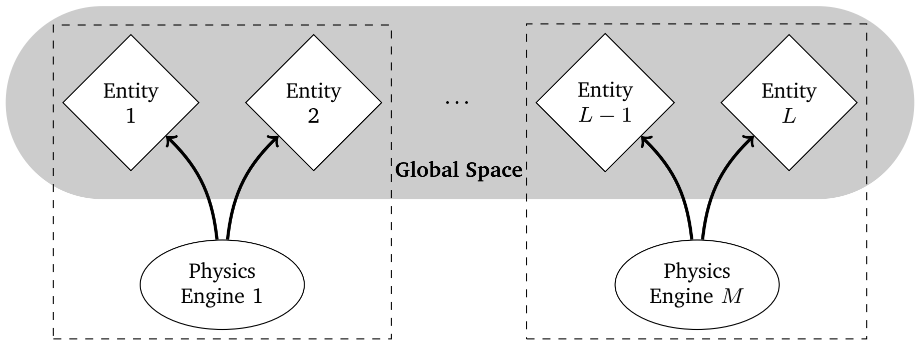
[17]`http://doc.qt.nokia.com/`
[18]`http://www.povray.org/`
[19]`http://www.gnuplot.info/`



(a) Sensor update and control step.



(b) Robot state update, excluding physics-related information.



(c) Update of the physics-related entity information.

Figure 2: Information flow in the various phases of the main simulation loop of ARGoS. The robot entities live in the global space. A controller and a set of sensors and actuators are associated to each robot. (a) In the initial phase, robot sensors collect information from the global space. Subsequently, robot controllers query the sensors and update the actuators with the chosen actions to perform. (b) The chosen actions stored in the actuators are executed, that is, the robot state is updated. At this point, positions and orientations have not been updated yet. (c) The physics engines calculate new positions and orientations for the mobile entities under their responsibility. Collisions are solved where necessary.

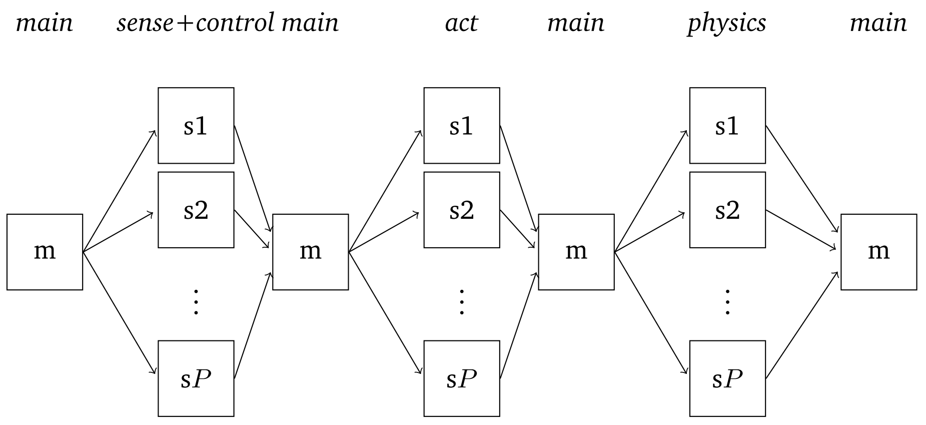


Figure 3: The multi-threading schema of ARGoS is scatter-gather. The master thread, marked with 'm', coordinates the activity of the slave threads, marked with 's'. The *sense+control*, *act* and *physics* phases are performed by P parallel threads. P is defined by the user.

form. The actions are stored in the actuator objects, but they are not executed yet. Being read-only, this phase excludes race conditions.

In the next two phases, the actions stored in the actuator objects are executed. First, the actuators update the portion of the state of the robots that does not involve physics calculations (lines 6–8). For instance, message exchanging happens in this phase. As depicted in Figure 2(b), each actuator is linked to a single robot entity. Therefore, even though actuators are updated in different threads, race conditions are not possible. At the end of this phase, positions and orientations of entities have not been updated yet.

It is the purpose of the last phase (lines 9–11 and Figure 2(c)) to perform such activity. Physics engines are called in parallel to update the entities under their control. Each physics engine first calculates new positions and orientations in its internal representation of the space, solving collisions where needed. Subsequently, physics engines store the updated information in the global space. Recalling the discussion in Section 4.1, race conditions are avoided because of design choices #2 (mobile entities can belong to only one physics engine at a time) and #3 (mobile entities in different physics engines do not physically interact with each other).

Multi-threading follows a scatter-gather paradigm. The three phases that form the simulation loop are coordinated by a main thread, marked with 'm' in Figure 3. At the beginning of the experiment, the main thread creates P slave threads to execute the different phases of the simulation loop. Each of these threads is initially idle, awaiting a signal from the main thread to proceed. The number P of threads to create is set by the user as part of the configuration of the experiment. The lifetime of these threads overlaps that of the main thread, to avoid wasting time destroying and recreating them at each iteration of the simulation loop. The slave threads are marked with 's' in Figure 3.

To execute the first phase of the simulation loop, the main thread sends the *start* signal to the P slave threads. As a consequence, the slave threads first calculate the portion of robot entities to update, and then execute lines 2–5 of Algorithm 1 over such portion. At the end of the computation, each thread sends the *done* signal to the main thread and switches to idle state.

After all the slave threads have sent the *done* signal, the main thread sends back another *start* signal to trigger the second phase of the simulation loop. Similarly to the previous phase, the slave threads execute lines 6–8 of Algorithm 1 over their portion of robot entities to update, send the *done* signal to the main thread, and finally switch to idle state.

The final phase, physics update, is executed in an analogous fashion. The main thread sends them the *start* signal, and each physics engine executes lines 9–11 of Algorithm 1 over the set of mobile entities under its responsibility. Once more, the *done* signal informs the main thread of the end of the computation by a slave thread.

As the experimental evaluation in Section 5 shows, the usage of threads improves the performance of the ARGoS simulator. A notable aspect of the architecture of ARGoS is that the performance advantage given by multi-threading does not entail an increase in complexity for plug-in developers. In fact, due to the way information flows in the simulation loop, plug-ins do not need to synchronize or cope with resource conflicts. As a consequence, libraries that do not support multi-threading can be used inside a plug-in without unpredictable side effects.

4.3 Handling Spatial Data

To perform their computation, sensors and actuators need to access the information present in the global space. Most of the communication devices, as well as sensors like cameras, infrared and laser devices, are typically simulated by casting rays and checking for intersections with nearby objects. Clearly, to perform these calculations, it is necessary to have access to the geometrical shape of the body of an object.

In traditional simulator designs, such as those of Stage and Webots, this is not a problem, since the simulated space contains both the state of the objects and the information about their bodies. On the contrary, in ARGoS the shape of an object is stored inside the physics engines, not in the global space. This design choice could potentially pose a crucial issue: since the physics engines to use are set by the user, sensor and actuator developers cannot foresee which engines will be selected. Moreover, thousands of ray cast operations are usually performed even in a simple simulation, making it one of the most expensive activities, if not the most. Optimizing these operations has dramatic impact on the overall performance.

To solve these issues, we designed the interfaces of the global space and of the physics engines to interoperate in an efficient manner and let each entity in the global space store its 3D bounding box.

The query is split in two phases. First, the global space builds a list of candidate entities whose bounding box intersects the given ray. This operation is performed very efficiently. Since entities are composable (see Section 3.1.2), each entity that occupies a portion of the global space has a component called *embodied entity* that stores its position, orientation and bounding box. In this way, the code that checks for intersections is general and fast, as it needs only to go through the embodied components of an entity. No expensive type conversions are necessary. Furthermore, the list of candidate entities is built with logarithmic complexity, thanks to the fact that embodied entities are indexed in a space hash [23].

In the second phase of the ray intersection query, the global space delegates each ray-candidate intersection computation to the physics engines. To make this possible in an efficient manner, the interface of a physics engine has been divided in two parts: *(1)* an interface to interact with the engine itself and its local view of the global space and *(2)* an interface that defines the properties of and interactions with the individual entities in the local view of the space. We refer to this type of entity as *physics entity*. Each embodied entity in the global space is associated to a physics entity.[20] The physics entity interface offers a method to calculate the intersection of a ray with its body, and returns the position of the closest intersection point (if any) to the starting point of the ray in the global space. Therefore, to complete the query, the global space needs only to loop through the candidate entities and call that method for each of them.

When a ray intersection check is due, from the point of view of a sensor/actuator developer it is enough to query the global space providing the starting and ending points of the ray, without the need to know which engine will be chosen by the user. Also, since the global space filters the candidate entities efficiently, it is not necessary for physics engines to offer optimized ray cast operations—only a simple ray-shape intersection routine is necessary. Therefore, even simple, non scalable engines have little impact on performance.

5 Experimental Evaluation

In the following, we study the performance of the ARGoS simulator for different configurations.

To date, there is little work in assessing the performance of general purpose simulators for numbers of agents comparable to interesting swarm robotics scenarios ($\sim 1,000$). 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 agents is Stage [24]. In that work, Vaughan studies Stage's performance in a very simple experiment in which robots diffuse 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. The experiments show the effect of increasing the number of agents and the size of the environment on the time to complete a simulation.

For our evaluation, we employ an experimental setup similar to Vaughan's. Figure 4 depicts a screenshot of the arena where the robots disperse. The arena is a square whose side is 40 m long. The space is structured into a set of connected rooms that loosely mimic the layout of a real indoor scenario.

[20]Immobile entities can be associated to multiple physics entities, but mobile entities are always associated to only one.

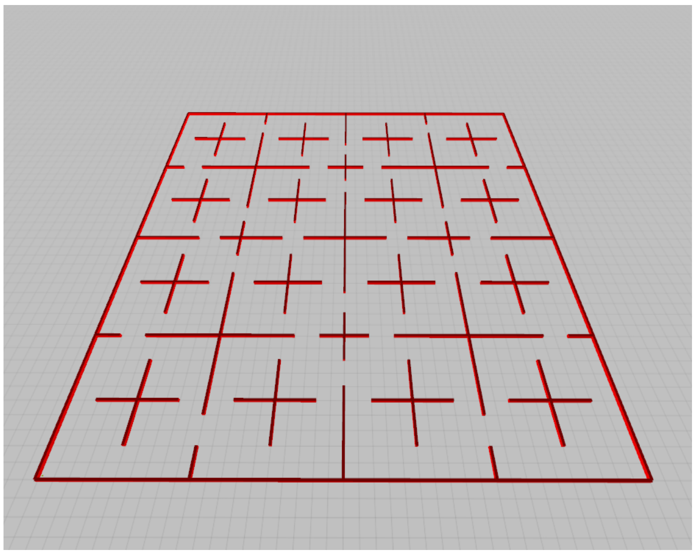


Figure 4: A screenshot from ARGoS showing the simulated arena created for experimental evaluation.

Analogously to Stage's evaluation, which was performed with a basic robot model, in our experiments we used the simplest robot available in ARGoS: the e-puck [19]. Each robot executes the diffusion routine reported in Algorithm 2.

To keep the evaluation meaningful with respect to typical use cases, we run all the experiments with full 2D dynamics, including collision checking and complete force calculations. The employed engine is based on Chipmunk, a simple and fast 2D physics library largely used in games and physics-based simulations. In comparison, Stage's engine is based on 2D kinematic calculations.

We employ as performance measures two classic quantities:

Wall clock time (w): it corresponds to the real elapsed time between application start and end, or, equivalently, run-time length as experienced by the user. Multi-tasking operating systems switch between processes continuously, so usually in a second of perceived computation many applications obtain a time slice. Therefore, this measure is usually noisy because it includes also the time spent running the other active applications. To hinder unwanted interference, 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.

CPU usage (u): it is a derived measure. To calculate it, first we measure the total CPU time c obtained by the process running the experiment. The difference between w and c is that the latter is increased only when the process is actively using the CPU. In multi-thread systems, c is calculated as the sum of the active time slices obtained by the process on each core c_i:

$$c = \sum_i c_i.$$

Algorithm 2 The basic diffusion algorithm executed by the robots in the experiments for performance evaluation. Proximity sensor readings are 2D vectors whose length is 0 when nothing is sensed and 1 when the sensed object is touching the sensing robot.

```
 1: Read the $N$ proximity sensor readings as 2D vectors $\vec{r}_i$
 2: $\vec{v} \leftarrow \vec{0}$
 3: **for all** readings $\vec{r}_i, i \in [1, N]$ **do**
 4:     $\vec{v} \leftarrow \vec{v} + \vec{r}_i$
 5: **end for**
 6: $\vec{v} \leftarrow \vec{v}/N$
 7: **if** angle($\vec{v}$) $\in$ 'go straight' range **then**
 8:     Go straight
 9: **else**
10:     **if** angle($\vec{v}$) $> 0$ **then**
11:         Turn right
12:     **else**
13:         Turn left
14:     **end if**
15: **end if**
```

Therefore, c does not account for parallelism—spending 20 active seconds on core 1 and 10 on core 2 with perfect parallelism results in $c = 30$ s, but the user would experience $w \approx 20$ s. CPU usage is defined as

$$u = \frac{c}{w}.$$

In single-core CPUs or in single-thread applications, $u \leq 1$. With multi-thread applications on multi-core CPUs, the aim is to obtain $u > 1$. Denoting with K the number of available cores, we have perfect parallelism when $u = K$. Perfect parallelism corresponds to the maximum CPU usage possible.

Experiments aim to assess the effect on w and u of different configurations of the described experiment. In particular, we identify three factors that strongly influence performance: *(a)* the number of e-pucks N, *(b)* the number of parallel slave threads P, and *(c)* the way the environment is partitioned into multiple physics engines.

Concerning the number of e-pucks, we run experiments with $N \in [1, 10^5]$. As for the number of slave threads P, it is a well known result that maximum performance is obtained when $P = K$ [11]—when the threads are less than the cores, not all the cores are used, lowering u; when the threads are more than the cores, the operating system must spend time switching between the threads, decreasing c and increasing w, which results again in a lower u. To test the effect of the number of threads P, we run our experiments on a machine with 16 cores[21], and let $P \in \{0, 2, 4, 8, 16\}$. Finally, we defined five ways to partition the environment among multiple physics engines, each differing from another in how many engines are used and how they are distributed. We refer

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

to a partitioning with the symbol F_E, where E is the number of physics engines employed. The partitionings are depicted in Figure 5. For each experimental setting $< N, P, F_E >$, we run 40 trials.

Each trial simulates $T = 60$ s of virtual time. In the first stages of the analysis of the data, we observed that the initialization of the experiment was polluting the measures of w and u significantly, making comparisons difficult. For low numbers of e-pucks (1 to 100) most of the initialization time was due to XML parsing and the creation of the plug-in objects. The time in the actual simulation loop was dominated by this phase, contributing to w only marginally. On the other hand, for numbers of e-pucks above 1,000, the most expensive initialization activity was placing the robots in the environment, which is performed by the main thread. This was significantly lowering the measures of u. Therefore, to ensure meaningful comparisons in the entire range of N we considered, the measures of wall clock time and CPU usage were taken only inside the main simulation loop, discarding initialization time.

Results are reported in Figures 6 and 7. The first result that we would like to highlight is the fact that some configurations reach real-time with swarm sizes of 10,000 robots. This is a remarkable result, given that it has been obtained in experiments with complete dynamics calculations.

The effect of increasing threads is positive on wall clock time. Parallelism indeed lowers computation times, even when only one physics engine is employed. Partitioning the space into multiple physics engines has two positive effects: it not only lowers wall clock time, but it also increases CPU usage. The expected result that maximum performance is reached when the number of threads is equal to the number of core is matched by the showed graphs. Even though wall clock time changes little when 8 or 16 threads are used, it is evident from Figure 7 that CPU resources are exploited much better.

TODO: Elaborate more

6 Vision: Millions of Robots in Real-Time

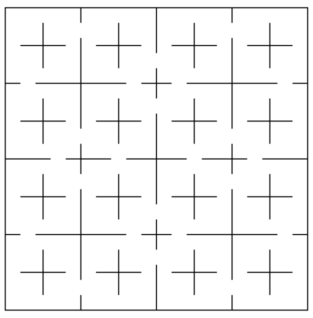
The results shown in Section 5 show that ARGoS can simulate 10,000 simple wheeled robots in real-time. An interesting research question is how to reach the next level—a real-time simulation of 100,000 or one million entities.

We believe that a possible way to approach this issue involves a better distribution of the computation. In the rest of this section, we discuss two promising improvements with respect to the distributed architecture currently implemented in ARGoS.

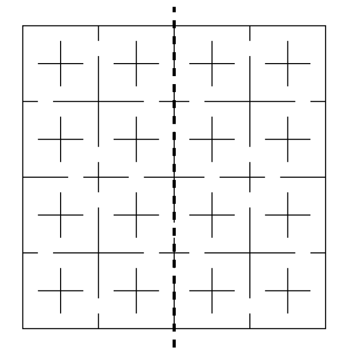
6.1 More Sophisticated Threading Models

The scatter-gather paradigm currently implemented in the architecture of the ARGoS simulator was chosen for its simplicity. However, we are currently working on an enhancement that involves substituting scatter-gather with the *h-dispatch* paradigm [11]. The advantages of the latter paradigm are that *(i)* it opens the way for better computational load balancing among the slave threads and *(ii)* it supports in a natural way computation on GPGPU hardware.

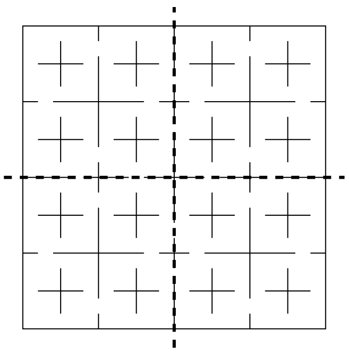
It is a well known result [13] that the use of GPGPU hardware can greatly improve the performance of large loops under specific assumptions on the op-



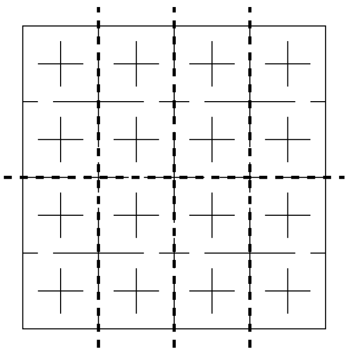
(a) $E = 1$ engine (F_1).



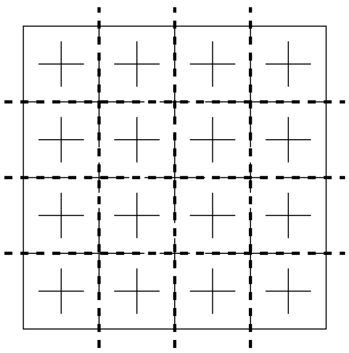
(b) $E = 2$ engines (F_2).



(c) $E = 4$ engines (F_4).

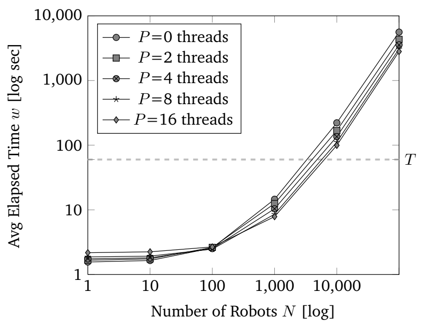


(d) $E = 8$ engines (F_8).

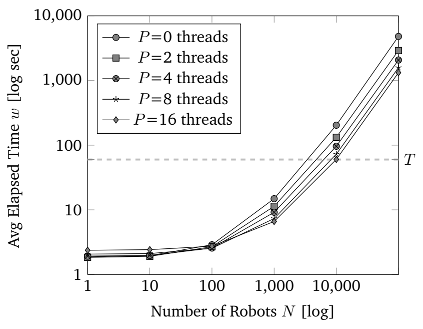


(e) $E = 16$ engines (F_{16}).

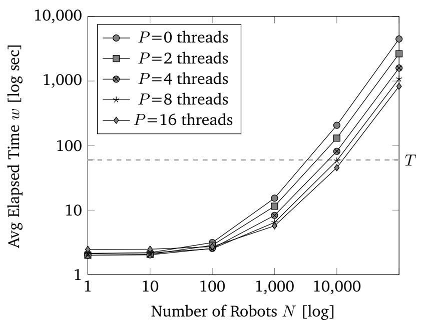
Figure 5: The different space partitionings (F_1 to F_{16}) of the arena used to evaluate ARGoS' performance.



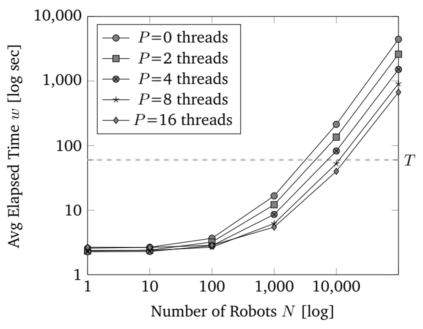
(a) $E = 1$ engine (F_1).



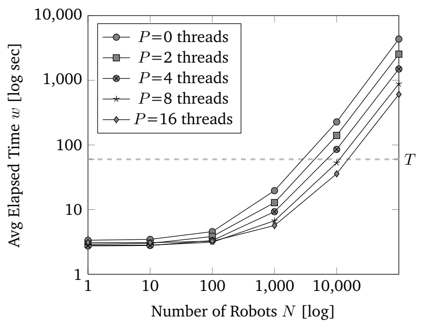
(b) $E = 2$ engines (F_2).



(c) $E = 4$ engines (F_4).

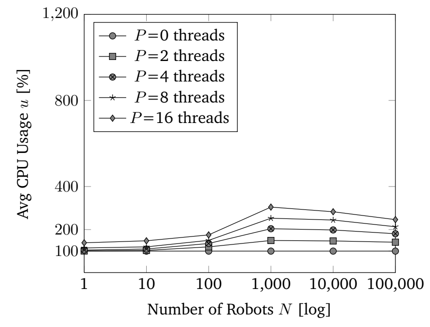


(d) $E = 8$ engines (F_8).

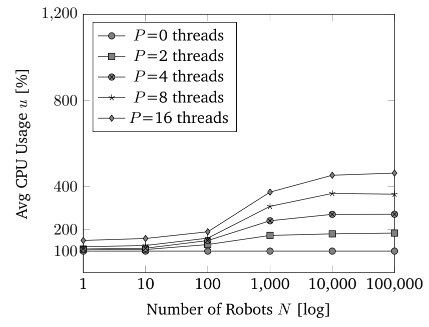


(e) $E = 16$ engines (F_{16}).

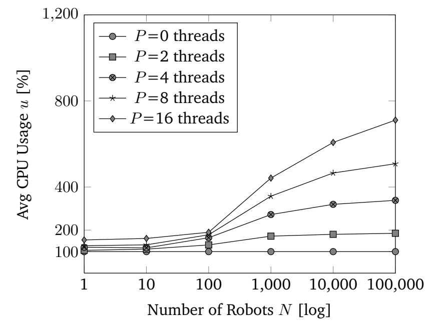
Figure 6: Average wall clock times for different experimental settings. Each point corresponds to a set of 40 trials with a specific configuration $< N, P, F_E >$. Points under the dashed line mean that the simulations were faster than real time; above it, they were slower. The standard deviation is omitted because its value is so small that it would be invisible in the graph.



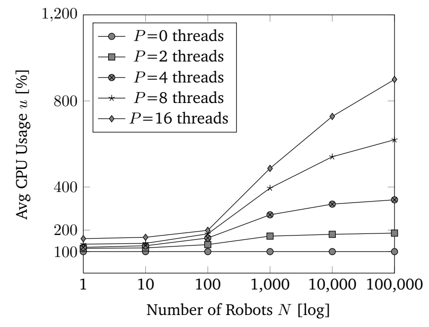
(a) $E = 1$ engine (F_1).



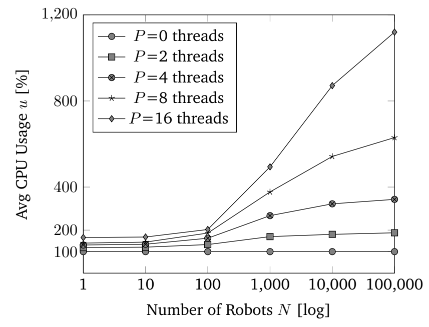
(b) $E = 2$ engines (F_2).



(c) $E = 4$ engines (F_4).



(d) $E = 8$ engines (F_8).



(e) $E = 16$ engines (F_{16}).

Figure 7: Average CPU usage for different experimental settings. Each point corresponds to a set of 40 trials with a specific configuration $< N, P, F_E >$. The standard deviation is omitted because its value is so small that it would be invisible in the graph.

erations submitted to the GPU, such as the absence of branches [20]. A heterogenenous threading model comprising threads on CPU and on GPU may result in a sizable improvement in performance. GPGPU computation may be useful in lightening the main CPU from calculating the millions of ray cast operations needed by most of the simulated sensors and communication devices of the robots.

6.2 Multi-Processor Architecture

In ARGoS, computation is distributed only among the cores of a single CPU. However, distribution could happen also across multiple CPUs.

The most successful massively multi-player online games (MMOG) can sustain the simultaneous connection of millions of users playing in real-time. These architectures exploit the fact that each user runs locally a thick client that takes care of most of the calculations related to its neighborhood, partially exchanging the results with other users whose avatars are nearby in the virtual world. These systems often retain a certain deal of centralization, requiring that some of the exchanged information pass through the main servers, because such games must prove robust to cheating [2].

MMOG architectures can be a source of inspiration to reach the goal of millions of entities simulated in real-time [15, 22]. There are two complementary ways to distribute a simulation among multiple CPUs: *(i)* employing ARGoS' spatial decomposition, in which physics engines become thick clients that update in parallel the space; and *(ii)* making the space itself distributed, for example implementing each entity as a node in a peer-to-peer or publish-subscribe system.

It is our intention to explore these research directions and study their effectiveness over performance.

7 Conclusions and Future Work

In this paper we introduced ARGoS, a simulator specifically designed for large heterogeneous swarms of robots. With respect to existing simulators, ARGoS offers a more flexible 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 as new plug-ins, promoting exchange and cooperation among researchers.

A unique feature of design 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 the latter can be optimized for the experiment at hand. Robots can migrate from a physics engine to another transparently.

In addition, the multi-thread architecture of ARGoS, despite its simplicity, proves very scalable, showing low run-times while increasing the exploitation of the multiple cores of modern CPU architectures. Results in Section 5 show that ARGoS can simulate 10,000 robots in real-time, using multiple physics engines in 2D with full dynamics. It is important to notice that the development of new plug-ins is not influenced by the presence of multiple running threads—shared resources and reading/writing phases were designed to avoid race conditions.

Future work will be mainly devoted to support new robot types and to further increase scalability. The simplest way to increase performance is to optimize code. So far, systematic code profiling has not been our main concern, but it is likely to improve the overall performance to some extent.

The most interesting research directions, in our opinion, involve reaching real-time performance for swarms composed of millions of entities. Clearly, it is not possible to achieve these figures by mere code optimization. More sophisticated architectures, such as those mentioned in Section 6, may be viable solutions: *(i)* employing a heterogeneous threading model performing the computation both on CPU and on GPU and *(ii)* modifying the multi-thread architecture of ARGoS into a mixed multi-thread and multi-process architecture, in which multiple physics engines and the global space are distributed across different machines.

References

[1] Bachrach, J., Beal, J., and McLurkin, J. Composable continuous-space programs for robotic swarms. *Neural Computation & Applications 19* (2010), 825–847.

[2] Bezerra, C., Cecin, F., and Geyer, C. A3: A novel interest management algorithm for distributed simulations of MMOGs. In *12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*. IEEE Press, Piscataway, NJ, 2008, pp. 35–42.

[3] Bonani, M., Longchamp, V., Magnenat, S., Rétornaz, P., Burnier, D., Roulet, G., Vaussard, F., Bleuler, H., and Mondada, F. The marXbot, 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)*. IEEE Press, Piscataway, NJ, 2010, pp. xxx–yyy.

[4] Bonani, M., Magnenat, S., Rétornaz, P., and Mondada, F. 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)*, vol. 5928 of *Lecture Notes in Computer Science*. Springer, Berlin, Germany, 2009, pp. 11–22.

[5] Browning, B., and Tryzelaar, E. ÜberSim: a multi-robot simulator for robot soccer. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. ACM, New York, NY, 2003, pp. 948–949.

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

[7] Cohen, M. A., Ritter, F. E., and Haynes, S. R. Applying software engineering to agent development. *AI Magazine 31*, 2 (2010), 25–44.

[8] Collett, T. H., MacDonald, B. A., and Gerkey, B. P. Player 2.0: Toward a practical robot programming framework. In *Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*. Proceedings on CD-ROM, 2005.

[9] Ferrante, E. A control architecture for a heterogeneous swarm of robots: The design of a modular behavior-based architecture. Tech. Rep. TR/IRIDIA/2009-010, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, 2009.

[10] Gerkey, B. P., Vaughan, R. T., and Howard, A. The Player/Stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR)*. IEEE Press, Piscataway, NJ, 2003, pp. 317–323.

[11] Holmes, D. W., Williams, J. R., and Tilke, P. An events based algorithm for distributing concurrent tasks on multi-core architectures. *Computer Physics Communications 181*, 2 (February 2010), 341–354.

[12] Joselli, M., Clua, E., Montenegro, A., Conci, A., and Pagliosa, P. A new physics engine with automatic process distribution between CPU-GPU. In *Proceedings of the ACM SIGGRAPH symposium on video games (Sandbox'08)*. ACM, New York, NY, 2008, pp. 149–156.

[13] Kerr, A., Diamos, G., and Yalamanchili, S. Modeling GPU-CPU workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU'10)*. ACM, New York, NY, 2010, pp. 31–42.

[14] Klein, J., and Spector, L. 3D multi-agent simulations in the breve simulation environment. In *Artificial Life Models in Software*, A. Adamatzky and M. Komosinski, Eds., 2nd ed. Springer, Berlin, Germany, 2009, pp. 79–106.

[15] Knutsson, B., Lu, H., Xu, W., and Hopkins, B. Peer-to-peer support for massively multiplayer games. In *Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004)*. IEEE Press, Piscataway, NJ, 2004, pp. 96–107.

[16] Koenig, N., and Howard, A. 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)*. IEEE Press, Piscataway, NJ, 2004, pp. 2149–2154.

[17] Kramer, and Schultz. Development environments for autonomous mobile robots: a survey. *Autonomous Robots 22*, 2 (2007), 101–132.

[18] Michel, O. Cyberbotics Ltd. – Webots: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems 1*, 1 (March 2004), 39–42.

[19] Mondada, F., Bonani, M., Raemy, X., Pugh, J., Cianci, C., Klaptocz, A., Magnenat, S., Zufferey, J.-C., Floreano, D., and Martinoli, A.

The e-puck, a robot designed for education in engineering. In *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, vol. 1. IPCB, Castelo Branco, Portugal, 2009, pp. 59–65.

[20] OWENS, J., HOUSTON, M., LUEBKE, D., GREEN, S., STONE, J., AND PHILLIPS, J. GPU computing. *Proceedings of the IEEE 96*, 5 (May 2008), 879–899.

[21] ROBERTS, J., STIRLING, T., ZUFFEREY, J., AND FLOREANO, D. Quadrotor using minimal sensing for autonomous indoor flight. In *European Micro Air Vehicle Conference and Flight Competition (EMAV)*. Proceedings on CD-ROM, 2007.

[22] SÜSELBECK, R., SCHIELE, G., AND BECKER, C. Peer-to-peer support for low-latency massively multiplayer online games in the cloud. In *8th Annual Workshop on Network and Systems Support for Games (NetGames)*. IEEE Press, Piscataway, NJ, 2009, pp. 1–2.

[23] TESCHNER, M., HEIDELBERGER, B., MUELLER, M., POMERANETS, D., AND GROSS, M. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of the Vision, Modeling, and Visualization Conference*. Aka GmbH, Heidelberg, Germany, 2003, pp. 47–54.

[24] VAUGHAN, R. Massively multi-robot simulation in stage. *Swarm Intelligence*, 2 (2008), 189–208.