AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTRONIKI, INFORMATYKI I TELEKOMUNIKACJI

KATEDRA INFORMATYKI

Rozprawa doktorska

Concurrent Execution Models for Agent-Based Computing Systems

Author:	*Daniel Krzywicki*
Degree programme:	*Informatyka*
Main Supervisor:	*dr hab. inż. Marek Kisiel-Dorohinicki, prof. AGH*
Second Supervisor:	*dr inż. Roman Dębski*

Kraków, 2018

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Acknowledgments

I would like to express my very great appreciation to my main supervisor, dr hab. inż. Marek Kisiel-Dorohinicki, prof. AGH, for the inspiration to explore scientific research and his invaluable help in guiding my scientific efforts throughout all my doctoral years. Thank you for your knowledge, patience, and constructive suggestions during the planning and development of this research work

I am particularly grateful for the assistance given by my second supervisor, dr inż. Roman Dębski, without whom this doctoral dissertation could not have been written. Thank you for your willingness to give your time so generously, for your encouragement and constructive recommendations.

My special gratitude goes to dr hab. inż. Aleksander Byrski, prof. AGH for his professional guidance and valuable support.

I also wish to thank the employees of the Computer Science Department at AGH, especially dr. inż. Wojciech Turek, dr inż. Kamil Piętak and mgr inż. Łukasz Faber for their support, kindness and successful cooperation.

I direct separate, equally important thanks to my wife Katarzyna, for her constant support and encouragement throughout my study. Finally, I wish to thank my parents for awakening my curiosity for science and supporting me.

English Summary

Multi-Agent Systems (MAS) are a bottom-up approach to the modeling of complex problems. Instead of explicitly defining the behavior of a complex system as a whole, one only describes the behavior of the simple constituents of the system. In this approach, complex system behaviors are said to *emerge* as a result of the interactions between these basic components.

The following dissertation concerns computationally demanding multi-agent systems, in which the number of agents is very large and the interactions between them are intensive or non-trivial. Interactions are intensive when they occur very often or when many calculations are needed to determine their outcomes. They are non-trivial when it is not possible to predict beforehand and in a general way which agents will interact with each other.

The main subject of this research is the concurrency of such interactions between agents, understood as the way agents perceive their interactions and the effects of these. The manner in which interactions between agents are organized is referred to in this dissertation as their *concurrent execution model*. Various concurrent execution models are possible and differ both in the way information propagates in the system (i.e. causality as it is observed by the agents), as well as in their technical characteristics, such as their amenability to parallelization and their scalability. The aim of this research is to investigate how different concurrent execution models affect the properties of an agent-based computation, both in terms of the behavior of the algorithm and of the efficiency of the computation.

This research is focused on Evolutionary Multi-Agent Systems (EMAS) as an example of such computationally intensive multi-agent systems. EMAS consist in combining multi-agent systems and evolutionary algorithms in order to solve difficult optimization problems. In contrast to classical evolutionary algorithms, selective pressure, which is one of the main mechanisms of evolution, is not centrally enforced, but instead is an emergent property resulting from the interaction of independent agents.

EMAS are interesting in the context of this research, because the organization of agents' interactions has a significant impact on the behavior of the algorithm. For example, one possible organization is to select at each step one random pair of agents for them to compete or reproduce. An alternative approach could be to separate the entire population into pairs of agents, and, after an independent execution of the interactions of each pair, merge the resulting agents into a new population. Depending on the choice, the dynamics of evolution will be different. In the first case, the outcome of the interaction is visible immediately to all other agents, so the diffusion of genes in the population will be faster. However, depending on the way pairs of agents are selected, some agents may have more opportunities to reproduce, which

introduces additional, uncontrolled selective pressure. In the second case, the population changes more uniformly; at each step, each agent has one opportunity to reproduce. However, it must wait for the others before being able to make another interaction, so the overall throughput of the simulation can be lower.

In general, it is difficult to predict which variant will be more effective in solving a particular optimization problem. However, most of the available software frameworks used to implement agent systems impose a specific concurrent execution model. The semantics of the multi-agent system, i.e. the decomposition into basic constituents and the definition of their elementary behavior, must be expressed within this model and is closely coupled to it. In order to be able to compare different concurrent execution models for the same algorithm, I introduce in this dissertation a formalism which makes it possible to separate the semantics of an agent-based algorithm from its execution model. This formalism consists in defining the semantics of the agent algorithm with the help of two functions: a *behavior function* specifying agent behavior based on their state, and a *meetings function* describing the interaction between agents with similar behavior. The execution model then comes down to the way these functions are applied on a set of agents.

As part of this dissertation, I propose and analyze several concurrent execution models based on this formalism:

– The first model corresponds to a typical, iterative implementation of an agent-based computation: in every step, the population is divided into groups with similar behavior using the behavior function. Then, each group is transformed individually using the meeting function. Agents appearing as a result of the meetings of all the groups are combined into a new population.

– The second execution model is based on the actor model of concurrency. In simplified terms, an actor can be understood as a process which requires few resources, can execute independently and potentially in parallel with other actors, and communicates with them only by exchanging messages (and not by modifying some shared state). In this model, a separate actor corresponds to every agent, every possible behavior, and every interaction. Within their actors, agents cyclically determine their behavior using the behavior function and send a message to the actor representing this behavior. That actor mediates in matching the agents into pairs and creates a new actor representing the interaction. The interaction actor applies the meetings function. As a result of the interaction, a message is sent to the actors representing the participating agents in order to update their state, and new agents/actors can be created or existing ones deleted.

– The third model is based on parallel *Algorithmic Skeletons* In short, this approach consists in a static analysis of a program in order to find patterns that can be replaced with an alternative, more effective implementation, without changing the behavior of the program. For example, consider a function that takes an array as an input, and transforms each element independently to create an output array. An implementation that transforms each element in turn can be converted into an implementation that performs all the transformations in parallel on multiple processors, without changing the behavior of the function itself. The implementation of the third model is thus the

effect of applying this type of program transformation in the first model in order to introduce parallel processing where possible. The overall structure remains similar, however.

– The fourth execution model is an application of *reactive streams*. Such streams consist in a graph of stages through which data flows. Each stage transforms subsequent input elements and sends them to the next stages. Reactive streams are characterized by the fact that the stages are able to coordinate on the rate of production and consumption of elements, i.e. the throughput may be different in various parts of the stream. In practice, this means that within each stage, one input element may result in zero, one, or many output elements being emitted downstream. In this execution model, the agents correspond to elements flowing in a closed, recursive stream (that is, output elements are passed back to the input). Depending on the behavior function, the flows of agents are split into substreams corresponding to specific behaviors. Due to the variable throughput available in a reactive stream, the flow of agents in each sub-stream can be locally compressed, which allows them to interact by applying the meetings function on groups of subsequent agents. A solution within this model enables to change the order of elements in the stream, which allows for controlling the concurrency of interactions as observed by the agents.

All above concurrent execution models are subjected to experimental evaluation by applying them to an optimization problem. One of the main metrics recorded is the quality of the best solution found depending on the number of interactions, i.e. the effectiveness of the algorithm. This metric allows for differentiating between models, regardless of the number or type of processors used in the calculation. In this respect, the actor-based and the stream-based models achieve significantly better results than the others. In addition, the experiments show that the right choice of parameters in the stream-based execution model can recreate the characteristics of the other models - in other words, it is a more general solution, as it can simulate other models

To sum up, I show in this research that the concurrent execution model of agent interactions can be decoupled from the semantics of the algorithm itself. This makes it possible to meaningfully compare alternative execution models for the same algorithm and, potentially, make an informed choice to best match a specific hardware architecture or problem size. Among the models being considered, the one based on reactive streams proves to be the most promising, both in terms of efficiency and functionality. The effects of the research presented in this dissertation could help to improve the existing software used for agent-based computing and, consequently, allow the modeling of more complex problems.

Streszczenie w języku polskim

Systemy wieloagentowe (ang. Multi-Agent Systems, MAS) są oddolnym podejściem do modelowania złożonych problemów. Zamiast definiować wprost złożone zachowania systemu jako całość określa się jedynie proste zachowania jego elementarnych składowych. W podejściu tym, złożone zachowania systemu pojawiają się w sposób zwany *emergentnym*, w wyniku oddziaływań między jego składowymi.

Tematyka niniejszej rozprawy dotyczy wymagających obliczeniowo systemów agentowych, w których liczba agentów jest bardzo duża, a oddziaływania między nimi są intensywne lub nietrywialne. Oddziaływania są intensywne, gdy następują bardzo często lub gdy do określenia ich skutków potrzebne jest wykonanie czasochłonnych obliczeń. Nietrywialne są zaś wtedy, gdy nie jest możliwe przewidzenie *a priori* i w sposób ogólny tego, którzy agenci będą wchodzić ze sobą w interakcje.

Głównym przedmiotem moich badań jest współbieżność oddziaływań między agentami rozumiana jako sposób, w jaki agenci postrzegają swoje oddziaływania oraz ich skutki. Sposób, w jaki zorganizowane są interakcje między agentami zwany jest w niniejszej pracy *modelem współbieżnego wykonania*. Możliwe są różne modele współbieżnego wykonania, różniące się zarówno sposobem propagacji informacji w systemie, czyli tym, jakie związki przyczynowo-skutkowe mogą być obserwowane przez agentów, jak również właściwościami technicznymi, takimi jak podatność na zrównoleglenie i skalowalność. Celem niniejszej pracy jest zbadanie, w jaki sposób różne modele współbieżnego wykonania wpływają na właściwości systemu, zarówno pod kątem zachowania algorytmu, jak i efektywności obliczeń.

Przykładem wymagających obliczeniowo systemów agentowych, na którym skupiam swoje badania, są Ewolucyjne Systemy Wieloagentowe (ang. Evolutionary Multi-Agent Systems, EMAS). Polegają one na połączeniu elementów systemów wieloagentowych i algorytmów ewolucyjnych w celu rozwiązywania trudnych problemów optymalizacyjnych. W przeciwieństwie do klasycznych algorytmów ewolucyjnych w EMAS presja selekcyjna, czyli jeden z głównych mechanizmów ewolucji, nie jest wprowadzona odgórnie, lecz pojawia się jako zjawisko emergentne wynikające z interakcji niezależnych agentów.

EMAS są interesujące w kontekście moich badań, gdyż organizacja interakcji agentów ma istotny wpływ na działanie algorytmu; na przykład, dynamika ewolucji będzie inna, gdy w każdym kroku jedna losowa para agentów będzie współzawodniczyć albo się rozmnażać, niż gdy cała populacja zostanie rozdzielona na pary w jednym kroku, a po niezależnym wykonaniu wszystkich interakcji agenci zostaną połączeni w nową populację. W pierwszym przypadku konsekwencje interakcji są widoczne natychmiast, więc dyfuzja genów w populacji będzie szybsza, ale w zależności od sposobu dobierania par,

niektórzy agenci mogą mieć więcej okazji na rozmnażanie, co wprowadza dodatkową, niekontrolowaną presję selekcyjną. W drugim przypadku populacja zmienia się bardziej jednolicie; w każdym kroku każdy agent ma jedną szansę na reprodukcję, jednak musi czekać na pozostałych przed podjęciem kolejnej interakcji.

Nie jest łatwo w sposób ogólny przewidzieć, który wariant będzie skuteczniejszy w rozwiązywaniu konkretnego problemu optymalizacyjnego. Większość dostępnych środowisk programistycznych służących do implementowania systemów agentowych narzuca jednak konkretny model współbieżnego wykonania. Semantyka systemu agentowego, czyli podział na elementy składowe i definicja ich elementarnych zachowań, musi być wyrażona w ramach tego modelu i jest ściśle z nim związana. Aby móc porównywać rożne modele współbieżnego wykonania dla tego samego algorytmu, w ramach niniejszej pracy wprowadzam formalizm pozwalający rozdzielić semantykę algorytmu agentowego od modelu wykonania interakcji agentów. Formalizm ten polega na zdefiniowaniu semantyki algorytmu agentowego przy pomocy dwóch funkcji: *funkcji zachowania*, określającej zachowanie agentów na podstawie ich stanu, oraz *fukcji spotkań*, opisującej interakcję między agentami o podobnym zachowaniu. Model wykonania sprowadza się wtedy do organizacji wywołań tych funkcji na zbiorze agentów.

W ramach pracy proponuję i analizuję kilka modeli współbieżnego wykonania:

– Pierwszy model odpowiada typowej, iteracyjnej implementacji obliczenia agentowego: w każdym kroku populacja dzielona jest na grupy o podobnym zachowaniu przy pomocy funkcji zachowań. Następnie, każda grupa z osobna jest przekształcana przy pomocy funkcji spotkań. Agenci pojawiający się w wyniku spotkań wszystkich grup łączeni są w nową populację.

– Drugi model wykonania oparty jest na aktorowym modelu współbieżności. W uproszczeniu, aktor może być rozumiany jako wymagający niewielu zasobów proces, który może działać niezależnie i potencjalnie równolegle z innymi aktorami i komunikuje się z nimi jedynie przez wymianę wiadomości (a nie poprzez modyfikowanie współdzielonego stanu). W tym modelu oddzielny aktor zostaje przypisany każdemu agentowi, każdemu możliwemu zachowaniu oraz każdej interakcji. W ramach swojego aktora, każdy agent cyklicznie określa swoje zachowanie przy pomocy funkcji zachowań i wysyła wiadomość do aktora przypisanego temu zachowaniu. Aktor ten pośredniczy w dobieraniu agentów w pary, tworząc nowego aktora reprezentującego interakcję. Aktor interakcji wywołuje funkcję spotkań. W konsekwencji spotkania, do aktorów odpowiadającym agentom wysyłana jest wiadomość zmieniająca ich stan, mogą też być tworzone nowe agenty/aktorzy lub usuwane istniejące.

– Trzeci model wykonania wykorzystuje tzw. równoległe szkielety algorytmiczne (ang. Algorithmic Skeletons). W skrócie, podejście to polega na statycznej analizie programu w celu znajdowania wzorców, które mogą być wymienione na alternatywną, bardziej efektywną implementację, bez zmiany zachowaniu programu. Na przykład, rozważmy funkcję, która przyjmuje na wejściu tablicę i przekształca każdy jej element niezależnie w celu stworzenia tablicy wyjściowej. Implementacja, która przekształca każdy element po kolei, może zostać zamieniona na implementację,

która dokonuje wszystkich przekształceń równolegle na wielu procesorach, bez zmiany zachowania samej funkcji. Trzeci model wykonania jest więc efektem zastosowania tego typu transformacji programów w modelu pierwszym w celu wprowadzenia równoległego przetwarzania tam, gdzie to możliwe. Zasadniczo struktura pozostaje więc podobna.

– Czwarty model jest zastosowaniem tzw. reaktywnych strumieni danych (ang Reactive Streams). Strumień taki składa się z grafu etapów, przez które przepływają dane. Każdy etap przekształca kolejne elementy wejściowe i przesyła je do dalszych etapów. Reaktywne strumienie charakteryzują się tym, że etapy potrafią dostosowywać między sobą rytm produkcji i konsumpcji kolejnych elementów, czyli przepustowość strumienia może się zmieniać. W praktyce oznacza to, że w każdym etapie jednemu elementowi wejściowemu może odpowiadać zero, jeden albo wiele elementów wyjściowych. W tym modelu agenci odpowiadają elementom przepływającym w zapętlonym strumieniu (czyli elementy wyjściowe przekazywane są z powrotem na wejście). W zależności od funkcji zachowania agenci kierowani są przez podstrumienie odpowiadające konkretnym zachowaniom. Dzięki zmiennej przepustowości możliwej w reaktywnym strumieniu, przepływ agentów w każdym podstrumieniu jest lokalnie kompresowany, co umożliwia ich interakcję i wywołanie funkcji spotkań na grupach sąsiednich agentów w strumieniu. Wprowadzone w tym modelu rozwiązanie pozwala z kolei zmieniać kolejność elementów w strumieniu, co umożliwia dokładną kontrolę nad współbieżnością obserwowaną przez agentów.

Wszystkie modele współbieżnego wykonania poddaję ocenie eksperymentalnej poprzez zastosowanie ich do obliczeń optymalizacyjnych. Jedną z głównych metryk, którą stosuję, jest jakość znalezionego rozwiązania w zależności od ilości interakcji, czyli efektywność algorytmu. Metryka ta okazuje się istotnie inna dla różnych modeli wykonania bez względu na ilość procesorów użytych w obliczeniu. Pod tym względem modele aktorowy i strumieniowy pozwalają osiągnąć znacząco lepsze wyniki niż pozostałe. Dodatkowo, eksperymenty pokazują, że odpowiedni dobór parametrów modelu strumieniowego potrafi odtworzyć charakterystykę pozostałych modeli - jest więc rozwiązaniem bardziej ogólnym, gdyż można w nim symulować pozostałe rozwiązania.

Podsumowując, w moich badaniach pokazuję, że model współbieżnego wykonania interakcji agentów może zostać rozdzielony od semantyki samego algorytmu. Umożliwia to porównywanie alternatywnych modeli wykonania dla tego samego algorytmu i, potencjalnie, dostosowywanie modelu wykonania pod konkretną architekturę sprzętową lub rozmiar problemu. Spośród rozważanych modeli wykonania, ten oparty na reaktywnych strumieniach okazuje się najbardziej obiecujący, zarówno pod względem efektywności jak i funkcjonalności. Efekty prac przedstawionych w niniejszej rozprawie będą mogły przyczynić się do ulepszenia istniejącego oprogramowania służącego do obliczeń agentowych, a w konsekwencji pozwolić na modelowanie bardziej złożonych problemów.

Contents

1. Introduction

Multi-agent systems are a bottom-up approach to the design or modeling of complex systems. Instead of trying to introduce the expected properties of the system up-front, this approach consists in defining simple rules governing the behavior of the basic components of the system. The complex properties are then said to *emerge* from the interaction of these simple constituents.

Multi-agent systems have been used to model biological systems in areas such as environmental biology, social sciences or engineering [1, 2]. Multi-agent systems have also been the foundation of the *Software Agent* design paradigm, both a precursor and complementary approach to Service Oriented Architectures and, more recently, Microservices. [3]

In other words, agents are both a domain modeling approach and a software engineering paradigm. These are two separate, but potentially complementary levels of abstraction. Indeed, the former may be implemented with the latter, using software such as Jade [4] As software agents are complex and computationally expensive, they are well suited to systems with low granularity. However, they are inefficient in the case of systems containing large numbers of agents [5]. Such massive multi-agent systems appear frequently in the field of numerical simulations, such as crowd and traffic simulation, economical systems, etc. [6, 7]. Another such area of application is computational intelligence, where agents are used within meta-heuristics as means of problem solving and decision support [8, 9].

This dissertation focuses on a class of such computationally intensive systems where large numbers of agents are the basis of the domain model and where their interactions are intensive and non-trivial. Interactions are intensive when they occur very often or when many calculations are needed to determine their outcomes. They are non-trivial when it is not possible to predict beforehand and in a general way which agents will interact with each other.

The main subject of my research is the concurrency of such interactions between agents, understood as the way agents perceive their interactions and the effects of these, and the parallelism of the execution of these interactions. The concurrency of agent interactions dictates how agents' interactions are related causally during the computation, which is linked to how information propagates in the multi-agent system. The parallelism of the execution of agent interactions describes whether some distinct agents' interactions can be executed at the same time as seen by an external observant, for example by running the computation on a multi-core machine or distributing it across a cluster of machines.

The manner in which interactions between agents are organized is referred to in this paper as their *concurrent execution model*. Various concurrent execution models are possible and differ both in the way

information propagates in the system (i.e. causality as it is observed by the agents), as well as in their technical characteristics, such as their amenability to parallelization and their scalability.

The type and degree of concurrency of agent interactions in multi-agent systems determines how closely they can simulate complex systems. As we are considering computationally intensive simulations, parallelism is in turn a key factor to scalability and therefore the ability to simulate bigger systems.

1.1. Motivation

In the case of massive, computationally intensive simulations, the domain-level multi-agent system is usually implemented as a discrete event simulation [10, 11]. Implementations of such multi-agent systems differ with regard to the granularity of the events in the simulation.

The events can be fine-grained and represent particular agent interactions. In such a case, concurrent interactions can be modeled by interleaving the handling of different agents' events. However, introducing parallelism to such an approach is complex, because it requires to have distinct streams of events which can be processed independently, for example one stream of events per agent. This in turn makes reasoning about the causality of agent interactions more complex. For the sake of simplicity and strong consistency, implementations with fine-grained events are usually single-threaded.

The events can also simply represent the passing of time. In such cases, the simulation is step-based and the population of agents is simply transformed into a new one at each step. Such a transformation can sometimes be computed in parallel. However, the concurrency of agent interactions is limited, as the causality of interactions is very specific: the effects of any interaction in a given step are visible precisely at the next step.

In both cases, the traditional approach to have both concurrency of interactions and parallel execution consists in grouping agents into separate agent *environments*, which become basic units of distribution [8]. Such environments limit the range of agent interactions, run a synchronous simulation on the inside and use asynchronous communication on the boundaries. To minimize communication costs, interactions across these boundaries are minimal. However, a distributed approach is no longer the most efficient solution on modern massively multi-core hardware, where the cost of communication is much smaller then in a distributed cluster. Current High Performance Computing infrastructures commonly include nodes with dozens of cores and these numbers are increasing rapidly [12].

The domain semantics of the multi-agent simulations, i.e. the decomposition into basic constituents and the definition of their elementary behavior, are often tightly coupled with the execution model of the underlying software framework. This makes it difficult to compare alternative approaches with regard to concurrency and parallelism.

1.2. Research scope and objectives

The scope of this dissertation are computationally intensive agent-based systems. As an example of such computationally demanding multi-agent systems, I focus my research on Evolutionary Multi-Agent Systems (EMAS) [13, 8]. These consist in combining multi-agent systems and evolutionary algorithms in order to solve difficult optimization problems. In contrast to classical evolutionary algorithms, in EMAS selective pressure, which is one of the main mechanisms of evolution, is not centrally enforced, but instead is an emergent property resulting from the interaction of independent agents. EMAS are interesting in the context of this research, because the organization of the agents' interactions has a significant impact on the behavior of the algorithm, as described in Section 2.2.

The objectives of this research are as follows:

- To introduce a formalization of the concurrent and parallel properties of a agent-based computation in the form of its *concurrent execution model*;
- to investigate how different execution models affect the properties of an agent-based computation, both in terms of the behavior of the algorithm and of the efficiency of the computation;
- To define an execution model which makes it possible to precisely control the concurrent properties of agents' interactions, while being able to efficiently execute on modern many-core hardware.

As a result, computationally intensive multi-agent systems will be portable across different execution models. This will allow to choose the one with the most interesting properties and performance for a given use case. In particular, a highly concurrent *and* parallel execution model will be able to use modern many-core hardware more efficiently and therefore allow larger-scale simulations.

1.3. Structure of the dissertation

The remainder of this dissertation is structured as follows: In section 2.1, I present the concept and a simple implementation of Evolutionary Multi-Agent Systems as the main use case under study. In section 2.2, I define the *concurrent execution model* of agent interaction on the base of an EMAS. Then, in section 3.1, I introduce a design pattern which makes it possible to decouple the semantics of such a agent-based computation from its execution model. In the remaining sections of chapter 3, I describe several such execution models with different concurrency properties, along with the corresponding publications. Finally, I summarize the experimental results from those publications in chapter 4 and end with concluding remarks in chapter 5.

2. Agent Interactions and Execution Models

This chapter describes Evolutionary Multi-Agent Systems as an example of a computationally intensive agent-based computation. It also introduces the concept of the concurrent execution model of agents' interactions and its importance to the behavior of an agent-based computation.

2.1. Use case: Evolutionary Multi-Agent Systems

As an example of computationally intensive multi-agent systems, I focus my research on Evolutionary Multi-Agent Systems (EMAS). They consist in combining multi-agent systems and evolutionary algorithms in order to solve difficult optimization problems.

Evolutionary algorithms are universal metaheuristics capable of optimization. [14]. However, their most classical designs, such as the Simple Genetic Algorithm [15] or Evolution Strategies [16]), greatly simplify the underlying biological model of evolution. For the sake of simplicity, these algorithms exclude many phenomena observed in real-world biological systems, such as dynamically changing environmental conditions, co-evolution of species or genotype-phenotype mapping. More importantly, they assume global knowledge during selection and generational synchronization during reproduction.

Evolutionary Multi-agent Systems (EMAS) [13] weaken those assumptions and to introduce more sophisticated biological mechanisms. EMAS implementations prove more performant than classical approaches when applied to difficult optimization problems, such as multi-criteria and multi-modal optimization in continuous and discrete spaces [17, 18, 19, 20]. They can also solve some inverse problems more efficiently because of a reduced number of fitness function evaluations [21, 22].

The core idea in EMAS is to evolve a population of agents. Every agent owns a genotype which encodes a candidate solution to the optimization problem. The representation of this solution is problem dependent, but will usually be a binary or real-valued vector.

This solution is evaluated to determine the fitness of the agent. The fitness is a number or a vector of numbers in the case of multi-objective optimization. The optimization problem consists in finding the candidate solution(s) with the best fitness. Depending on the problem, the best fitness may mean the maximum or the minimal one across all candidate solutions (or the set of vectors such that no other ones are strictly better in all dimensions, in the case of multi-objective optimization).

The genotype also determines the observable properties and behavior of the agent. Agents repeatedly interact among themselves, either directly or through their environment. There is no global knowledge in a multi-agent system, and agents act in an autonomous way [23]. Therefore, in contrast to traditional evolutionary algorithms, there is no centrally driven selective pressure. Instead, it is designed to *emerge* from individual agents' interactions in a decentralized manner [20].

Simple EMAS The following is a formalization of a simple EMAS. The main observable property of an agent is its fitness, which is the evaluation of the candidate solution assigned to the agent.

Selective pressure is introduced by providing agents with a piece of discrete, non-renewable resource called energy [20].

The behavior of the agent is driven by its energy levels. "Good" behavior is rewarded with additional energy, "bad" behavior results in energy being taken. This feedback loop induces selective pressure in the system in an emergent way. The semantics of "good" and "bad" behavior are implementation dependent. In this dissertation, I assume a very simple strategy: Being good means having better fitness.

Therefore, the rules for managing energy are the following:

– Agents in the first generation are given some initial energy;

– Agents receive some energy from their parents when they are created;

– If the energy of an agent is below some threshold, it fights with another agent by comparing their fitness value – the better agent takes energy from the worse one;

– Agents with sufficient energy can reproduce and yield new agents. The genotype of the children is derived from their parents using classical genetic operators;

– When the energy of an agent drops to zero, it is removed from the system.

Figure 2.1 illustrates the idea of a fight between agents: agents compare fitness, and the loser gives some of its energy to the winner. Figure 2.2 shows agent reproduction for agents. Sexual and asexual reproduction is performed by deriving the genotypes and fitness of the children from the ones of the parents using traditional genetic operators and by transferring some energy from the parents to the children.

In contrast to traditional evolutionary algorithms, the number of agents may vary over time. The system remains stable as long as the total energy remains constant. It is possible to adapt the size of the population by externally changing the amount of energy in the system.

2.2. The execution model of agent interactions

What remains to be defined in the above algorithm is how we choose which agents fight or reproduce with each other and when they do it. We can generalize this question as one about the concurrency of

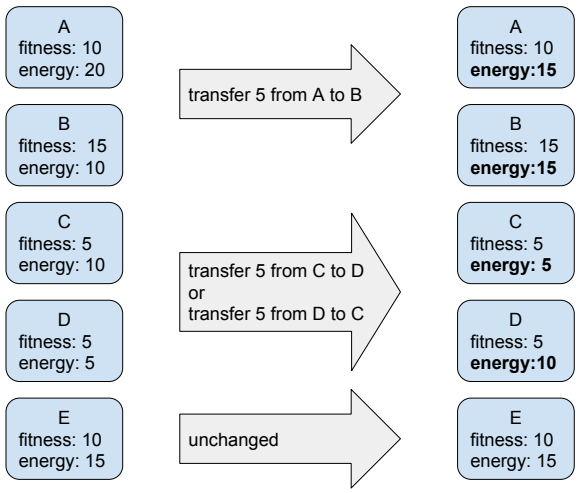


Figure 2.1. An example of a fight strategy for agents. Agents are paired together, and for each pair the agent with lower fitness transfer some of its energy to the one with higher fitness. Draws are resolved at random, and if there is an odd number of agents the last one is left unchanged.

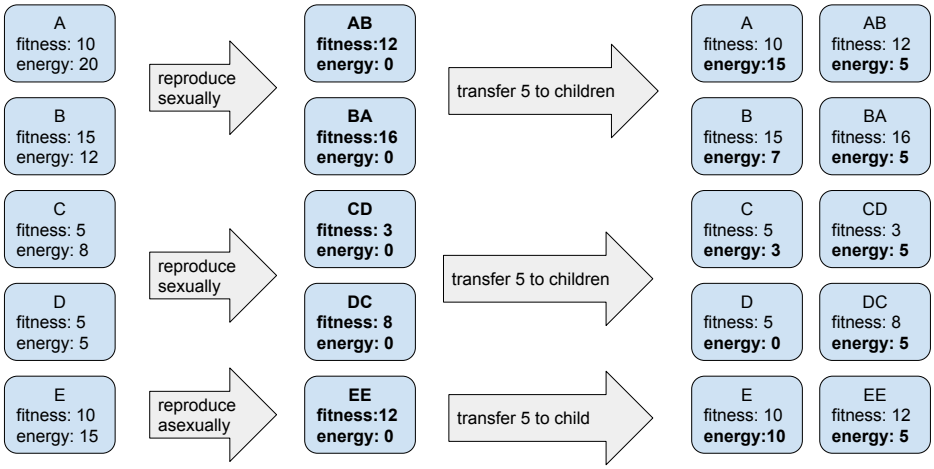


Figure 2.2. An example of a reproduction strategy. Agents are paired together, and each pair reproduces sexually. If there is an odd number of agents, the last one may reproduce asexually. Some problem dependent variation operators are used to generate the children genotypes and the resulting fitness. Finally, parents share some of their energy with their children.

interactions between agents, understood as the way agents perceive their interactions and the effects of these. The manner in which interactions between agents are organized is referred to in this paper as their *concurrent execution model*. Various concurrent execution models are possible and differ both in the way information propagates in the system (i.e. causality as it is observed by the agents), as well as in their technical characteristics, such as their amenability to parallelization and their scalability.

EMAS are interesting in this context, because the organization of the agents' interactions has a significant impact on the behavior of the algorithm. For example, a computation where every agent always interacts with a restricted set of neighbors will yield different results than a computation where every pair of agents can meet – in the first case there is a higher chance that multiple, genetically diverse subpopulations coexist, which is a phenomenon called *allopatric speciation* and can be useful for example in multimodal optimization [24, 25].

Another aspect of the model of interactions is how they are related causally during the computation. In other words, how concurrent are the interactions as perceived by the agents. For example, one strategy for the Simple EMAS above would be to draw a random pair of agents on every step for them to compete or reproduce. Another strategy would be to divide the whole population into pairs of agents, and after the interactions of every pair are done, merge the resulting agents into a new population.

Depending on the choice, the evolutionary dynamics will be very different. In the first case, the results of a single interaction are immediately visible to all agents in subsequent interactions. As a result, genes will diffuse much quicker in the population. However, depending on how we choose the pair of agents, some agents may have more opportunities to interact than others, which introduces additional, uncontrolled selective pressure. In the second case, the population changes in a more uniform way: in every step every agent is allowed precisely one interaction. However, every agent must also wait for all other agents to have finished their interaction to be able to move on to the next one, so the overall throughput is reduced

It is difficult to predict in a general way which variant will be more efficient in solving a particular optimization problem. The next chapter introduces an approach to the design of multi-agent systems which makes it possible to abstract from their execution model, in order to meaningfully compare alternatives.

3. Concurrent Execution Models

In this chapter, I propose a design for multi-agent systems which makes it possible to abstract the semantics specific to a particular application from the underlying execution model, which enables meaningfully comparing alternative ones. I subsequently introduce and analyze different execution models with varying strengths and weaknesses. Each model is described in more depth in the corresponding publication.

3.1. Abstracting the execution model

As described in Section 2.2, the order and organization of agents' interactions can determine the outcome of the algorithm. It also has a significant impact on the technical properties of the algorithm, such as its potential for parallel execution and therefore its efficiency.

The simplest strategy to organize agents' interactions is to shuffle the list of agents and then compute the results of the meetings between subsequent pairs of agents. This standard approach is very simple to implement but has major drawbacks:

- It requires to collect the whole population in a single operation, which is against the decentralized nature of the algorithm.
- Agents who already finished their meeting must wait for others - there is a synchronization barrier applied to the population as a whole.
- Agents are grouped together independently of the action they want to perform. This may be wanted, but it requires to handle all possible combinations of agent behaviors.

Meeting Arenas In order to be able to meaningfully compare execution models for the same algorithm, it is necessary to decouple the semantics of the multi-agent system, i.e. the decomposition into basic constituents and the definition of their elementary behavior, from the underlying execution model.

To that purpose, I introduce an approach based on the Mediator design pattern [26]. I use the metaphor of *meeting arenas* to convey its intuition. Based on their state, agents select choose an action they are willing to perform, such as fighting, reproduction, etc. Then, agents conceptually move to an arena where they can meet other agents willing to perform the same action.

In other words, meeting arenas allow to split a flow of incoming agents into groups of coherent behavior. Each kind of agent behavior is represented by a separate arena. Depending on the type of the behavior, agents are grouped together within arenas and interactions can proceed (see Figure 3.1).

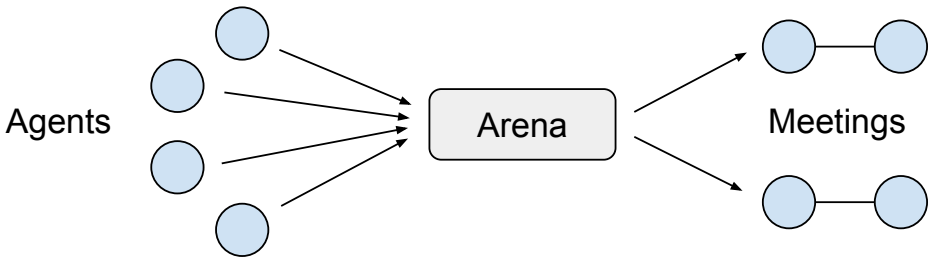


Figure 3.1. Meeting arenas group similar agents and coordinate meetings between them.

Therefore, the semantics of the agent-based algorithm are fully determined by two functions. The first one consists in *agent behavior*, which chooses the arena to meet on based on the state of an agent. The second corresponds to the *meeting operation* which is computed at every arena for groups of agents. This approach resembles the *MapReduce* programming model [27]. The agent behavior partitions the agents population into meeting arenas just as in the *mapping phase*. The meeting logic transforms the population, which is then trivially aggregated as in the *reduce phase*.

Agent Behavior Figure 3.2 shows an example of the behavior function for the Simple Emas described in section 2.1. The behavior function chooses a behavior based on the current energy level of the agent. If an agent has no energy, it dies. If it has enough energy it reproduces, otherwise it fights.

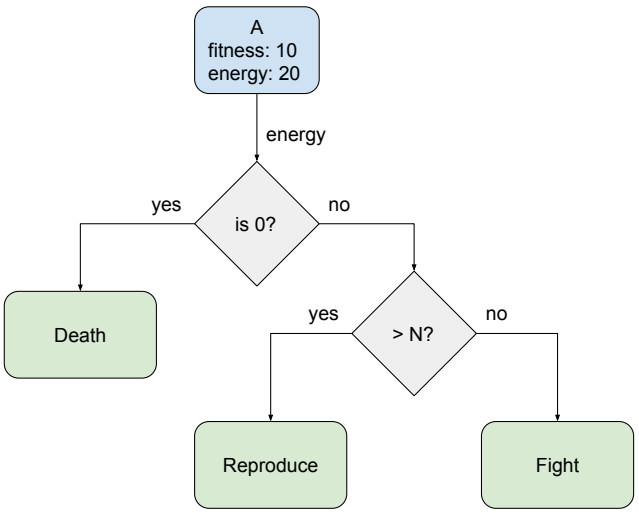


Figure 3.2. The behavior function chooses the behavior of an agent based on its current state. In this use case, if an agent has no energy, it dies. If it has enough energy it reproduces, otherwise it fights.

Agent Meetings For a group of agents exhibiting the same behavior, the meeting function produces a new group of agents (Figure 3.3). In particular, it can change the size of the group by removing agents or adding new ones. For example, the death strategy simply outputs an empty list, which removes agents from the system by not including them in the output. The specifics of each type of meeting is further decomposed in *meeting strategies*, already described in Figures 2.2 and 2.1.

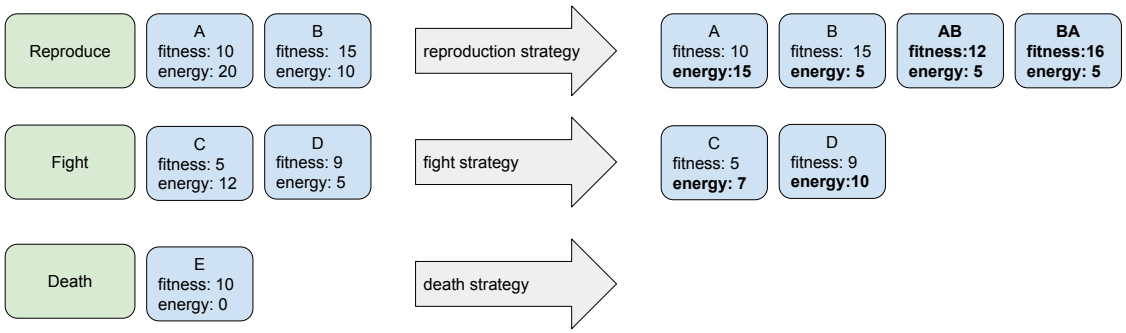


Figure 3.3. For a given behavior and a group of agents exhibiting that behavior, the meeting functions yields a new group of agents. The output group may contain new agents (which are added to the system) or skip some input agents (which are then removed from the system).

Meeting arenas also simplify the algorithm at the conceptual level. As agents only interact through coherent behaviors, the amount of handlers needed to be implemented is linearly proportional to the number of behaviors to handle. In contrast, if we allow agents with inconsistent behaviors to interact, we need to consider the full Cartesian product of possible scenarios.

Additionally, as arenas are specialized for a particular type of behavior, they can choose the best arity for the meeting operations. In particular, this strategy can be dynamic, e.g. a fighting arena may usually make agents compete in tournaments of size 10, but choose to trigger a smaller tournament if fewer agents have entered the arena within a time frame. Conversely, it may release the agents from the arena and let them potentially choose another action to perform.

The fact that an agent has to settle on a single behavior in the behavior function may seem limiting at first. For example, it is not possible to directly model the following strategy: "If there is someone around with less energy than us, attack. Otherwise, if we have sufficient energy, self-reproduce". However, exhibiting several different behaviors at the same time is, for most intents and purposes, the same as quickly alternating between them. In other words, such an agent could repeatedly enter and leave the fighting and self-reproduction arena in turns until the condition it seeks are fulfilled.

Another apparent limitation is that agents have to choose an arena not knowing who awaits them there. In fact, this is in accordance with MAS principle of avoiding global knowledge. Knowledge can only be acquired by interacting with other agents and the environment. Until it enters an arena and meets other agents, an agent should not have any additional information. Of course, it is allowed to remember past experiences and act accordingly.

Execution Model When the semantics of the multi-agent system are described using the behavior and meetings functions, all that is left to be defined is when and where these functions will be applied, as well as how agents with similar behavior will be grouped together. The design and implementation decisions about these aspects results in a particular *execution model*.

Meeting arenas allow to decouple the semantics of the algorithm from the execution model, which allows us to consider different concurrent execution models and compare their performance for the same algorithm.

3.2. Synchronous execution model

The simplest execution model is a typical, synchronous implementation of an agent-based computation, equivalent to a discrete event simulation (Figure 3.4). In every step, the population of agents is shuffled and partitioned by the behavior function into groups corresponding to separate behaviors. Every such group is then processed by the appropriate arena to yield an output group. These are merged in order to form the new population. The step function can then be iterated a given number of times on an initial population.

Note that even though the computation is synchronous, agents perceive the interactions within one step as being simultaneous - the effects of the meetings in any given step will only be visible to the population in the next step. We can also look at this property as a synchronization barrier - agents wait for each other at the end of the step, before simultaneously synchronizing their knowledge about the effects of all of their interactions. We can also identify discrete *generations* in the evolutionary algorithm. A single agent can survive in multiple generations, but agents can only interact within such a generation.

From a technical point of view, every agent is progressing at the same pace, no matter the type of interaction. All the interactions in a single step have to terminate before the whole population can move on to the next interaction. This can be inefficient if some types of interactions take more time than others, as it is the case in our Simple EMAS; reproductions take much more time than fights, as they need to apply genetic operators to the derive the genotypes of the children and also compute their new fitness, which are both a costly computation.

The following publication describes in more details the concept of meeting arenas introduced above and applies it to several variants of the synchronous model.

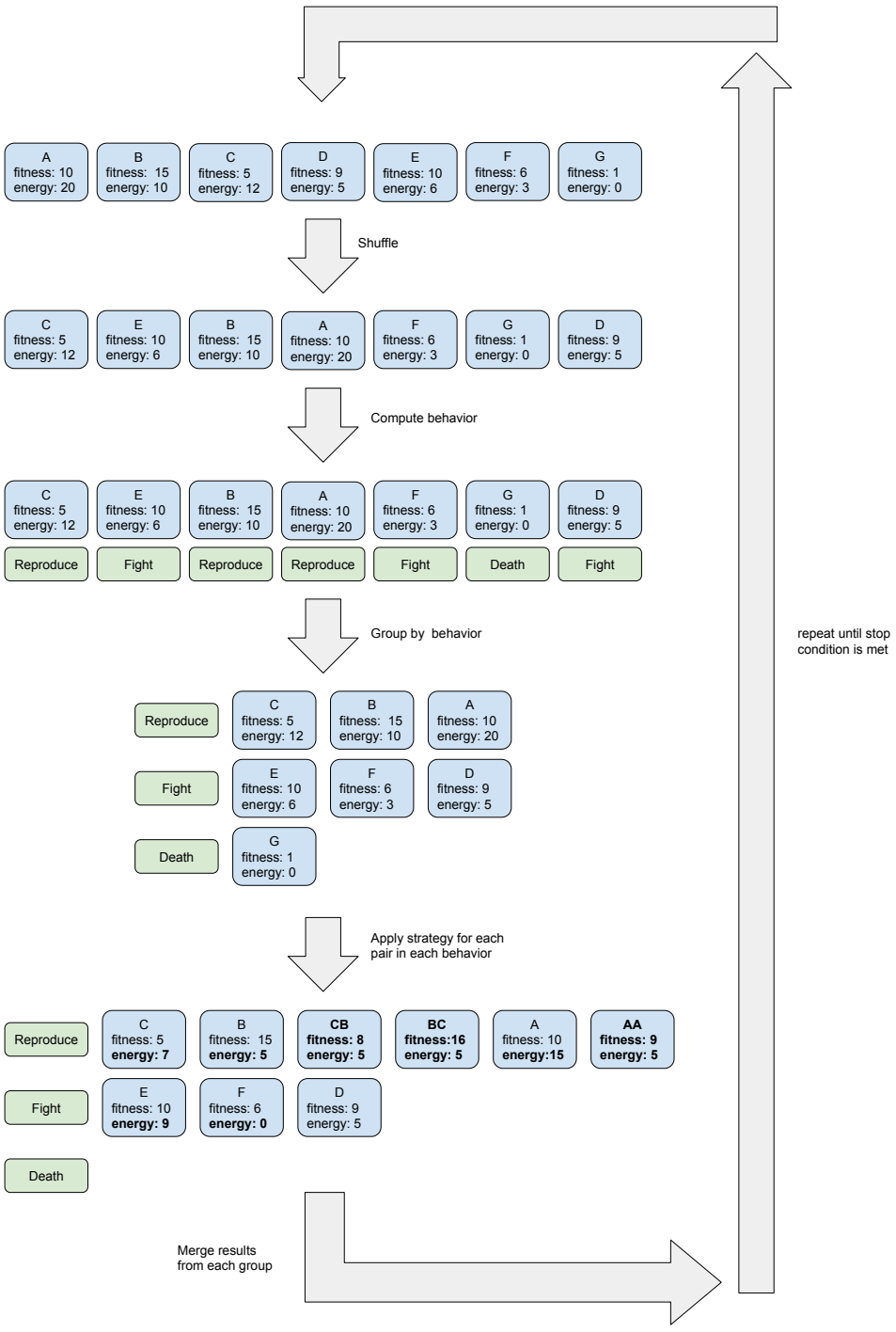


Figure 3.4. In a synchronous execution model, the population is transformed step by step by first grouping agents according to their behavior, then computing the meetings function on each group of similar behavior, and finally combining the results.

Future Generation Computer Systems 37 (2014) 390–400

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Computing agents for decision support systems

D. Krzywicki, Ł. Faber, A. Byrski *, M. Kisiel-Dorohinicki

AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Krakow, Poland

HIGHLIGHTS

- Applicability of agent-oriented metaheuristics to decision support systems.
- Functional-programming based prototypes of agent-based computing systems.
- Experiments regarding scalability and performance of the implemented systems.

ARTICLE INFO

Article history:
Received 4 July 2013
Received in revised form
1 January 2014
Accepted 7 February 2014
Available online 17 February 2014

Keywords:
Decision support systems
Multi-agent systems
Scalability
Performance

ABSTRACT

In decision support systems, it is essential to get a candidate solution fast, even if it means resorting to an approximation. This constraint introduces a scalability requirement with regard to the kind of heuristics which can be used in such systems. As execution time is bounded, these algorithms need to give better results and scale up with additional computing resources instead of additional time. In this paper, we show how multi-agent systems can fulfil these requirements. We recall as an example the concept of Evolutionary Multi-Agent Systems, which combines evolutionary and agent computing paradigms. We describe several possible implementations and present experimental results demonstrating how additional resources improve the efficacy of such systems.

1. Introduction

The need to gather and analyse vast amounts of information from numerous sources has grown in importance. Such data is often a basis for simulations and computations that support decision making. It may be needed to run many computing tasks, in order either to test different parameters in a model or to verify a statistical hypothesis. An exhaustive search for optimal solutions to a decision making problem is usually time-consuming and thus not acceptable in real-time conditions. Instead, metaheuristics may quickly provide good-enough options to be further considered in the decision making process [1].

Examples of use cases with real-time constrains, where a quick approximated solution may be better than an outdated optimal one, may include:

- portfolio optimisation—a decision support system can apply different models to the available market data and allow the user to quickly react to arising trends [2];
- crisis management—in crisis situations, such as fire outbreaks, flooding or earthquakes, intensive simulations are required in order to suggest possible evacuation routes or to assign rescue units to tasks. Geographical information usually needs to be considered, yielding optimisation problems similar to transportation-related ones [3];
- production planning—decision support systems can help in scheduling work, rescheduling production plans in the case of hardware failures, implementing just in time strategies or balancing conflicting goals (e.g. high system throughput vs low machinery usage) [4].

Metaheuristics may still require a significant computational power if the acceptable solution is to be found in a reasonable time. For this purpose, large-scale infrastructure is usually used, such as clusters, grids or clouds. To fully benefit from this computational power, it is required to appropriately plan their development and deployment, along with adequate tools and careful testing.

Because of its intrinsic decentralisation [5], the agent approach is well suited to design scalable distributed models and has been applied in various decision support systems. This approach may be summarised as the introduction of artificial intelligence techniques into the system, transforming it from a passive tool into an active collaborator in decision making. A number of such case-oriented systems have been proposed and verified in practice [6,7].

* Corresponding author. Tel.: +48 126339406.
E-mail addresses: daniel.krzywicki@agh.edu.pl (D. Krzywicki), faber@agh.edu.pl (Ł. Faber), olekb@agh.edu.pl (A. Byrski), doroh@agh.edu.pl (M. Kisiel-Dorohinicki).

http://dx.doi.org/10.1016/j.future.2014.02.002
0167-739X/

Well-known general-purpose agent-based development tools (such as JADE [8], RePast [9] or MadKit [10]) may not be the best choice to implement such computational intensive simulations, when throughput and scalability are more important than code migrations or FIPA-compliant communication. Therefore, over the last 10 years, we have been involved in the development of several alternative platforms dedicated to large scale agent-based simulations and computations [11–13].

In this work, we discuss the implementation aspects of using computing agents in large-scale environments, with a focus on performance. We compare different approaches to agent execution and parallelism, based on the metaheuristic called evolutionary multi-agent systems (EMAS), which is a hybrid of agent-oriented and evolutionary-based computing [14]. We introduce the concept of *meeting arenas*, which allows to design more efficient and scalable multi-agent systems. Nevertheless, we show that explicit parallelism, when each agent is mapped onto a thread, can be much less effective than a simple but optimised sequential implementation. Finally, we show that such agent-based metaheuristics can be easily scaled with additional computational resources.

We start the paper with a discussion on the applicability of the agent-oriented paradigm and metaheuristics in decision support systems (Section 2), along with an EMAS example. In Section 3, we introduce the most common approaches to parallelism in agent-oriented computing and follow with a review of popular agent platforms in Section 4. We describe in Section 5 how to implement an evolutionary multi-agent system using two different approaches—a synchronous and asynchronous one. Finally, we conclude the paper by comparing the performance and scalability of both approaches in Section 6.

2. Agent-based metaheuristics in decision support

Decision Support Systems (DSS) are information systems that support different business or organisational activities involving decision-making. They are especially useful in situations where quickly changing, hard to specify in advance conditions are encountered. Referring to Power's taxonomy for DSSs [15] this paper focuses on Model-driven DSSs, which help the users in the analysis of the current situation by allowing to manipulate statistical, simulational or optimisational models.

2.1. Metaheuristics for DSSs

The models used in DSSs are usually very complex and computationally hard, because the underlying problems are very difficult as well. In such cases, one often turns to solutions based on so-called heuristic methods, which provide "good-enough" solutions without caring whether they may be proved to be correct or optimal [1]. These methods trade-off precision, quality and accuracy in favour of smaller execution time and computational effort. They are necessary to deal with difficult problems, and are referred to as methods of the last resort [16].

A general definition of a heuristic algorithm, which does not specify details such as a particular problem, search space or operators, is called a *metaheuristic*. For example, a simple algorithm such as greedy search may be defined without going into more details as "an iterative, local improvements of a solution based on random sampling" [17].

A simple but adequate classification of metaheuristics (cf. [18]) distinguishes two groups of techniques. Single-solution metaheuristics work on a single solution to a problem, seeking to improve it. The examples are greedy search, tabu search or simulated annealing. Population-based metaheuristics explicitly work with a population of solutions and put them together in order to generate new solutions. The examples are evolutionary algorithms, immunological algorithms, particle swarm optimisation, ant colony optimisation, memetic algorithms and other similar techniques. They are usually inspired by nature and imitate different phenomena observed in e.g., biology, sociology, culture or physics [19].

2.2. Agent approach

The key concept in multi-agent systems (MAS) consist in intelligent interactions, such as coordination, cooperation, or negotiation. Therefore, multi-agent systems are ideal in representing problems which can be solved using multiple methods by numerous entities with various perspectives. One of the most important features in a multi-agent system is the autonomy of the agents, as they can fulfil the tasks assigned to them according to their own strategy and the situation observed in their environment. In consequence, agents are adaptable and proactive [5].

Combining the agent-oriented approach with population-based metaheuristics seems natural but has yet been the topic of little work. The entities processed in the course of the computation can often be considered autonomous and treated as agents in a common environment. The operations involving many such entities can be defined as interactions between these agents.

This change of modelling perspective allows to perceive parts of the system on a higher abstraction level and build hybrid systems which combine techniques from different metaheuristics. New problems often arise, such as the lack of global knowledge or the need for proper synchronisation of the agents' actions. However, interesting and useful effects also often result from the cooperation of different mechanisms in one system [20].

In this paper, we focus on an example of such a hybrid approach, in which agents are subject to an evolutionary process. Such a combination yields interesting new features when compared to classical evolutionary algorithms, such as a decentralised an emergent selective pressure.

2.3. Evolutionary multi-agent systems

Agents in an evolutionary multi-agent system (EMAS) represent solutions to a given optimisation problem.

Inheritance is achieved through reproduction, with the possible use of variation operators such as mutation and recombination, like in classical evolutionary algorithms. Yet agents are to be autonomous in their decisions and no global knowledge is available to them. Therefore, in contrast to classical evolutionary algorithms, selection needs to be decentralised and involve peer-to-peer interactions instead of being system-wide.

In order to do that, a solution based on the acquisition and exchange of non-renewable resources has been proposed in [21]. The quality of the solution represented by the agent is expressed by the amount of resources the agent owns. In general, these resources should pass from worse agents to better ones. This might be realised through encounters between agents, which cause better ones to end up with more resources and make them more likely to reproduce. Worse agents lose resources which increases the probability of their death. Because of such indirect dynamics of reproduction and death, agents' lifespans overlap and so do the generations. Moreover, the size of the population is dynamic and can be changed by varying the amount of available resources. A detailed study of computing with EMAS, in particular the influence of its different parameters on the computing efficiency may be found in [22].

Agents are grouped within *environments* which define the information and resources an agent has access to. Agents can interact

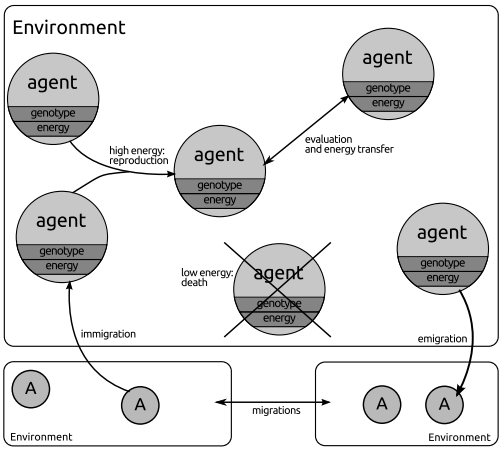


Fig. 1. An example of a simple evolutionary multi-agent system (EMAS). Agents with higher fitness take energy from agents with lower fitness. High levels of energy increase the probability of reproduction and reduce chances of death. In consequence, the selection process is decentralised and selective pressure softens. Multiple agent environments can be connected through agent migrations, like in the classical island model.

with each other directly only within the same environment. However, they are able to move to another environment, thus exchanging information and resources all over the system [14] (see Fig. 1).

Environments are largely independent and communicate only through agent migrations. Therefore, they can be easily treated as basic units of distribution, as in the classical island model in evolutionary algorithms. In addition to improving the performance of the algorithm, it also increases the diversity of solutions in the whole population (allopatric speciation). Other metaheuristics can also be introduced, such as immunological selection [23] and niching [24].

The principle of an evolutionary multi-agent system consist in the explicit hybridisation of agent-oriented and evolutionary computing. This contrasts with usual agent-oriented approaches, which use the agent-paradigm to solve certain tasks by delegating them to particular agents and combining the outcomes of their work (see, e.g. [25]).

3. Interaction and execution models for agents

In agent-oriented computing systems, agent interactions are one of the crucial aspects of their work. It is easy to predict that parallelising them can significantly increase the throughput of the system. However, this comes at the cost of increased communication and synchronisation. Therefore, an important issue is to choose the appropriate granularity of the entities in the computation.

As agents are defined as autonomous and independent beings, it seems natural to look for further concurrency within a single environment. The question is where to put the boundaries of concurrent execution, as it has consequences on both performance and ease of programming. This section discusses the most common models of execution and interaction in the existing agent software.

3.1. Heavyweight agents

In this model every agent is associated with a thread and communicates through message passing. Some agents may passively wait for incoming messages and react to them. Other agents may actively initiate interactions with other agents. It is difficult to achieve a coordinated life cycle among such agents, since the corresponding threads may be arbitrary interleaved. Therefore, some kind of synchronisation between agents still needs to be introduced, usually in terms of a specific communication protocol.

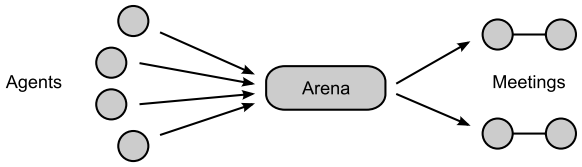


Fig. 2. Meeting arenas allow to group similar agents and coordinate meetings between them.

In order to interact with each other, agents need to locate other agents willing to perform the same actions. For example, in an evolutionary multi-agent system, an agent with enough resources to reproduce needs to find another one which also has enough resources. In order to do that, it could ask all other agents in the population. However, such a solution is obviously inefficient, because of the intensity and redundancy of the required communication.

A better approach, introduced in this paper, is to use a mediating entity, which we call a *meeting arena*. Every time an agent wants to perform an action, it chooses an appropriate arena to meet with other similar agents. The arena is then able to partition its members in groups of some given arity and mediate the meeting itself (see Fig. 2). Examples with pseudocode are given in Section 5.

The usage of meeting arenas should bring many benefits, not only in terms of efficiency, as the algorithm itself can be structured more clearly. Agents only need to be given a set of rules, in order to choose an arena on the basis of their state. The actual protocol of agents interactions can then be defined at the level of the appropriate arena.

Assigning a thread to each agent may feel very natural. In practice, however, the number of agents is often much higher than the number of cores, especially in simulations. Performance may then be seriously hindered by frequent context switches, although this overhead may be reduced by sharing a pool of threads among agents. However, this model still involves intensive communication and costly processor cache synchronisation. In consequence, the trade-off for such concurrency may be higher than expected.

3.2. Lightweight agents

An opposite approach is to consider agents as parts of the model, not parts of the implementation. As such, they are simply represented as data structures and processed like in a discrete event simulation.

The execution of an individual agent has to be divided into smaller parts which can be interleaved. These parts, which we will call *actions*, could for example consist in executing a single step or querying a neighbour. Given its current state, every agent decides which action to perform next. This action is submitted for later execution to an executor service owning a pool of threads, like in the Command Object design pattern.

The difference with regard to classical discrete event simulation is that actions are generated synchronously but can be executed asynchronously. In other words, the state of an agent during the execution of the action may be different from the time when the action was created.

The performance of such a model will usually be higher than in the previous one, more consistent memory access patterns resulting in more efficient processor usage. Even though the explicit parallelism is reduced, throughput can be improved, because frequent agent interactions no longer need to be synchronised between threads.

Moreover, independent actions can still be executed in parallel by the executor service. This is consistent with the meeting arena concept described above, as actions on common subsets of agents may be grouped together and considered as a single meeting.

4. Distributed and parallel multi-agent frameworks

This section provides an overview of existing multi-agent frameworks. In our review, we focus on parallelisation and distribution capabilities, with regard to the aspects discussed in the previous section.

First, we briefly describe some selected tools which specialise in metaheuristics. They are interesting examples of improving metaheuristics with agent systems, but lack more general agent-oriented features. However, all of them share the idea of an agent being an *executor* of the algorithm and not a *participant*.

The platforms described in the consecutive sections provide more sophisticated support for agent-based systems but are not necessarily well suited to metaheuristic computations. Some properties are shared by almost all of them, such as: the choice of an object-oriented programming language (mostly Java) and a representation of agents as objects with an internalised state. Other characteristics include: models that are too heavy (e.g., JADE due to FIPA compatibility), a large and complicated code base due to the implementation of many communication, distribution and component-oriented mechanisms in the platform instead of using ready solutions (e.g., Jadex).

4.1. Metaheuristics frameworks

The four frameworks presented below are examples of introducing multi-agent systems to metaheuristic computations. They are specialised to this purpose and lack well-defined distribution facilities. We present them as alternative models but we do not discuss their implementations in-depth.

MAGMA (Multi-agent architecture for metaheuristics): MAGMA [26] is a multi-level (hierarchical) architecture of a multi-agent system. Each level of agents has different objectives and represents a different level of abstraction of the algorithm. For example, level 0 agents generate a sample solution and then level 1 agents improve it by searching the neighbourhood of that solution. There may be several agents that participate in an algorithm on each level. Composing different metaheuristics is also possible with a coordination provided by a higher level of the architecture (level 3). This way agents wrap selected functions of the algorithm and not the whole algorithm itself (as it will be the case for further platforms).

MAS-DGA (Multi-agent system for distributed genetic algorithms): the attention of the authors in [27] is focused on approaches to the question of migration. They propose the MAS-DGA framework that comes from the concept of Distributed Genetic Algorithms, where the population is divided into interacting subpopulations handled by different genetic algorithms (GA). In the case of MAS-SGA, these GA are encapsulated in agents. The authors suggest a possibility of distribution on the agent level but they do not provide descriptions of any specific examples nor implementations of this model.

AMF (Agent metaheuristic framework): the authors of AMF [28] extend metaheuristics with an agent-oriented approach and an organisational model based on *roles* and *interactions*. The RIO meta-model described in the paper involves the three following concepts: Role, Interaction and Organisation. Metaheuristics are organisations, and agents play specific roles in these organisations. Some of the defined roles are: the intensifier (performs a search in a search space), the diversifier (identifies new promising regions in the search space), the guide (structures the information from two previous roles), etc.

MAS4EVO (Multi-agent system for evolutionary optimisation): in [25] the authors propose a model and a framework (DAFO—Distributed Agent Framework for Optimisation) that is a significant improvement over the previous three. The framework is built on MadKit (see Section 4.3). In this model, authors introduce three types of agents: problem solving agents which optimise functions, fabric agents which are responsible for initialising and configuring the computation, and observing agents which generate output for the end-user.

4.2. Jadex

Jadex[1] is an agent-based programming framework that exploits a novel approach to agents–components unification called "active components" [29].

The concept of Active components unifies SCA (Service Component Architecture) components with agents. This results in components that are able to use, in addition to traditional required and provided service interfaces, asynchronous messaging and that can act autonomously. It has a tremendous impact on behaviours of these components. They, for example, can refuse service call execution when they cannot or do not want to process the request.

Agents: Jadex offers two ways to implement agents. It is possible to use full-featured, BDI (belief–desire–intention) agents and simple, so-called *micro agents*. Micro agents are usually just annotated POJO (Plain Old Java Objects) classes. They follow three-phased execution semantics: initialisation, execution and termination. Additionally, an agent can schedule actions to be run later. As an agent is also an active component, it may receive service calls and incoming messages.

Distribution: distribution in Jadex is provided transparently to the developer and it is implemented using a layered architecture. Services, for example, may use remote asynchronous method calls. Transparency is achieved by using proxy interfaces implementation. Internally, remote calls are implemented using asynchronous messaging between remote management system components. Messages are encoded and transmitted through some chosen *stream*. The encoding of messages is provided by *codecs* which need to (un)marshall Java objects to binary or XML format but which can also provide more sophisticated features: e.g., compression or encryption. A stream can use different communication transport: TCP, HTTP and others.

The second aspect of distribution is the peer awareness and discovery. Jadex takes care of it automatically on all levels, including service (i.e. interface) binding. When there is a look-up for a required service, proxy components on a local node redirect search requests to the remote management system to perform remote look-ups of services.

Before that, remote platforms in the network need to be discovered. For this purpose, Jadex provides a few different mechanisms: e.g., broadcast discovery which sends UDP announcements about a platform on a local network or registry discovery in which there is one, central registry created for all platforms to announce themselves.

Other features of Jadex include, among other things, support for interaction with external systems using web services and a GUI-based control centre.

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

4.3. MadKit

MadKit[2] is a generic, customisable multi-agent platform based on a specific organisational model [10]. Agents are divided into groups and they may have particular roles in them. The centralisation of the platform around the organisational concepts is, in the view of the authors of the platform, a key element for building heterogeneous systems.

The MadKit architecture is built on the agent-group-role (AGR) model. This model is used to built organisations: an *organisation* is described using terms of interacting *groups* and *roles* and is separated from the concept of an agent.

Agents: a MadKit agent is an entity that can communicate and which has several roles within one or more groups. Groups are atomic structures aggregating agents and they can overlap. Roles are tags for agent functions within groups. Agents request them on their own and they may be granted or denied them. Communication between agents is achieved using asynchronous messaging. Addressing of agents is done using their addresses or by their specific roles in one of their groups.

Architecture and distribution: the architecture of MadKit is developed around the AGR concept. Moreover, it follows some additional design decisions, the most interesting being: micro-kernel architecture and agentified services.

The micro-kernel, responsible for basic platform management, handles only most essential functions. The rest of the needed services is handled by agents. The micro-kernel tasks are: control of groups and roles, life-cycle management of agents, local messaging. It also supports so-called "kernel hooks" which allow the extension of its functionality by operations executed in the publish–subscribe model. Two types of hooks are supported: monitor and interceptor hooks. The former can be used by many agents at the same time whilst the latter can be hold on by only one agent and can be used to prevent the operation from successful execution. Additionally, it is possible to execute actions on kernel when an agent is a member of the *system* group.

The agentification of services describes a concept of turning system services (e.g., distributed message passing, migration) into agents. This makes the platform very extensible and flexible as every component can be easily replaced: communication with services is no different than with other agents.

MadKit has support for transparent distribution. Groups can span across many platform nodes. It is provided by two roles in the *system* group: communicator (routes messages to other nodes) and synchroniser (keeps information about groups memberships synchronised across all nodes).

MadKit provides also a graphical environment for visualising simulations and controlling the platform.

4.4. μ^2

μ^2 (micro-squared)[3] is a multi-agent platform centred around the concept of a μ-agent (or micro-agent): a small-size agent, that can be recursively constructed from other micro-agents with decomposition and fine-grained separation of concerns in mind [30]. μ^2 is implemented in Java and in Clojure and available under GPL 3.0 license.

Agents: micro-agents are autonomous, persistent, reactive and proactive. They can play one or more roles which fulfil so-called applicable intents. Intent is another name for an intention or "abstract request specification". The platform provides some organisational modelling approaches. An agent can be in a group leader role. In such case, it controls many sub-agents, propagates control messages and structures the society. Due to this approach agents can construct themselves using sub-agents, and sub-agents also can be group leaders. This is how decomposition can be implemented in the platform. Other roles include: social roles that allows agents to communicate asynchronously and passive roles which support only synchronous communication. The latter role was introduced to reduce possible performance penalties resulting from asynchronous messaging.

Distribution: as it is mentioned in [30], μ^2 is a platform that can be run on Android devices. Micro-agents are encapsulated into Android service and they are integrated into the rest of the system. Communication between normal applications and agents is transparent.

4.5. JADE

JADE[4] is a mature (founded in 2000) Java framework for developing agent-based applications with a very strong relationship with FIPA specifications [8]. Its architecture is focused on a peer-to-peer communication with some centralised services. Two software components are specified: agents (autonomous, using asynchronous messaging) and services (non-autonomous, running on a single or multiple nodes).

Agents: JADE agents exist in containers (basically Java processes) which can be distributed over the network. The way messages are structured is compliant with FIPA Agent Communication Language.

Distribution: the peer-to-peer nature of JADE made it possible to create many reimplementations of nodes, e.g., for mobile environments like Android [31]. Distribution is gained by splitting a JADE container into a frontend and backend. The former runs on a mobile device and is rather lightweight, the latter usually runs on a more powerful computer.

4.6. Repast suite family

Repast[5] is an open-source, agent-based modelling and simulation toolkit [9]. It has many versions for various programming languages. The most interesting ones are the newest: Repast Simphony (for Java) and Repast HPC (for C++). All of them use the "new BSD" license.

Repast Simphony is a complete rewrite of older Repast 3 with a modular architecture, extendable via plugins. Individual components (e.g., networking, logging) can be replaced easily. Plugins are layered and separate layers can be replaced with similar easiness. There is a separation between the model specification, execution, data storage and visualisation. A core of the Repast Simphony consists of components responsible for simulation functions (e.g., time scheduling, space management, random number generators).

Agents: agents are modelled as objects, collections of agents – as contexts, and the environment – as projections. A *context* is a set of objects and may represent an agents' population but does not describe any structure or relationships between agents. The second term – *projection* – was created to define structures of agents in contexts. They may be, for example, network or grid structures.

Distribution: Repast does not offer distribution facilities similar to other platforms. However, a user can prepare its own distributed environment using external facilities, for example, Java RMI (Remote Method Invocation), which is an object-oriented remote procedure call mechanism.

[2] http://www.madkit.net.
[3] http://sourceforge.net/apps/mediawiki/micro-agents/.
[4] http://jade.tilab.com/.
[5] http://repast.sourceforge.net.

5. Implementation aspects

Evolutionary multi-agent systems and similar agent-based computing systems need lightweight, reusable and easy-to-parallelise solutions. In particular, the implicit agent-orientation perceived at the implementation level of these platform does not seem inevitable to us. We think that agent features should be a part of the conceptual level, but do not need to be reflected in the implementation in the case of computing systems.

Considering the execution models described in Section 3 and their implementation in existing software tools for multi-agent systems, we wanted to compare these two approaches to tell what granularity is best suited for agent-based computing and simulation. In order to abstract from the properties of these frameworks not relevant to the problem, we implemented two custom versions of an evolutionary multi-agent system.

In the first version, agents are asynchronous and can be mapped to separate threads or share a thread pool. The second version is synchronous and optimised for single-thread execution. Both versions are written in the Scala programming language, a relatively new programming language for the Java Virtual Machine. Scala[6] is suited for both object-oriented and functional programming, supports parallel and asynchronous programming and is compatible with Java code and existing libraries.

Both versions are based on the concept of meeting arenas introduced in Section 3.1. Every agent is assigned with a solution to the optimisation problem, its fitness and some "life energy" (a single resource). The behaviour of the agents is the same in both versions (see Listing 1). They differ in how agents join arenas and how arenas execute meetings.

```
1  def chooseArena = energy match {
2    case 0 => deathArena
3    case e if e > threshold
4           => reproductionArena
5    case e => fightingArena
6  }
```

Listing 1: Agents choose an arena to join based on their current resources, in this case energy.

In this evolutionary multi-agent system, we use the following arenas:

- agents are removed from the system in the *death arena*
- agents compare their fitness in the *fighting arena*. Losers give some of their energy to the winners
- new agents are created in the *reproduction arena*. Children solutions are derived from their parents using variation operators. Parents give some of their energy to their children.

5.1. Asynchronous EMAS

This version is similar to the approach in frameworks like Jade, in which agents are the basic unit of concurrency. They are independent entities which do not directly expose state and can only query each other for information.

Agents and arenas have been implemented using the Akka[7] actor library. They are represented by actors which execute asynchronously and communicate through message passing. As such, agents can be mapped to threads in a very flexible way. Akka actors are handled by a component called the *dispatcher*. The dispatcher allows each actor in turn to process one or more messages from its mailbox. It is also used to execute asynchronous tasks.

The processing of a message or task can happen in any thread owned by the dispatcher, which behaviour is fully configurable. The dispatcher can use a single thread, a pool of threads or assign a separate thread to each actor. Akka ensures happens-before relationships between the processing of consecutive messages and preserves memory consistency.

After its previous meeting have ended, every agent chooses an arena and join it by sending a `JoinMeeting` message. Every arena has a fixed size and acts as a cyclic barrier: a meeting is triggered as soon as the capacity of the arena have been reached (see Listing 2). Multiple meetings may be happening at the same time, but every agent can only take part in one of them. When the meeting is finished, a `MeetingEnded` message is sent asynchronously to its participants so that they can choose a new arena to join.

```
1  def receive = {
2    case JoinMeeting =>
3      waitingRoom.add(sender)
4      if(waitingRoom.isFull()) {
5        val members = waitingRoom.flush()
6        performMeeting(members) andThen {
7          members foreach {
8            member => member tell MeetingEnded
9          }
10       }
11     }
12   }
```

Listing 2: Asynchronous arenas act as a cyclic barrier and trigger an asynchronous meeting as soon as they are full.

An additional mechanism, omitted above for clarity, triggers a meeting after some inactivity timeout. This may be beneficial for the algorithm (e.g., reproduction with mutation only) or help avoid deadlocks (when the number of agents in the environment is lower than the capacity of the arena).

Listing 3 shows an example of a meeting in the fighting arena. As the arena has no direct access to agents' state it needs to query them with the use of messages. A Scala feature known as *futures* and *for comprehension* allows to implement asynchronous and non-blocking meetings. The `askForFitness` and `getEnergyFrom` functions return a *future value* which will be *completed* only when all the members reply to messages. *For comprehension* composes these futures into a new one which is returned from the `performMeeting` function, allowing the installation of a *completion hook* (Listing 2, lines 6–9). The important thing is that the `performMeeting` function can return before the meeting has actually ended, so that another meeting may be triggered in the arena.

```
1  def performMeeting(members) = for(
2      fitnesses <- askForFitness(members);
3      val winner = zip(members, fitnesses)
4        .maxBy { (m, f) => f }
5        .map { (m, f) => m }
6      val losers = members - winner;
7      energies <- getEnergyFrom(losers)
8      ) yield winner tell ReceiveEnergy(energies.sum)
```

Listing 3: Non-blocking asynchronous fight using Scala futures and *for comprehension*.

5.2. Synchronous EMAS

In this version, agents are considered parts of the model rather than the implementation, like in Netlogo. As such, they are not represented as individual entities but as data structures.

[6] http://www.scala-lang.org/.
[7] http://akka.io/.

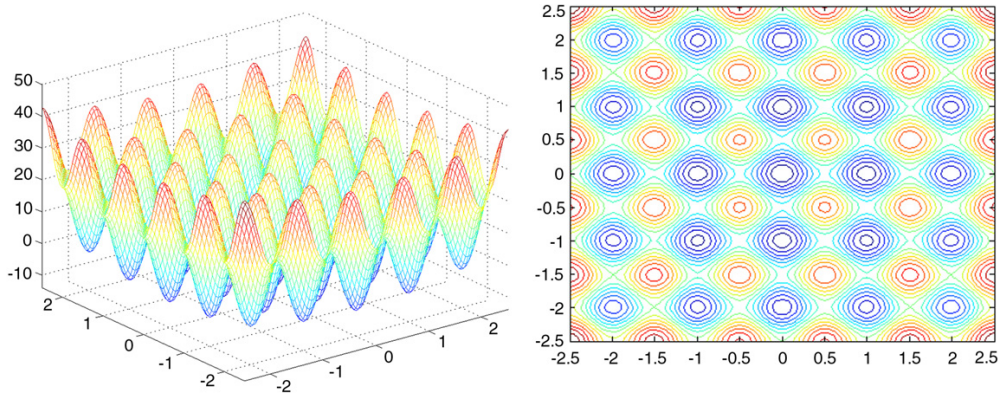


Fig. 3. The Rastrigin function in two dimensions.

Populations are collections of agents, processed step by step by arenas to yield new collections (see Listing 4). Agents are split into arenas (lines 2–4) and grouped accordingly to the arity of each arena (line 7). Finally, groups are processed by arenas and the agents resulting from each meeting are combined into a new population (lines 5–10).

```
1  def step(population) = {
2    val agentsInArenas = population groupBy { agent =>
3      agent.chooseArena
4    }
5    val newPopulation =  agentsInArenas flatMap {
6      (arena, agents) =>
7        agents grouped(arena.size) flatMap {
8          members => arena.performMeeting(members)
9        }
10   }
11   return newPopulation shuffled
```

Listing 4: Agents are split between arenas, grouped and processed. These action repeatedly transform the population.

The `performMeeting` method of each arena should in this case return a collection of agents representing the result of a meeting. These collections are merged into the new population. The Listing 5 shows the implementations of a synchronous fighting arena, which is similar but simpler than in the asynchronous version, as arenas can now have direct and synchronous access to the state of agents.

```
1  def performMeeting(members) = {
2    val winner = members maxBy {
3      agent => agent.fitness
4    }
5    val losers = members − winner
6    val energies = getEnergyFrom(losers)
7    winner.energy += energies.sum
8    return members
9  }
```

Listing 5: A synchronous fighting arena transforms its members by transferring energy from losers to winners.

The step function from Listing 4 could be executed in a simple loop. However, we used an Akka actor which repeatedly sends a Step message to itself, in order to minimise the performance impact of the Akka framework itself when comparing both versions.

It should be added that the structure of the synchronous version is similar to the MapReduce pattern and could be parallelised in a similar way. While this is a topic of the current research, we decided to stick to a possibly simple version in this work.

Table 1
EMAS parameters.

Initial-size	50
Initial-energy	10
Reproduction-threshold	10
Reproduction-transfer	5
Fight-transfer	10
Fight-arena-size	2
Migration-probability	0.001
Problem-size	100
Mutation-rate	0.1
Mutation-range	0.05
Mutation-probability	0.75
Recombination-probability	0.3

6. Experimental results

We carried out a series of experiments to measure the performance and scalability of the implementations described in the previous section. We applied the evolutionary multi-agent system to the optimisation task of finding the global minimum of the Rastrigin benchmark function (Eq. (1)), a highly multimodal function with many local optima and one global minimum equal 0 at $\bar{x} = 0$ (Fig. 3). We used a problem size (a dimension of the function) equal to 100.

$$f(x) = 10n + \sum_{i=1}^{n}(x_i^2 - 10\cos(2\pi x_i)). \quad (1)$$

The parameters used in our experiments are listed in Table 1.

The environment was initialised with *initial-size* agents, each given *initial-energy*. Agents were fighting on an arena of *fight-arena-size* capacity, transferring *fight-transfer* energy from a loser to a winner. As soon as an agent's energy exceeded *reproduction-threshold*, it entered a reproduction arena of size 2. Each pair of agents in the reproduction arena reproduced using a set of genetic operators described below, creating 2 new agents, each one given *reproduction-transfer* energy from one of their parents.

In the second stage of our experiments, at each step every agent had a *migration-probability* of migrating to some other environment. The target environment was chosen at random and including the original one.

In order to create new solutions to be assigned to newborn agents, the following genetic operators were used. Solutions were encoded as real-valued vectors. At each reproduction, crossover

Table 2
Final best fitness found in each of the models with a given number of cores. The asynchronous models were run 60 min, the synchronous one was run 10 min. The results are averaged over 30 runs.

	Cores				
	1	2	4	8	12
Own	22.1500	19.4800	12.7577	9.2573	3.6087
Pool	18.0173	17.6303	11.1574	5.4219	3.8016
Single	15.2845	19.2584	6.3323	8.1770	3.4879
Sync	0.0371	0.0398	0.0321	0.0186	0.0257

Table 3
Total number of fitness evaluations in each of the models with a given number of cores. The asynchronous models were run 60 min, the synchronous one was run 10 min. The results are averaged over 30 runs.

	Cores					
	1	2	4	8	12	
Own	1.0407	1.5252	1.6836	2.0900	2.9918	
Pool	1.4270	1.7994	2.0061	2.6123	2.8876	
Single	1.7423	1.4949	2.2076	2.2561	2.9611	$\times 10^7$
Sync	3.3296	3.1524	3.7204	5.6854	4.6629	

and mutation happened with respectively *recombination-probability* and *mutation-probability*. We used random average crossover, which consists in picking a random point in the hypercube defined by the parents genotypes. Every feature in the solution vector was mutated with probability *mutation-rate*. We used Gaussian mutation with *mutation-range* standard deviation.

6.1. Performance testing

In our performance testing, we distinguished four experimental scenarios. All of them share the same set of parameters listed above and have been run on Pl-Grid[8] infrastructure. We used nodes with an Intel Xeon X5650 2,66 GHz processor, with 1 GB of memory and a variable number of active cores (up to 12).

The first three scenarios correspond to the asynchronous implementation with an Akka dispatcher configured with respectively *own-thread*, *thread-pool* and *single-thread* policy (see Section 5.1). The fourth scenario is the synchronous implementation (Section 5.2).

Each scenario was repeated 30 times with different random generator seed values. The asynchronous models have been run for 60 min each, while the synchronous one only for 10 min.

We gathered two metrics: (a) the fitness of the best solution found so far at any given time, (b) the number of fitness function evaluations at any given time. The former metric shows the efficiency of the evolutionary algorithm itself and is also dependent on i.e. the parameters of the evolutionary operators. The latter reflects the number of agent meetings and only depends on the execution model and threading strategy. Of course, the dynamics of the underlying multi-agent system have an impact on the efficiency of the evolutionary algorithm.

The results in Tables 2 and 3 indicate that:

- there was no statistically significant difference in the performance of the asynchronous version using different thread policies (as verified by a two-sample Kolmogorov–Smirnov test with $p = 0.5$);
- the asynchronous versions greatly improved when given more cores …
- … but were *dramatically* worse than the synchronous version.

[8] http://www.plgrid.pl/en.

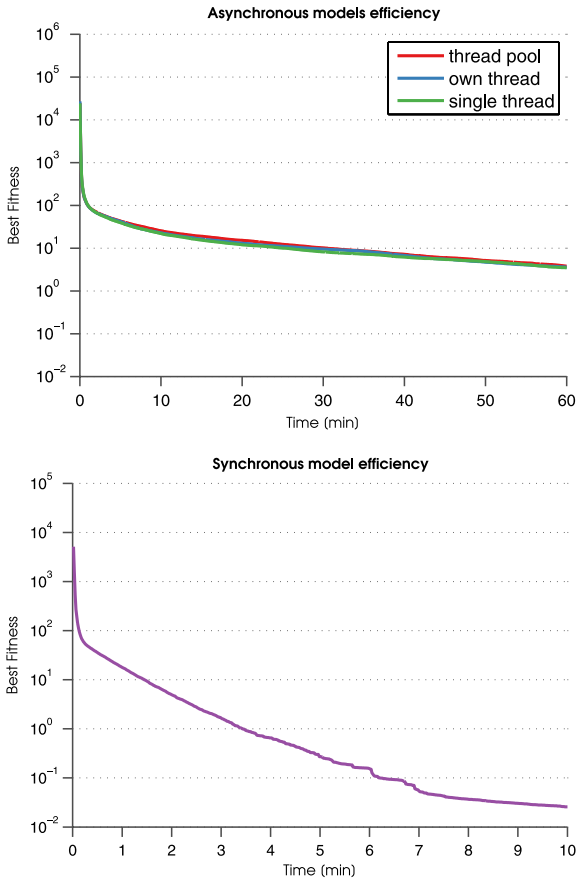


Fig. 4. Average best fitness found in each of the models after a given amount of time (for 12 cores used). Top—asynchronous models, bottom—synchronous model.

This difference in efficiency did not come from a flaw in the evolutionary algorithm itself, but rather from the underlying implementation model. The best asynchronous version only performed about 8×10^3 fitness evaluations per second, while the synchronous version did more than 7×10^4—nearly an order of magnitude faster. This efficiency gap can clearly be seen in Figs. 4 and 5.

Profiling data suggest that the asynchronous implementation was not idle or blocking on I/O, but rather very busy managing threads and passing messages between actors.

Fig. 6 shows the empirical distribution functions of the final fitness achieved in separate runs of each model. In many cases, the asynchronous version did converge to acceptable solutions (though given much more time). However, in many runs, they clearly needed more time. In contrast, all the runs of the synchronous version converged to the attraction basin of the global optimum (which corresponds in the case of the Rastrigin function to a fitness value lower than 1). It took an average of 132.13 s ($\pm$23.82 s), the empirical cumulative distribution is shown in Fig. 7.

6.2. Scalability testing

Having determined that the synchronous model is more efficient, we went on to test the scalability of the algorithm when new resources were added. We modified the implementation to

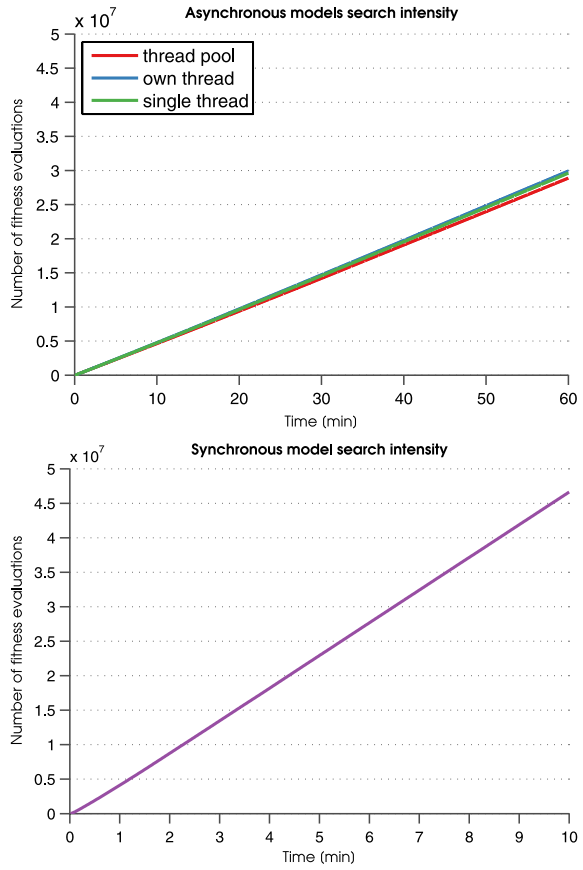


Fig. 5. Amount of fitness evaluations performed in each of the models after a given amount of time (for 12 cores used). Top—asynchronous models, bottom—synchronous model.

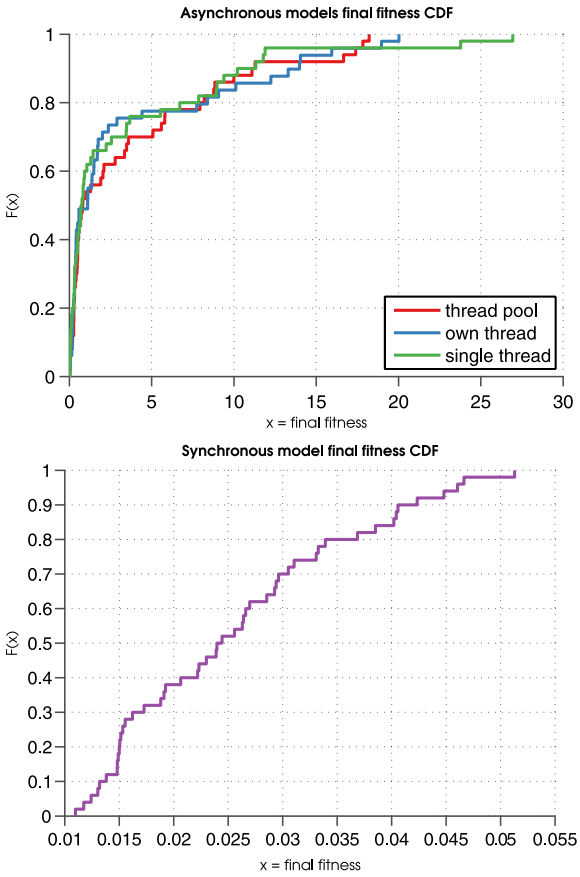


Fig. 6. Empirical cumulative distribution functions of the final fitness values found at the end of the computations (for 12 cores used). Top—asynchronous models, bottom—synchronous model.

simultaneously start a synchronous EMAS environment on many nodes in a cluster. The environments discovered each other in the cluster and connected, enabling the migration of agents between them.

We considered several scenarios with an increasing number of nodes. Each scenario was run for 10 min and repeated 30 times.

Fig. 8 shows the fitness of best solution found after a given time, averaged over all environments in a given scenario and all runs of the scenario. We can see that even adding a second node to the computation leads to significantly better results. Moreover, adding more nodes increases the convergence rate.

These results show that it is efficient to decompose multi-agent systems into distributed environments, like in the classical island model of evolutionary algorithms. However, the decentralised semantics of agent interaction may lead to more intelligent migration strategies, for example where agent populations automatically balance the load in the cluster.

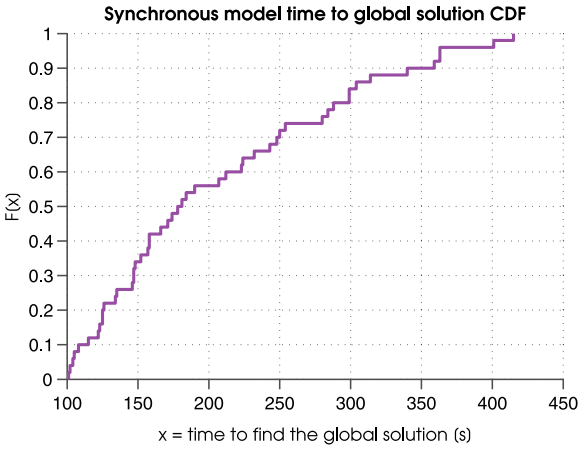


Fig. 7. Empirical cumulative distribution function of the time required to find the global solution in the synchronous model (for 12 cores used).

7. Conclusion

Metaheuristics can be valuable in decision support systems with time constraints. We discussed in this paper how the agent approach can be applied to these systems in order to build efficient and scalable software. We described the concept of evolutionary multi-agent systems, an example of a metaheuristic combining agent-based and evolutionary techniques.

The main goal of our work was to investigate the existing methods of building agent software and suggest new directions of development. In particular, we wanted to see if the dominant approach, which considers every agent as a unit of concurrency, is really efficient in computational intensive simulations.

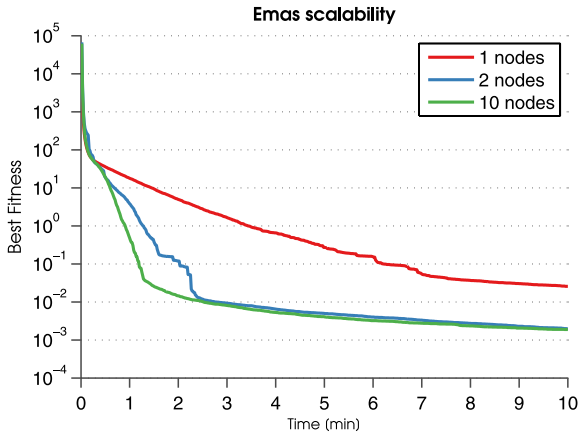


Fig. 8. Best fitness reached within a given time when using different amounts of nodes (parallel environments).

To that purpose, we developed two alternate implementations of an evolutionary multi-agent system which generalise the trends in the existing agent software. We introduced the idea of meeting arenas, an agent-based realisation of the Mediator design pattern which allow to efficiently structure multi-agent systems. We applied this concept to two versions of the algorithm: an asynchronous one, where every agent is a fully independent entity, and a synchronous one which treats agents as simple data structures.

Our experiments revealed that an asynchronous implementation, which may feel more *the agent way*, is nearly an order of magnitude less efficient than a synchronous one based on the same design. Several prototypes in other technologies supported these results. Further experiments on the synchronous implementation demonstrated that it can easily be scaled in a distributed setting, so that the efficiency of the algorithm increases when new nodes are added to the computation.

Therefore, we showed that the prevailing approach in existing agent platforms is not best suited in this particular class of applications. Instead, there is still room for improvement in the field of agent software dedicated to intensive simulations and computations. To that purpose, the concept of meeting arenas introduced in this paper allow to retain the expressive power of existing agent-based algorithms but can lead to much more efficient synchronous implementations.

In the nearest future, we want to see if concepts used in the functional programming paradigm could be more suited or more efficient in multi-agent software than the dominant object-oriented approach. Future work could also tell what kind of parallelism could be efficiently introduced in populations of agents. In particular, it would also be interesting to see how the Map/Reduce paradigm could be used to develop efficient massive multi-agent systems with hundreds of thousands of agents.

Acknowledgements

The research presented in the paper was partially supported by the European Commission FP7 through the project ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems, under contract no.: 288570 (http://paraphrase-ict.eu).

The research presented in the paper was conducted using PL-Grid Infrastructure (http://www.plgrid.pl/en).

The research presented in the paper was partially supported by the Polish Ministry of Science and Higher Education under AGH University of Science and Technology Grant 11.11.230.015.

References

[1] Z. Michalewicz, D. Fogel, How to Solve It: Modern Heuristics, Springer, 2004.
[2] R. Dreżewski, J. Sepielak, L. Siwik, Classical and agent-based evolutionary algorithms for investment strategies generation, in: A. Brabazon, M. O'Neill (Eds.), Natural Computing in Computational Finance, in: Studies in Computational Intelligence, vol. 185, Springer-Verlag, 2009, pp. 181–205.
[3] J. Koźlak, G. Dobrowolski, M. Kisiel-Dorohinicki, E. Nawarecki, Anti-crisis management of city traffic using agent-based approach, J. UCS 14 (2008).
[4] A. Ławrynowicz, A survey of evolutionary algorithms for production and logistics optimization, Res. Logist. Prod. 1 (2011).
[5] M. Wooldridge, N. Jennings, Intelligent agents, in: LNAI, vol. 890, Springer Verlag, 1995.
[6] Y. Luo, K. Liu, D. Davis, A multi-agent decision support system for stock trading, IEEE Netw. (2002) 20–27.
[7] S. Ossowski, A. Fernandez, J.M. Serrano, J. Perez-de-la Cruz, M. Belmonte, J. Hernandez, A. Garcia-Serrano, J. Maseda, Designing multiagent decision support system—the case of transportation management, in: Proc. of AAMAS, ACM Press, 2004, pp. 1468–1469.
[8] F. Bellifemine, A. Poggi, G. Rimassa, JADE: a FIPA2000 compliant agent development environment, in: Proceedings of the Fifth International Conference on Autonomous Agents, ACM, 2001, pp. 216–217.
[9] M.J. North, N.T. Collier, J. Ozik, E.R. Tatara, C.M. Macal, M. Bragen, P. Sydelko, Complex adaptive systems modeling with Repast Simphony, Complex Adapt. Syst. Model. 1 (2013) 3.
[10] O. Gutknecht, J. Ferber, The MadKit agent platform architecture, in: T. Wagner, O. Rana (Eds.), Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems, 2001.
[11] K. Piętak, A. Woś, A. Byrski, M. Kisiel-Dorohinicki, Functional integrity of multi-agent computational system supported by component-based implementation, in: Proceedings of the 4th International Conference on Industrial Applications of Holonic and Multi-Agent Systems, 2009.
[12] A. Byrski, R. Dębski, M. Kisiel-Dorohinicki, Agent-based computing in an augmented cloud environment, Comput. Syst. Sci. Eng. 27 (2012).
[13] Ł Faber, K. Pietak, A. Byrski, M. Kisiel-Dorohinicki, Agent-based simulation in AgE framework, in: A. Byrski, Z. Oplatkova, M. Carvalho, M. Kisiel-Dorohinicki (Eds.), Advances in Intelligent Modelling and Simulation, in: Studies in Computational Intelligence, vol. 416, Springer, Berlin, Heidelberg, 2012, pp. 55–83.
[14] A. Byrski, R. Dreżewski, L. Siwik, M. Kisiel-Dorohinicki, Evolutionary multi-agent systems, Knowl. Eng. Rev. 30 (2015) in press.
[15] D. Power, Decision Support Systems: Concepts and Resources for Managers, Quorum Books, 2002.
[16] Z. Michalewicz, Ubiquity symposium: evolutionary computation and the processes of life: the emperor is naked: evolutionary algorithms for real-world applications, Ubiquity 2012 (2012) 3:1–3:13.
[17] F. Glover, G.A. Kochenberger, Handbook of Metaheuristics, Springer, 2003.
[18] J. Dréo, A. Pétrowski, P. Siarry, E. Taillard, A. Chatterjee, Metaheuristics for Hard Optimization: Methods and Case Studies, Springer, 2005.
[19] E.-G. Talbi, Metaheuristics: From Design to Implementation, Wiley, 2009.
[20] M. Kisiel-Dorohinicki, G. Dobrowolski, E. Nawarecki, Agent populations as computational intelligence, in: L. Rutkowski, J. Kacprzyk (Eds.), Neural Networks and Soft Computing, Physica Verlag, 2002, pp. 608–614.
[21] K. Cetnarowicz, M. Kisiel-Dorohinicki, E. Nawarecki, The application of evolution process in multi-agent world (MAW) to the prediction system, in: M. Tokoro (Ed.), Proc. of the 2nd Int. Conf. on Multi-Agent Systems, ICMAS'96, AAAI Press, 1996.
[22] A. Byrski, Tuning of agent-based computing, Comput. Sci. 14 (3) (2013).
[23] A. Byrski, M. Kisiel-Dorohinicki, Immunological selection mechanism in agent-based evolutionary computation, in: Proc. of IIS: IIPWM'05 Conference: Gdansk, Poland, Advances in Soft Computing, Springer Verlag, 2005, pp. 411–415.
[24] D. Krzywicki, Niching in evolutionary multi-agent systems, Comput. Sci. 14 (1) (2013).
[25] G. Danoy, P. Bouvry, O. Boissier, A multi-agent organizational framework for coevolutionary optimization, in: Transactions on Petri Nets and Other Models of Concurrency IV, Springer, 2010, pp. 199–224.
[26] M. Milano, A. Roli, Magma: a multiagent architecture for metaheuristics, IEEE Trans. Syst. Man Cybern. Part B, Cybern. 34 (2004) 925–941.
[27] E. Noda, A.L. Coelho, I.L. Ricarte, A. Yamakami, A.A. Freitas, Devising adaptive migration policies for cooperative distributed genetic algorithms, in: 2002 IEEE International Conference on Systems, Man and Cybernetics, Vol. 6, IEEE, 2002, pp. 6–11.
[28] D. Meignan, J.-C. Créput, A. Koukam, An organizational view of metaheuristics, in: First International Workshop on Optimisation in Multi-Agent Systems, AAMAS, Vol. 8, 2008, pp. 77–85.
[29] L. Braubach, A. Pokahr, Developing distributed systems with active components and jadex, Scalable Comput. Pract. Exp. 13 (2012) 100–119.
[30] C. Frantz, M. Nowostawski, M. Purvis, Dynamic ad hoc coordination of distributed tasks using micro-agents, in: Agents in Principle, Agents in Practice, 2011, pp. 275–286.
[31] M. Ughetti, T. Trucco, D. Gotta, Development of agent-based, peer-to-peer mobile applications on ANDROID with JADE, in: 2008 The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008, pp. 287–294.

D. Krzywicki obtained his M.Sc. in 2012 at AGH University of Science and Technology in Cracow and is currently a Ph.D. student at the Department of Computer Science of AGH-UST. His research interests include agent-based computations, functional programming and distributed systems.

Ł. Faber obtained his M.Sc. in 2012 at AGH University of Science and Technology in Cracow and is currently a Ph.D. student at the Department of Computer Science of AGH-UST. His research interests include agent-based modelling and distributed systems.

A. Byrski obtained his Ph.D. in 2007 and D.Sc. (habilitation) in 2013 at AGH University of Science and Technology in Cracow. He works as an assistant professor at the Department of Computer Science of AGH-UST. His research focuses on multi-agent systems, biologically-inspired computing and other soft computing methods.

M. Kisiel-Dorohinicki obtained his Ph.D. in 2001 and D.Sc. (habilitation) in 2013 at AGH University of Science and Technology in Cracow. He works as an assistant professor at the Department of Computer Science of AGH-UST. His research focuses on intelligent software systems, particularly using agent technology and evolutionary algorithms, but also other soft computing techniques.

3.3. Execution model based on actors

The second execution model I examined was based on the actor Model [28] of concurrency. In simplified terms, an actor can be understood as an addressable process which requires few resources, can execute independently and potentially in parallel with other actors. Each actor may have mutable state, but can only affect other actors by sending them messages.

In this execution model, every agent, every arena, and every interaction is represented by a separate actor (Figure 3.5). Within their actors, agents compute the behavior function to determine their next behavior and send a summary of their state to the arena actor corresponding to the chosen behavior.

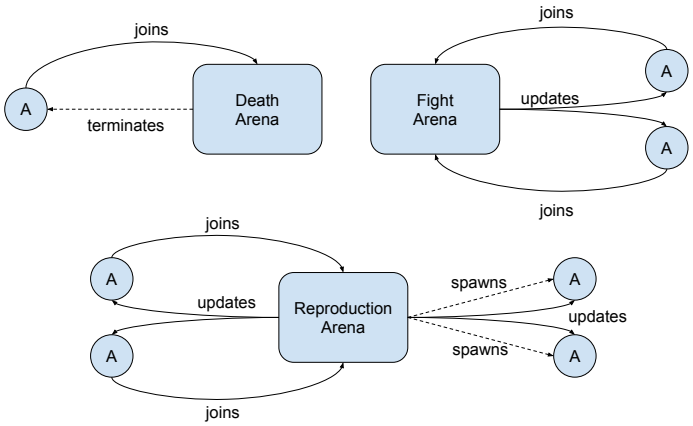


Figure 3.5. In the execution model based on actors, every agent and every arena is mapped into an actor and communicate through message passing. Agents repeatedly compute the behavior function to decide which arena to join. The meeting function is computed within arenas when enough agents have joined. As a result of the meeting, agents (and the underlying actor) can be spawned, updated or terminated. The supervision hierarchy of the actor is not represented in this figure.

The arena actors receive incoming agents. When the capacity of the arena is reached, or when a timeout happens, a meeting is triggered. As a result, depending on the type of behavior, new agents may be born, existing agents may be modified or die. As necessary, the meeting arena will spawn or terminate actors, then distribute new agents state across those actors for the behavior function to be computed again.

This execution model is highly concurrent, as the computation of the behavior and meeting functions in different agents and arenas can happen in any order. There is only a partial causal order in the system, as defined by the exchange of messages [29]. This execution model is also highly parallel and can easily be distributed, just as the underlying actor model. However, if the dispatching of actors is being done in parallel, the computation becomes non-deterministic, and the concurrent properties of the algorithm cannot be easily controlled for.

For the sake of simplicity, the supervision hierarchy of the actors is not detailed on Fig 3.5. In practice, there is a parent actor responsible for supervising a set of actors and arenas, and arenas delegate to

their parent for the spawning of new actors. This parent actor makes it very simple to model multiple agent environments, as in the island model. Every environment is defined by a disjoint set of arenas across which agents can interact. For an agent, migrating to a different environment only means to change its supervisor and the set of arenas which it will join.

The following to publications describe implementations of the actor-based model in different languages (Scala and Erlang). They show that the general properties of the model as observed during experimental validation are consistent in both cases, despite the differences in the underlying technologies.

Journal of Computational Science 11 (2015) 153–162

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Massively concurrent agent-based evolutionary computing

D. Krzywicki, W. Turek, A. Byrski*, M. Kisiel-Dorohinicki

AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Poland

ARTICLE INFO

Article history:
Received 15 October 2014
Received in revised form 29 June 2015
Accepted 20 July 2015
Available online 29 July 2015

Keywords:
Multi-agent systems
Evolutionary computing
Functional programming

ABSTRACT

The fusion of the multi-agent paradigm with evolutionary computation yielded promising results in many optimization problems. Evolutionary multi-agent systems (EMAS) are more similar to biological evolution than classical evolutionary algorithms. However, technological limitations prevented the use of fully asynchronous agents in previous EMAS implementations. In this paper we present a new algorithm for agent-based evolutionary computations. The individuals are represented as fully autonomous and asynchronous agents. An efficient implementation of this algorithm was possible through the use of modern technologies based on functional languages (namely Erlang and Scala), which natively support lightweight processes and asynchronous communication. Our experiments show that such an asynchronous approach is both faster and more efficient in solving common optimization problems.

1. Introduction

Biological systems are asynchronous by nature. This fact is not always considered in biologically inspired computing methods (e.g. metaheuristics, such as evolutionary algorithms [24]). These systems usually use notions such as "discrete" generations, loosing such concepts as parallel ontogenesis, or lack of global control. Nevertheless, such computing systems have proven to be effective in different optimization problems. Moreover, some of them can be mathematically proven to work in terms of asymptotic stochastic guarantee of success (cf. works of Vose on simple genetic algorithm [32]).

Agent-oriented systems should also be asynchronous by nature, as they are inspired by social or biological systems. Over the last decade, our group has worked on the design and development of decentralized evolutionary computations [2] in the form of evolutionary multi-agent systems [9]. EMAS is a hybrid metaheuristic which combines multi-agent systems with evolutionary algorithms. A dedicated mathematical formalism, based on Markov chains (similar to Vose's approach) was constructed and analysed [4], showing that EMAS may be also treated as a general-purpose optimization system. Besides that, a number of other formalisms along with dedicated frameworks implemented in different programming languages (like Java, Scala or Python) were developed (see, e.g. [6,29,7]).

The concept of hybridization of agent-based systems with evolutionary techniques can be implemented in different ways, especially with regard to asynchronicity and concurrency, as well as distribution and parallelism. There are a number of popular agent-oriented frameworks which offer asynchronously communicating agents (such as Jadex [27], JADE [1] or MadKit [15]). However, they all share similar properties, such as heavyweight agents, at least partial FIPA-compliancy (JADE) or a BDI model (Jadex). It is also common for each agent to be executed as a separate thread (e.g. in JADE). These traits are indeed appropriate to model flexible, coarse-grained, open systems. However, evidence suggest they are not best suited for closed systems with homogeneous agents nor for fine-grained concurrency with large numbers of lightweight agents, which are both common in biologically inspired population-based computing systems [31].

Therefore, dedicated tools for the above-mentioned class of agent-based systems have been constructed over the last 15 years. One of the successful implementations is the AgE platform,[1] which supports phase-model and hybrid concurrency features (parts of the system are concurrent and parts are implemented as sequential processes). The AgE platform also has other advantages, such as significant support for reuse and flexible configuration management. Dedicated AgE implementations were constructed using Java, NET and Python technologies.

However, the renewed interest in functional programming and languages such as Erlang and Scala brought new possibilities in terms of concurrent programming support. In a recent paper

* Corresponding author.
E-mail addresses: daniel.krzywicki@agh.edu.pl (D. Krzywicki), wojciech.turek@agh.edu.pl (W. Turek), olekb@agh.edu.pl (A. Byrski), doroh@agh.edu.pl (M. Kisiel-Dorohinicki).

[1] http://age.agh.edu.pl.

http://dx.doi.org/10.1016/j.jocs.2015.07.003
1877-7503/

[21], we proposed a promising new approach to these kinds of agent-based algorithms. We used Erlang lightweight processes to implement a fined grained multi-agent system and bring more asynchronicity into usually synchronously implemented agent actions and interactions.

In this paper, we present a significant progress over the research presented in [21], by extending our experiments to a Scala-based implementation (in addition to the Erlang-based one) and comparing these two approaches. The previous Erlang asynchronous implementation showed a small but statistically significant improvement in terms of efficiency, compared to a synchronous version. The results for the new Scala implementation provide much stronger evidence for the superiority of a massively concurrent EMAS implementation over a traditional, synchronous one.

In the next sections we present the current state-of-the-art and introduce concepts of evolutionary multi-agent systems. Then, we describe the implementation of the synchronous and asynchronous versions of our algorithm, followed by our experimental settings and results. We end with a discussion of our results and conclude the paper with possible opportunities for future work.

2. Large-scale agent-based systems

The development of the software agent concepts and the theory of multi-agent systems took place in the last decades of the 20th century. At this point in time, the software engineering domain was strongly focused on the popularization of the object-oriented paradigm. As a result, the majority of agent systems and platforms for agent systems development was based on imperative languages with shared memory. This approach is in opposition to the assumptions of agent systems, which are based on the concept of message passing, communication between autonomous execution threads and do not allow any explicit shared state.

Implementations of message passing concurrency in object oriented technologies resulted in significant limitations in both scale and performance of the developed solutions. The evaluation presented in [31] shows that the most popular agent development platforms (JADE and Magentix) are limited to several thousands of simultaneous agents on a single computer. The limit was caused by the method used for implementing concurrently executing code of agents – each agent required a separate operating system thread. This situation inhibited the development of large scale multi-agent systems for long time.

Current trends in the domain of programming languages development focus on the integration of concepts from different paradigms and on the development of new languages dedicated for particular purposes. The renewed interest in the functional paradigm seems very significant in the context of agent systems development. The agent system for human population simulation, presented in [12], was implemented in Haskell. The authors emphasize that the source code was very short in respect to the complex functionality of the system. The discussion presented in [14] focuses on the language features of Haskell in the context of agent systems. The authors show the usefulness of algebraic data types, roles and sessions in implementation of multi-agent algorithms.

The popularization of the functional paradigm concepts is mostly caused by difficulties in efficient usage of multi-core CPUs in languages implementing a shared-memory concurrency model. The need of synchronization of all the operations on shared memory effectively prevents the applications from scaling on higher numbers of cores. The issue does not exist in the message passing concurrency model, which allows massively concurrent applications to run effectively on parallel hardware architectures. This fact triggered the development of languages and runtime environments which efficiently implement the message passing concurrency model. There are currently two major technologies of this kind being successfully used in industrial applications: Erlang and Scala with the Akka library.

The concurrency model implemented by these technologies is based on the same assumptions as in the case of software agents. Therefore, these technologies are a very good basis for large scale and high performance multi-agent systems. Recent example of a Scala-based implementation of a custom multi-agent architecture can be found in [22], where the authors present a system capable of processing and storing a large amount of messages gathered from sensing different devices.

Erlang technology has been found useful in this kind of applications much earlier. In 2003 the first agent development platform, called eXAT (*erlang eXperimental Agent Tool*), has been presented in [11]. The goal of this platform was to test the feasibility of using functional programming languages as a development tool for FIPA-compliant agents. An agent in eXAT is an actor-based, independent entity composed of *behaviours*, which represent the functionality of an agent as a finite-state machine. Transitions between states are triggered by changes in the knowledge-base facts or by external messages. The original version of eXAT does not support agent migrations, however there is a version supporting this functionality [25].

eXAT platform overcomes the basic limitations of Java-based solutions. eXAT agents are based on Erlang lightweight processes, which can be created in millions on a single computer. Although the platform never became a mainstream tool, it should be noticed that it was the first environment which allowed to test the behaviour of large scale systems with truly parallel agents.

Recent years brought different solutions based on Erlang technology, like the eJason system [10]. eJason is an Erlang implementation of Jason, which is a platform for the development of multi-agent systems developed in Java. The reason for rewriting the Jason platform in Erlang, pointed by the authors, are significant similarities between Jason agents and Erlang processes and the high performance and scalability of Erlang processes implementation.

The ability to build and test massively concurrent agent-based systems opens new possibilities of research in this domain. The algorithm presented in this paper is made possible by the high-performance implementations of a message-passing concurrency model offered by Erlang and Scala.

3. Evolutionary multi-agent systems (EMAS)

Generally speaking, evolutionary algorithms are usually perceived as universal optimization-capable metaheuristics (cf. theory of Vose [32]). However, the classical designs of evolutionary algorithms (such as simple genetic algorithm [13], evolution strategies etc. [30]) assume important simplifications of the underlying biological phenomena. Such simplification mainly consists in avoiding direct implementation of such phenomena observed in real-world biological systems, as dynamically changing environmental conditions, a dependency on multiple criteria, the co-evolution of species, the evolution of the genotype–phenotype mapping, the assumption of neither global knowledge nor generational synchronization.

Of course, it does not mean that they are wrong per-se, as they have clearly proven themselves in solving difficult problems. However, there is still room for improvement, and the No Free Lunch Theorem [33] reminds us that the search for new optimization techniques will always be necessary.

One of the important drawbacks of classical evolutionary techniques is that they work on a number of data structures

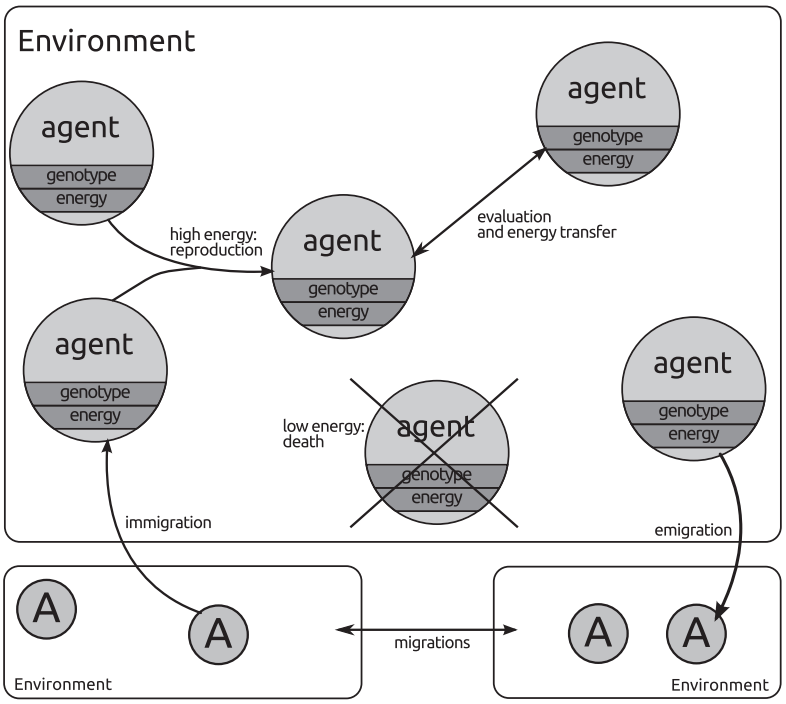


Fig. 1. Structure and behaviour of EMAS operation.

(*populations*) and repeat in cycles (*generations*) the same process of selecting parents and producing offspring using variation operators. Such an approach makes it difficult to implement large-scale, parallel implementations of evolutionary algorithms. Only trivial approaches to the parallelization of such algorithms were proposed (e.g. the master–slave model or parallel evolutionary algorithm [8]).

For the over 10 years, our group tried to overcome some of these limitations by working on the idea of decentralised evolutionary computations [2], namely evolutionary multi-agent systems (EMAS) [9]. EMAS is a hybrid meta-heuristics which combines multi-agent systems with evolutionary algorithms. The basic idea of EMAS consists in evolving a population of agents (containing the potential solutions to the problem in the form of genotypes). The agents are capable of doing different actions, communicating among themselves and with the environment, in order to find the optimal solution of the optimization problem.

According to classic definitions (cf. e.g., [17]) of a multi-agent system, there should not be global knowledge shared by all of the agents. They should remain autonomous without the need to create any central authorities. Therefore, evolutionary mechanisms such as selection needs to be decentralized, in contrast with traditional evolutionary algorithms. Using agent terminology, one can say that selective pressure is required to *emerge* from peer to peer interactions between agents instead of being globally driven.

Thus, selection in EMAS is achieved by introducing a non-renewable resource, called life-energy. Agents receive part of the energy when they are introduced in the system. They exchange energy based on the quality of their solution to the problem: worse agents move part of their energy to the better ones. The agents reaching certain energy threshold may reproduce, while the ones with low amount of energy die and are removed from the system. We show the principle of these operation in Fig. 1. For more details on EMAS design refer to [2].

Up till now, different EMAS implementations were applied to different problems (global, multi-criteria and multi-modal optimization in continuous and discrete spaces), and the results clearly showed superior performance in comparison to classical approaches (see [2]). It is to note, that besides acclaimed benchmark problems, as multi-modal functions [26,5,3], and discrete benchmarks with clear practical application, like Low Autocorrelation Binary Sequence or Golomb Ruler problem [19,18] EMAS was also successfully applied to solving selected inverse problems [34,28] leveraging its capability of quite low computational cost evaluated as number of fitness function calls, compared to other classic approaches.

During the last years, we made also several approaches to construct efficient software targeted at variants of EMAS and at agent-based computing in general. We implemented dedicated tools in order to prepare fully fledged frameworks convenient for EMAS computing (and several other purposes, as agent-based simulation). First, we focused on implementing decentralized agent behaviour, and the outcome were several fully synchronous versions, resulting in the implementation of the AgE platform.[2] In this implementation we applied a phase-model of simulation, efficiently implementing such EMAS aspects as decentralized selection. We also supported the user with different component-oriented utilities, increasing reuse possibilities and allowing flexible configuration of the computing system.

In a recent paper [21], we have presented a promising new approach to these kinds of algorithms. We used Erlang lightweight processes to implement a fine-grained multi-agent system. Agents are fully asynchronous and autonomous in fulfilling their goals, such as exchanging resources with others, reproducing or being removed from the system. Agents are able to coordinate their behaviour with the use of mediating entities called meeting arenas.

This approach brings us closer to the biological origins of evolutionary algorithms by removing artificial generations imposed by

[2] http://age.agh.edu.pl.

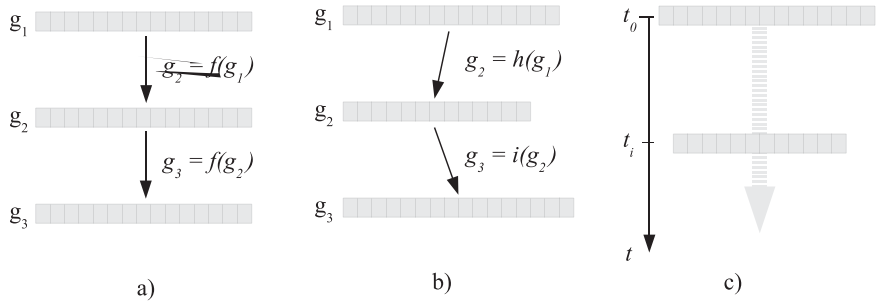


Fig. 2. Concept of generation-oriented and generation-free evolutionary computing.

step-based implementations. We show the concept of generation-oriented and generation-free computing in Fig. 2.

The first case (*a*) shows the classical approach, consisting in transforming a population of individuals by applying a stochastic state transition function. This function is usually composed of some predefined operators, such as selection, crossover and mutation. The second case (*b*) shows the EMAS approach (as it was realized in AgE platform), where different transformation functions are applied as the results of agent actions. This model still assumes the existence of generations, but on a technical level, as a result of a step-based simulation. In fact, both above cases use discrete-time based simulation.

In contrast, the third case (*c*) is a nearly continuous-time simulation (if we disregard the discrete nature of the machine itself). In this model, all agents may initiate actions at any possible time. The process scheduler makes sure they are given computing resources when they need them.

4. Massively concurrent EMAS implementation

The system presented in this work has been implemented in Erlang and Scala, as the lightweight concurrency model provided by these technologies is well suited for creating large-scale multi-agent systems. The implementation focuses on comparing different computational models in terms of their features and efficiency. The rest of this section describes the algorithms used in our evolutionary multi-agent system, the model of agents interactions and different implementations.

4.1. Principle of system operation

Every agent in the system is characterized by a vector of real values representing potential solution to the optimization problem. The vector is used for calculating the corresponding fitness value. The process of calculating the fitness value for a given solution is the most expensive operation. It is executed each time a new solution is generated in the system.

Emergent selective pressure is achieved by giving agents a piece of non-renewable resource, called energy [2]. An initial amount of energy is given to a newly created agent by its parents. If the energy of two agents is below a required threshold, they fight by comparing their fitness value – the better agent takes energy from the worse one. If an agent looses all its energy, it is removed from the system. Agents with enough energy reproduce and yield new agents. The genotype of the children is derived from their parents using variation operators. The number of agents may vary over time, however the system remains stable as the total energy remains constant.

As in the case of other evolutionary algorithms, the population of agents can be split into separated sub-populations. This approach helps preserving population diversity by introducing allopatric speciation and can also simplify parallel execution of the algorithm. In our case the sub-populations are called *islands*. Information can be exchanged between the islands through agent migrations.

4.2. Arenas

An efficient implementation of meetings between agents is crucial for the overall performance of the algorithm. The meetings model has a significant impact on the properties of the algorithm, on its computational efficiency and on its potential for parallel execution.

A general and simple way to perform meetings is to shuffle the list of agents and then process pairs of agents sequentially or in parallel. However, this approach has several limitations:

- The whole population must be collected in order to shuffle agents and form the pairs. This approach is inappropriate in an algorithm which should be decentralized by nature.
- Agents willing to perform different actions can be grouped together – all combinations of possible behaviours must be handled.

In our previous work [20], a different approach was proposed. We proposed to group agents willing to perform the same action in dedicated *meeting arenas*, following the Mediator design pattern. Every agent enters a selected arena depending on its amount of energy. Arenas split incoming agents into groups and trigger the actual meetings (see Fig. 3). Each kind of agent behaviour is represented by a separate arena.

Therefore, the dynamics of the multi-agent system are fully defined by functions. The first function represents agent behaviour, which chooses an arena for each agent. The second function represents the meeting operation which is applied in every arena.

This approach is similar to the MapReduce model, where arena selection corresponds to mapping and meeting logic to reduce operation. The pattern is very flexible, as it can be implemented

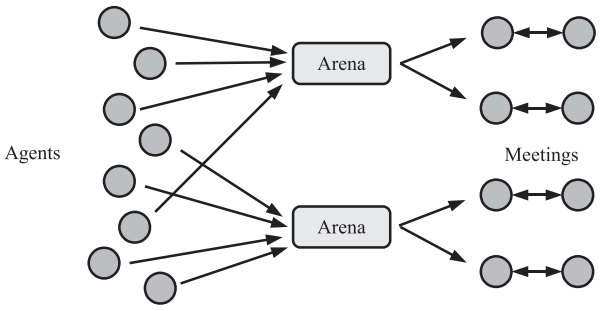


Fig. 3. Meeting arenas group similar agents and coordinate meetings between them.

in both a centralized and synchronous way or a decentralized and asynchronous one.

4.3. Sequential implementation

The sequential version of the presented multi-agent system is implemented as a discrete event simulation. In each step the *behaviour* function (see Listing 1) divides the population of agents into groups corresponding to available arenas.

Listing 1. In every step, agents choose an arena based on their current state

```
1  def behaviour (a: Agent) = a. energy match {
2    0 => death
3    x if x > 10 => reproduction
4    x => fight
5  }
```

Agents are first grouped according to their chosen arena and then a meeting function is applied on each such partition of the population. The partitions can be further subdivided into pairs of meeting agents and processed by applying a different meeting function which depends on the type of the arena (see Listing 2). Every meeting results in a group of agents. The group can contain some new agents created as a result of reproduction and some with their state changed (e.g. by transferring energy). Some agents may be removed from the group if their energy equals 0. Resulting groups are merged in order to form the new population, which is randomly shuffled before the next step.

Listing 2. Depending on the type of the arena, a different meeting happens, which transforms the incoming subpopulation of agents. The death arena simply return an empty sequence. Other arenas shuffle incoming agents, group them into pairs and apply a binary operator on every pair, concatenating results.

```
1  def meeting (arena: Arena, agent s: Seq [Agent]) =
2    arena match {
3      death =>
4        Seq. empty [Agent]
5      reproduction =>
6        agents
7          .shuffle
8          .grouped (2)
9          .flatMap (doReproduce)
10     fight =>
11       agents
12         .shuffle
13         .grouped (2)
14         .flatMap (doFight)
15 }
```

If several islands are considered, each is represented as separate lists of agents. Migration between islands is performed at the end of each step by moving some agents between lists.

4.4. Hybrid implementation

The introduction of coarse-grained concurrency in such a multi-agent system is rather straightforward. In our second implementation every island is assigned to a separate Erlang process/Scala actor responsible for executing the loop of the sequential algorithm described above. Islands communicate through message-passing, no other synchronization is needed.

The most significant difference regards agent migration. The behaviour function from Listing 1 is modified by adding a migration action which is chosen with some fixed, low probability. The migration process is performed by a dedicated migration arena present on every island.

The migration arena removes agent from the local population and forwards it agent to a selected island chosen according to some topology and migration strategy. In every step the processes responsible for executing islands loop incorporates the incoming agents into their population.

4.5. Concurrent implementation

The first two implementations presented so far do not require massively concurrent execution, as the agents are represented as data structures processed sequentially by islands and arenas. Such approach does not reflect the autonomy of entities in the population and the true dynamics of relation between the entities, as each agent is forced to perform exactly one operation in each step of the algorithm.

In order to achieve asynchronous behaviours of agents in the population, every agent and every arena has been implemented as a separate process/actor which communicates with the outside world only through message passing. The algorithm becomes fully asynchronous, as every agent acts at its own pace and there is no population-wide step. Meeting arenas are especially useful in this implementation, as they greatly simplify communication protocols.

The algorithm of each agent is relatively simple. Depending on its current energy, every agent selects the action to perform and sends a message to the appropriate arena. Afterwards it waits for a message with the results of the meeting.

As soon as enough agents gather in an arena, a meeting is triggered. As a result, new agents may be created and existing agents may be killed or replied with a message containing their new state.

Islands are logically defined as distinct sets of arenas. Each agent knows the addresses of all the arenas defining a single island, therefore it can only meet with other agents sharing the same set of arenas. Fights and reproductions arenas behave just as described in the previous version of the algorithm.

This migration process is greatly simplified in this version. Migrating an agent simply means changing the arenas it meets on. Migration arenas choose an island according to specified topology and send the addresses of the corresponding arenas back to the agent. The agent updates the set of arenas available to itself and resumes its behaviour. As it will now be able to meet with a different set of agents, it has indeed migrated.

The implementations in Erlang and Scala are relatively similar, as both realize exactly the same algorithms. In case of first two approaches (sequential and hybrid) no differences in behaviour of the implemented system was expected. On the other hand the execution of massively concurrent version can be significantly dependent on scheduling mechanisms implemented by the underlying process management system. The time of activities performed by each agent depends on the provided CPU access.

The details of process scheduling mechanisms implemented by Erlang and Scala (Akka) are slightly different. Erlang was designed as a soft real-time platform, which means that each a process can be preempted in every moment in time, and none can claim more computational time than the others. The model implemented by Scala is more suited for reactive, message-driven programming, where a process can use CPU until it finished processing current messages. These tiny differences can influence the overall performance of the implemented systems but also can result in different behaviour of the populations.

Another difference is memory management. In Erlang, every process owns a separate stack and the content of messages needs to be copied between process memories. Scala uses the Java Memory Model and adds a message-passing layer on top of shared memory. The Akka actor library follows the principle “going from remote to local by way of optimization”, therefore communication

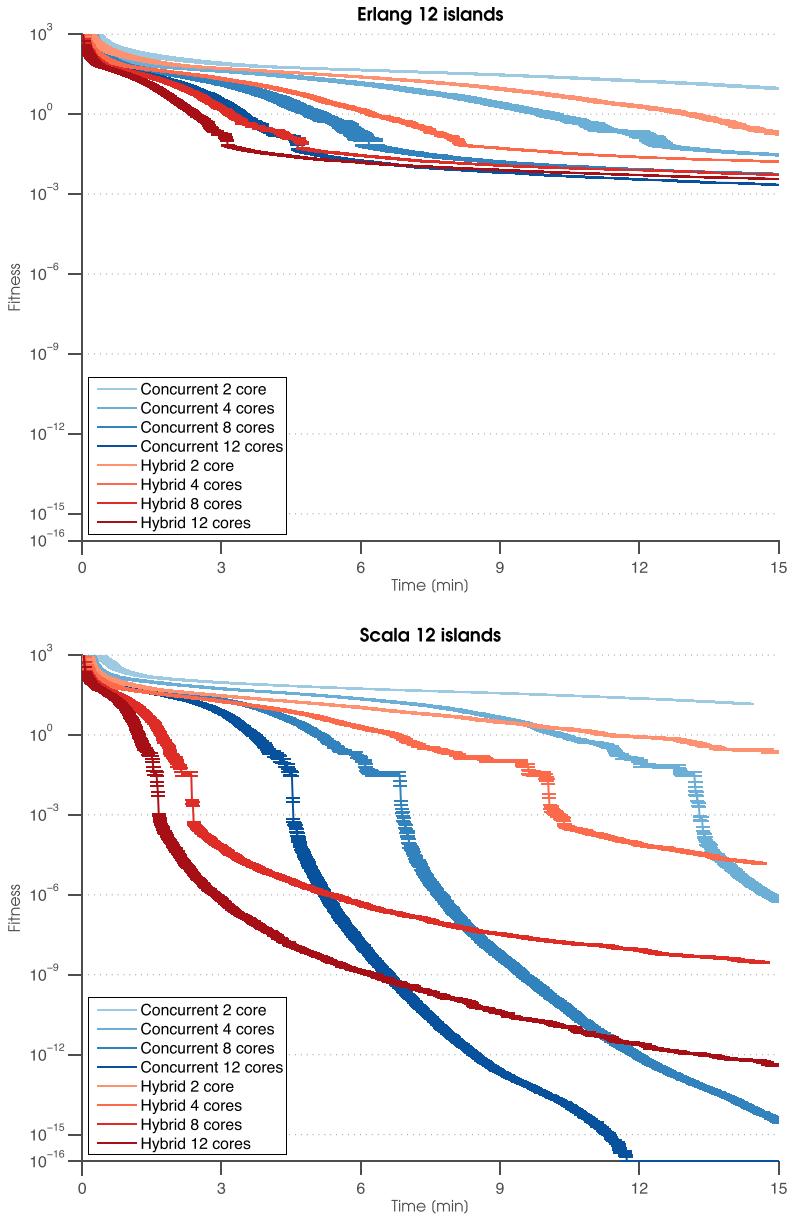


Fig. 4. Best fitness ever over time, depending on the model and the number of cores. Stripes indicate 95% confidence intervals. 10^{-16} constant has been added to the fitness values to visualize the global optimum.

within a single Java Virtual Machine is very memory efficient, as immutable messages are safe to be shared between the sender and receiver.

5. Methodology and results

We used our multi-agent system to minimize the Rastrigin function, a common benchmarking function used to compare evolutionary algorithms. This function is highly multimodal with many local minima and one global minimum equal 0 at $\vec{x} = 0$. We used a problem size (the dimension of the function) equal to 100, in a domain equal to the hypercube $[-50, 50]^{100}$.

The simulations were run on Intel Xeon X5650 nodes provided by the PI-Grid[3] infrastructure at the ACC Cyfronet AGH.[4]. We used up to 12 cores and 1 GB of memory.

We tested the alternative approaches described in the previous section, implemented in both Erlang and Scala. The experiments for hybrid and concurrent models were run on 1, 2, 4, 8 and 12 cores. A sequential version in each language was also run on 2 cores (the second core being used for logging an management) – more cores did not improve the results.

[3] http://www.plgrid.pl/en.
[4] http://www.cyfronet.krakow.pl/en/.

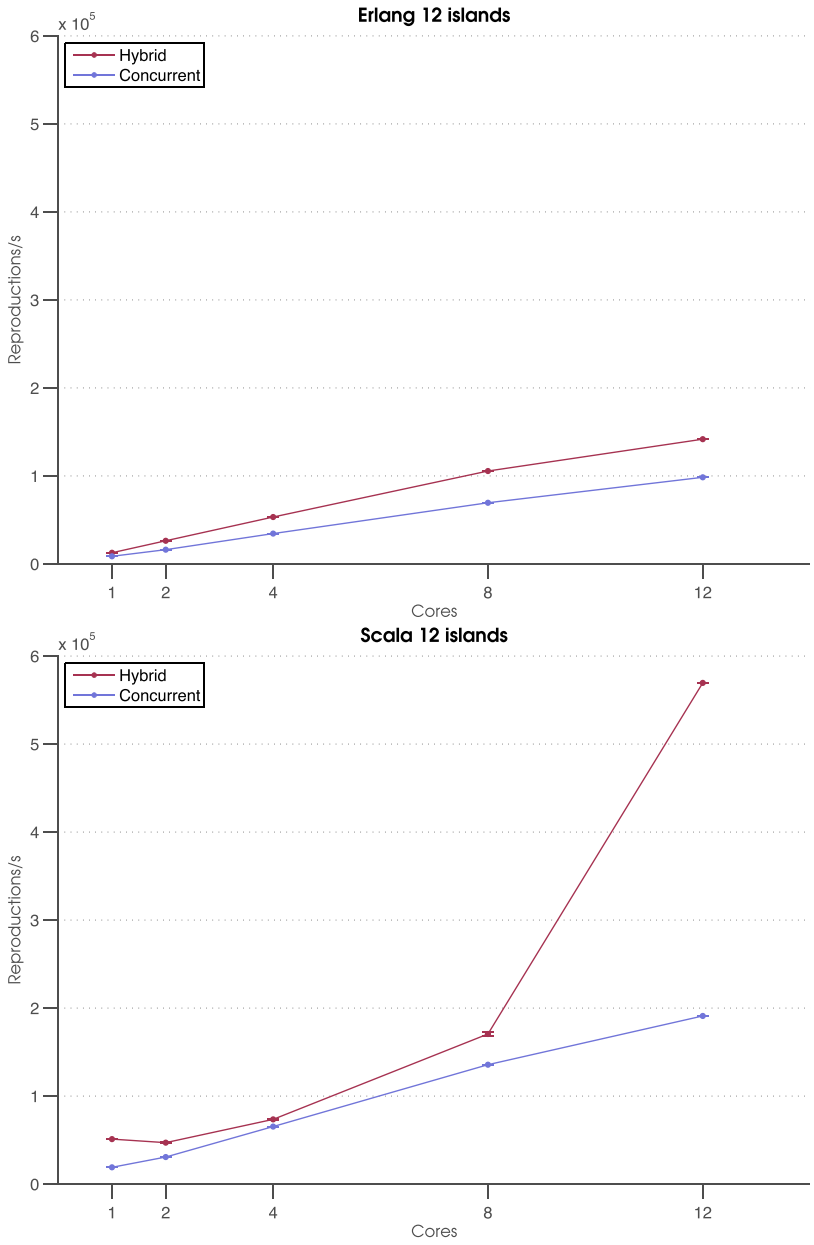


Fig. 5. The relation of the number of reproductions per second to the number of cores, for each model and implementation, along with 95% confidence intervals.

Every experiment lasted 15 min and was repeated 30 times in order to obtain statistically significant results. The results further below are averaged over these 30 runs.

The hybrid model does not benefit from a number of cores higher than the number of islands. As we had 12 cores at our disposal, we used 12 islands in every experiment. Migration destinations were chosen at random in a fully connected topology.

Results. We examined our models under two criteria: how well the algorithm works and how fast the meeting mechanism is.

We assessed the quality of the algorithm by recording: the fitness of the best solution found so far at any given time on any island (see Fig. 4).

We estimated the speed of the models by counting the amount of agent meetings performed in a unit of time. These numbers appeared to be proportionally related across different arenas. Therefore, we only consider below the amount of reproductions per second (see Fig. 5). The number of reproductions may depend not only on the implementation and number of cores but also on the algorithm itself. Therefore, it is a useful metric of speed when comparing the same model but with different implementations and numbers of cores. However, the number of reproductions relates to the number of fitness function evaluations, it is also a metric of efficiency between the alternate models.

Discussion. The results of the optimisation experiments are shown in Fig. 4. Both models in both implementations improve when cores are added. The sequential versions in both Erlang and Scala behaved just like the hybrid models with 1 core, minus some

small communication overhead, so these results are omitted further on.

The results for the Erlang and Scala implementations are similar in the early stages of experiments. In the later stages, the Erlang version becomes much slower than the Scala one. However, a close-up on the Erlang results reveals that its characteristics are also similar to the Scala results, but over a larger timespan.

The Scala results show that the hybrid version is initially faster. However, the concurrent model takes over at some point and becomes much more effective than the hybrid model in the later stages. In fact, in the case of 12 cores, 100% of the experiments with the concurrent model found the global optimum by the 12th minute of the experiments.

Another difference between the models is the number of reproduction happening every second (Fig. 5). In the case of the Erlang implementation, these numbers increase nearly linearly with the increase of nodes.

In the Scala implementation, the concurrent version also scales linearly, but the hybrid version drastically improves when the number of cores equals the number of islands. Looking at it another way, the performance of the Scala hybrid model drastically declines when there is less cores than islands, which is not the case of the

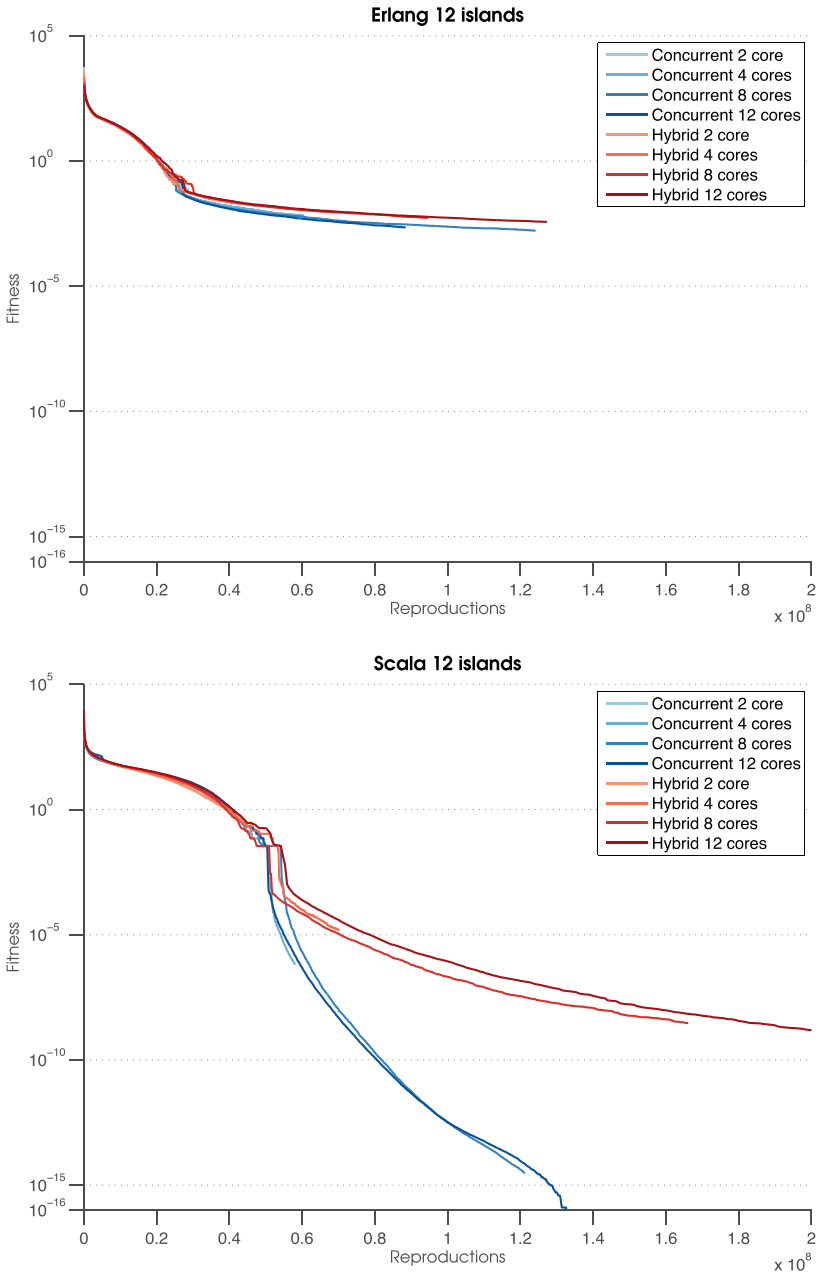


Fig. 6. Best fitness ever over *the total number of reproductions*. Confidence intervals are the same as in Fig. 4 and have been omitted for clarity.

Erlang implementation. In that regard, Erlang real-time scheduling proves to be more efficient than naive JVM threading (thread per island).

In both implementations the number of reproduction per second is significantly smaller in the case of the concurrent version. The interpretation can be twofold: on one hand, they may indicate that the concurrent implementation is slow at processing agent meetings. On the other hand, the number of reproductions reflect the number of fitness evaluations. As the concurrent version still achieves better results, but with fewer fitness evaluations, it can be considered more efficient.

This observation becomes all the more evident when plotting the best fitness over the number of total reproductions instead of over time. Fig. 6 confirms that the concurrent and hybrid models are in fact two different algorithms. Increasing the number of cores simply increases the number of reproductions per second and therefore reduces the time to reach a given value. However, the concurrent version needs a much lower number of reproductions, and therefore less function evaluations, to reach a given value.

This difference in dynamics could be explained in the following way: in the hybrid version, agents in the population are effectively synchronized, in the sense that all fights and all reproductions in a step need to end for any agent to move on. In contrast, in the concurrent version fights happen independently of reproduction and the population evolves in a much more continuous way. Information spreads faster in the population and the solution can be found with fewer generations.

Therefore, as the concurrent version needs less function evaluation, we conjecture that it should perform even better compared to the hybrid one when faced with real-life problems, where the computation of the fitness function itself can take much time.

The difference in performance between the Erlang and Scala versions can have several causes. First, the Erlang VM is usually less efficient than the Java VM when it comes to raw arithmetic. Both languages use 64 bit floating point numbers, though, so problems with numerical precision can be ruled out. Second, both languages strongly differ in memory management. In Erlang, every process owns a separate stack. With a few exceptions, all messages need to be copied from the memory of one process to another, even if the data is immutable and could be sent by reference. In contrast, Scala and Akka are based on the shared Java Memory Model [23], but offer different concurrency primitives in order to use message passing. Therefore, immutable data can be transferred within a VM as fast as in Java and as safely as in Erlang. All in all, the Scala implementation incurs less overhead and the differences in characteristics between the algorithms are amplified. However, it is the insight from designing the algorithm for Erlang first that led us to that efficient implementation in Scala.

6. Conclusions

The massively concurrent implementation of a evolutionary multi-agent system presented in this paper gives very promising results in terms of scalability and efficiency. Its asynchronous nature allows to better imitate the mechanisms observed in biological evolution, going beyond the classical approach of discrete generations and synchronous population changes.

We applied this algorithm to a popular optimization benchmark. The results of our experiments indicate that when many agents are involved, the concurrent model is significantly more efficient in terms of approaching the optimum versus number of fitness function evaluations. This result shows that this technique is very promising when the complexity of fitness function is high (e.g. in the case of solving inverse problems).

The key to achieve an efficient implementation was using Erlang and Scala technologies, in particular their features like lightweight processes and fast message passing concurrency. The Scala version appears to be more efficient, mainly because of a better memory management of the underlying library.

A further development of this method on modern multicore supercomputers [16] seems a promising direction of research. Broader tests will also be performed in multicore systems consisting of a higher number of cores than examined here.

Acknowledgements

The research presented in the paper was partially supported by the European Commission FP7 through the project ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems, under contract no.: 288570 (http://paraphrase-ict.eu). The research presented in this paper received partial financial support from AGH University of Science and Technology statutory project. The research presented in the paper was conducted using PL-Grid Infrastructure (http://www.plgrid.pl/en).

References

[1] F. Bellifemine, A. Poggi, G. Rimassa, JADE: a FIPA2000 compliant agent development environment, in: Proceedings of the Fifth International Conference on Autonomous Agents, AGENTS'01, ACM, New York, NY, USA, 2001, pp. 216–217.
[2] A. Byrski, R. Dreżewski, L. Siwik, M. Kisiel-Dorohinicki, Evolutionary multi-agent systems, Knowl. Eng. Rev. 30 (2015) 171–186.
[3] A. Byrski, W. Korczyński, M. Kisiel-Dorohinicki, Memetic multi-agent computing in difficult continuous optimisation, in: Advanced Methods and Technologies for Agent and Multi-Agent Systems, IOS Press, 2013, pp. 181–190.
[4] A. Byrski, R. Schaefer, M. Smołka, Asymptotic guarantee of success for multi-agent memetic systems, Bull. Pol. Acad. Sci. – Tech. Sci. 61 (1) (2013).
[5] A. Byrski, Tuning of agent-based computing, Comput. Sci. (AGH) 14 (3) (2013) 491.
[6] A. Byrski, M. Kisiel-Dorohinicki, Agent-based model and computing environment facilitating the development of distributed computational intelligence systems, in: G. Allen, J. Nabrzyski, E. Seidel, G. van Albada, J. Dongarra, P.M.A. Sloot (Eds.), Computational Science – ICCS 2009, Lecture Notes in Computer Science, vol. 5545, Springer, Berlin/Heidelberg, 2009, pp. 865–874.
[7] A. Byrski, R. Schaefer, Formal model for agent-based asynchronous evolutionary computation, in: Proceedings of the IEEE Congress on Evolutionary Computation, CEC, Trondheim, Norway, 18–21 May, IEEE, 2009, pp. 78–85.
[8] E. Cantú-Paz, A survey of parallel genetic algorithms, Calc. Paralleles Reseaux Syst. Repartis 10 (2) (1998) 141–171.
[9] K. Cetnarowicz, M. Kisiel-Dorohinicki, E. Nawarecki, The application of evolution process in multi-agent world (MAW) to the prediction system, in: M. Tokoro (Ed.), Proc. of the 2nd Int. Conf. on Multi-Agent Systems (ICMAS'96). AAAI Press, 1996.
[10] Á.F. Díaz, C.B. Earle, L.-Â. Fredlund, eJason: an implementation of Jason in Erlang, in: M. Dastani, J.F. Hübner, B. Logan (Eds.), Programming Multi-Agent Systems, Lecture Notes in Computer Science, vol. 7837, Springer, Berlin/Heidelberg, 2013, pp. 1–16.
[11] A. Di Stefano, C. Santoro, eXAT: an experimental tool for programming multi-agent systems in Erlang, in: WOA, 2003, pp. 1–127.
[12] A.U. Frank, S. Bittner, M. Raubal, Spatial and cognitive simulation with multi-agent systems, in: D.R. Montello (Ed.), Spatial Information Theory, Lecture Notes in Computer Science, vol., Springer, Berlin/Heidelberg, 2001, pp. 124–139.
[13] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989.
[14] C. Grigore, R. Collier, Supporting agent systems in the programming language, in: 2011 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT), Aug, vol. 3, 2011, pp. 9–12.
[15] O. Gutknecht, J. Ferber, The MadKit agent platform architecture, in: Infrastructure for Agents, Multi-Agent Systems and Scalable Multi-Agent Systems, 2001.
[16] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. Gonzalez-Velez, P. Kilpatrick, R. Keller, M. Rossbory, G. Shainer, The paraphrase project: parallel patterns for adaptive heterogeneous multicore systems, in: FMCO: 10th International Symposium on Formal Methods for Components and Objects-Revised Selected Papers, vol. 7542, Springer LNCS, 2013, pp. 218–236.
[17] N.R. Jennings, K. Sycara, M. Wooldridge, A roadmap of agent research and development, J. Auton. Agents Multi-Agent Syst. 1 (1) (1998) 7–38.

[18] M. Kolybacz, M. Kowol, L. Lesniak, A. Byrski, M. Kisiel-Dorohinicki, Efficiency of memetic and evolutionary computing in combinatorial optimisation, in: W. Rekdalsbakken, R.T. Bye, H. Zhang (Eds.), Proceedings of the 27th European Conference on Modelling and Simulation, ECMS, Ålesund, Norway, May 27–30, European Council for Modeling and Simulation, 2013, pp. 525–531.
[19] M. Kowol, A. Byrski, M. Kisiel-Dorohinicki, Agent-based evolutionary computing for difficult discrete problems, in: D. Abramson, M. Lees, V.V. Krzhizhanovskaya, J. Dongarra, P.M.A. Sloot (Eds.), Proceedings of the International Conference on Computational Science, ICCS, Cairns, Queensland, Australia, 10–12 June, vol. 29 of Procedia Computer Science, Elsevier, 2014, pp. 1039–1047.
[20] D. Krzywicki, Ł. Faber, A. Byrski, M. Kisiel-Dorohinicki, Computing agents for decision support systems, Future Gener. Comput. Syst. 37 (2014) 390–400.
[21] D. Krzywicki, J. Stypka, P. Anielski, Ł. Faber, W. Turek, A. Byrski, M. Kisiel-Dorohinicki, Generation-free agent-based evolutionary computing, Procedia Comput. Sci. 29 (2014) 1068–1077, 2014 International Conference on Computational Science.
[22] B. Manate, V.I. Munteanu, T.-F. Fortis, Towards a scalable multi-agent architecture for managing IOT data, in: 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), October, 2013, pp. 270–275.
[23] J. Manson, W. Pugh, S.V. Adve, The java memory model, SIGPLAN Not. 40 (1) (2005) 378–391.
[24] Z. Michalewicz, Genetic Algorithms Plus Data Structures Equals Evolution Programs, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1994.
[25] M. Piotrowski, W. Turek, Software agents mobility using process migration mechanism in distributed Erlang, in: Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang, Erlang'13, ACM, New York, NY, USA, 2013, pp. 43–50.
[26] S. Pisarski, A. Rugała, A. Byrski, M. Kisiel-Dorohinicki, Evolutionary multi-agent system in hard benchmark continuous optimisation, in: A.I. Esparcia-Alcázar (Ed.), Applications of Evolutionary Computation, Lecture Notes in Computer Science, vol. 7835, Springer, Berlin/Heidelberg, 2013, pp. 132–141.
[27] A. Pokahr, L. Braubach, K. Jander, The Jadex project: programming model, in: M. Ganzha, L.C. Jain (Eds.), Multiagent Systems and Applications, Intelligent Systems Reference Library, vol. 45, Springer, Berlin/Heidelberg, 2013, pp. 21–53.
[28] M. Polnik, M. Kumiega, A. Byrski, Agent-based optimization of advisory strategy parameters, J. Telecommun. Inf. Technol. 2 (2013) 54–55.
[29] R. Schaefer, A. Byrski, J. Kolodziej, M. Smolka, An agent-based model of hierarchic genetic search, Comput. Math. Appl. 64 (12) (2012) 3763–3776.
[30] H.-P. Schwefel, G. Rudolph, Contemporary evolution strategies, in: European Conference on Artificial Life, 1995, pp. 893–907.
[31] W. Turek, Erlang as a high performance software agent platform, Adv. Methods Technol. Agent Multi-Agent Syst. 252 (2013) 21.
[32] M. Vose, The Simple Genetic Algorithm: Foundations and Theory, MIT Press, Cambridge, MA, USA, 1998.
[33] D.H. Wolpert, W.G. Macready, No free lunch theorems for optimization, IEEE Trans. Evol. Comput. 67 (1) (1997).
[34] K. Wrobel, P. Torba, M. Paszynski, A. Byrski, Evolutionary multi-agent computing in inverse problems, Comput. Sci. (AGH) 14 (3) (2013) 367–384.

D. Krzywicki obtained his M.Sc. in 2012 at AGH University of Science and Technology in Cracow and is currently a Ph.D. student at the Department of Computer Science of AGH-UST. His research interests include agent-based computations, functional programming and distributed systems.

W. Turek obtained his Ph.D. in 2010 at AGH University of Science and Technology in Cracow. He works as an Assistant Professor at the Department of Computer Science of AGH-UST. His research focuses on agent-based systems, multi-robot systems and functional programming.

A. Byrski obtained his Ph.D. in 2007 and D.Sc. (habilitation) in 2013 at AGH University of Science and Technology in Cracow. He works as an Assistant Professor at the Department of Computer Science of AGH-UST. His research focuses on multi-agent systems, biologically inspired computing and other soft computing methods.

M. Kisiel-Dorohinicki obtained his Ph.D. in 2001 and D.Sc. (habilitation) in 2013 at AGH University of Science and Technology in Cracow. He works as an Assistant Professor at the Department of Computer Science of AGH-UST. His research focuses on intelligent software systems, particularly using agent technology and evolutionary algorithms, but also other soft computing techniques.

Journal of Computational Science 17 (2016) 234–248

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Highly scalable Erlang framework for agent-based metaheuristic computing

Wojciech Turek, Jan Stypka, Daniel Krzywicki, Piotr Anielski, Kamil Pietak, Aleksander Byrski*, Marek Kisiel-Dorohinicki

AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Al. Mickiewicza 30, 30-059 Krakow, Poland

ARTICLE INFO

Article history:
Received 30 June 2015
Received in revised form 1 March 2016
Accepted 2 March 2016
Available online 8 March 2016

Keywords:
Metaheuristic computing
Concurrent programming
Scalability
Erlang

ABSTRACT

Difficult search and optimization problems, usually solved by metaheuristics, are very often implemented in concurrent and parallel environment, as many metaheuristics (e.g. population- or agent-based) are inherently easy to parallelize. Therefore search for easy-to-use, robust and efficient frameworks dedicated for such computing methods, especially in the era of ubiquitous many and multi-core systems, is very desirable. Indeed, the development of multi-core architectures is incredibly fast and multicore CPUs can be found nowadays not only in supercomputers, but also in ordinary laptops or even phones. Efficient use of multicore architectures requires applying suitable languages and technologies, like Erlang. Its concurrency model, based on lightweight processes and asynchronous message-passing, seems very well suited for running massively concurrent code on many cores. Most of existing Erlang industrial applications are deployed on computers with up to 24 CPU cores, and there are hardly any reports on using Erlang on architectures exceeding 32 physical cores. In this paper we present our experiences with developing a concurrent Erlang-based computing platform, scaling computationally-intensive and memory-intensive applications up to 64 cores, using as examples global optimization and urban traffic planning problems.

1. Introduction

Tackling difficult search problems calls for applying unconventional methods. This necessity is imposed by having little or no knowledge of the intrinsic features of the problem, topology of search space, etc. In such cases, approximate techniques, like metaheuristics become the methods of last resort. Having a plethora of metaheuristics to choose from, those population-based (as opposed to single solution oriented) seem to be the best choice, both at algorithmic and implementation level, and as they process more than one solution at a time, they can evade local extrema easier than single-solution approaches. Moreover, it is easy to implement them efficiently using ubiquitous parallel systems, such as multi-core processors, graphical processing units, clusters and grids.

Evolutionary processes are by nature decentralized and therefore evolutionary processes in a multi-agent system at a population level may be easily introduced. It means that agents are able to *reproduce* (generate new agents), which is a kind of cooperative interaction, and may *die* (be eliminated from the system), which is the result of competition (selection). This idea came into fruition by Cetnarowicz in 1996 [1] as Evolutionary Multi-Agent System, and since that time implemented (cf., e.g., [2–4]) a number of times, analysed [5] and extended [6,7]. It is to note that EMAS turned out to be an efficient paradigm for solving different optimization problems [8,9].

The development of multi-core architectures over the last ten years is amazingly fast. Mainstream computer components manufacturers compete with smaller companies and startups in designing more and more sophisticated architectures, which include tens (soon hundreds) of cores in a single processor. Architectures like Intel Xeon Phi [10] or Adapteva Epiphany [11] are typically available as dedicated accelerators, however, general-purpose many-core CPUs, like the 100-core EZchip TILE-MX [12] will soon become standard equipment of computing stations.

This rapid development poses significant challenges for the software industry which seems unprepared for using this

* Corresponding author.
E-mail addresses: wojciech.turek@agh.edu.pl (W. Turek), janstypka@gmail.com (J. Stypka), krzywic@agh.edu.pl (D. Krzywicki), pr.anielski@gmail.com (P. Anielski), kpietak@agh.edu.pl (K. Pietak), olekb@agh.edu.pl (A. Byrski), doroh@agh.edu.pl (M. Kisiel-Dorohinicki).

http://dx.doi.org/10.1016/j.jocs.2016.03.003
1877-7503/

computational power. The growing number of independent cores available for a single process makes the most popular software development technologies inefficient. Being based on imperative programming paradigm and shared memory concurrency model, the technologies are unsuitable for handling hundreds of simultaneously running tasks.

In this paper, we present and compare several approaches to the problem of building a framework for metaheuristic computing in Erlang. Erlang [13] is a functional programming language, which have been continuously developed by Ericsson since the eighties of the last century. It was originally designed for programming telecommunication devices, where features like massive concurrency and durability are far more important than performance. Erlang programs are executed in the Erlang Virtual Machine, which implements its own lightweight processes, context switching with preemptive scheduler and message passing concurrency model. Erlang processes do not share state and communicate using asynchronous messages, which removes the possibility of creating synchronization bottlenecks. These features turn out to be very desirable in the era of multi-core processors, therefore Erlang is rapidly gaining popularity in the IT world as well. It has been successfully used for implementing various messaging platforms (RabbitMQ,[1] MangooseIM[2]) and no-SQL database servers (Riak,[3] CouchDB[4]). Its unique features are also an excellent foundation for building agent systems [14].

Computations performed within the Erlang virtual machine are often less efficient than similar algorithms written in different technologies, because compute-intensive applications have never been the major target of Erlang creators. However unique features of the Erlang technology allowed us to provide linear scalability of agent-based metaheuristic computing framework on a many-core architecture. This future can very important in the upcoming era of many-core hardware.

The created framework has been tested on two different types of computations. Firstly, we used a typical benchmark function with many local minima, which required time-consuming evaluation without complex operation on memory. In order to further evaluate features of the developed solution, a real life optimization problem has been considered. The problem of micro-scale urban traffic planning was solved using a novel, multi-variant planning approach, which continuously prepares various solutions to the most probable situations on the managed crossroad. This problem requires performing costly computations together with memory-intensive operations. We show that the scalability of certain internal mechanisms of the Erlang VM can be limited and we present several methods of diagnosing the reasons of particular problems and solutions to improve the scalability of Erlang.

After the introduction, we present our parallel computing model (an evolutionary multi-agent system, eMAS), as a universal optimization algorithm, along with the description of its parallel implementation using the functional approach. Then, we present the architecture of our Erlang computing framework and the details of four different implementations of eMAS. Next, we show how to scale the Erlang VM up to 64 cores, by reporting the problems we encountered and the solutions we found, along with experimental results illustrating consecutive steps. Finally, we describe a real-life problem of traffic management planning, presenting its implementation, our results and conclusions.

[1] RabbitMQ, an open-source messaging broker, http://www.rabbitmq.com/.
[2] MangooseIM, massively-scalable XMPP server, http://www.erlang-solutions.com/products/mongooseim.html.
[3] Riak, no-SQL key-value data store, http://basho.com/products/%23riak.
[4] CouchDB, a document-oriented database, http://couchdb.apache.org/.

2. Agent-oriented frameworks and computing

Agent-based software environments use agents as basic units of software abstraction. They focus on inter-agent relations and intra-agent intelligence, provide facilities for the discovery of agents, communication, life-cycle management, etc. The FIPA [5] standard allows to create such open, interoperable multi-agent systems, where fully-fledged autonomous software agents can express their needs or perceive the environment (and other agents) using specific languages, ontologies, etc. The most established solutions of this kind include JADE [15] and JADEX [16]. However, the approach of open systems with heavy agents is not applicable for computational systems – granularity of agents is similar to services in SOA (service-oriented architecture).

In case of simulations or computations, where the introduction of agents facilitates the modelling of complex phenomena, such as natural or social ones, agents constitute building blocks of the model. This approach is utilized in frameworks such as Mason, RePast or MadKit.

The first one, **MASON** [17], is a single-process discrete-event simulation core and visualization library written in Java developed at George Mason University. It is supposed to support efficiently up to a million agents without visualization facilities. The multi-layer architecture brings complete independence of the simulation logic from visualization tools. There are none ready-to-use distributed computing facilities, however it can be integrated into larger existing libraries. The programming model of MASON follows the basic principles of object-oriented design. Agents are lightweight entities represented as Java objects with step method. They are added to a scheduler and their step method is called during the simulation. Agents API is flexible and allows to model various computation models, but there is no built-in metaheuristics.

RePast [18] is a widely used agent-based modelling and simulation tool. It has multiple implementations in several languages (e.g RePast Simphony in Java and RePast HPC written in C++) and built-in adaptive features such as genetic algorithms and regression. The framework uses fully concurrent discrete event scheduling. Dynamic access to the models in the runtime (introspection) is possible using a graphical user interface. The RePast distribution has a large footprint: included in the package are neural networks, genetic algorithms, social network modelling, dynamic systems modeling, logging, GIS, and graphs and charts [17].

MadKit is a modular and scalable multi-agent platform written in Java, aimed at modelling different agent organizations, groups and roles in artificial societies. It is built based on a so-called Agent/Group/Role organizational model, using a plugin-based architecture. The architecture of MadKit is based on micro-kernels which provide only the basic facilities: local messaging, management of groups and roles, launching and killing of agents. Other features (remote messages, visualization, monitoring and control of agents) are performed by agents. Both thread based and scheduled agents may be developed.

All of the above platforms provide wide range of facilities supporting agent-based computations and simulations. They all are built using imperative languages such as Java or C++. Agents are represented as simple objects sequentially executed by a scheduler (so-called steppable agents) or as heavy agents usually executed as separated threads.

This approach makes the implementation of agent-based systems a very natural, as both concept and implementation paradigm are agent-oriented. Therefore it is very easy to map the agent concept onto the framework. However, we have been working

[5] Foundation for Intelligent Physical Agents http://www.fipa.org/.

for over 15 years on developing agent-based computing algorithms and implementing them using dedicated frameworks, not necessarily agent-oriented. This was caused by the observation, that computing-agents can be implemented as "lightweight agents", not necessarily requiring complex methods such as agent-communication languages, code-migration and ontologies for modelling of the data processed and the whole environment. Instead, we focused on implementing the framework in such way, that other important factors of such complex systems might be emphasized, such as flexibility, extensibility, scalability. It lead us to utilizing such technologies as Java, Python, Scala and Erlang (cf., e.g., [19–21]).

There above-cited different agent-based frameworks, such as RePast [18], MASON [17], MadKit [22], JADE [15] are seldom used for computing purposes. Instead more focused frameworks are constructed, as e.g. ParadisEO [23] or AgE [19]. These are often advocated as open-source and freely available, thanks to their creator's generosity. Some of these platforms leverage existing multi-core architectures (e.g. Paradiseo), however there is not too many reports on their scalability, although having such hardware easily available nowadays, as 64 core processors, encourages for doing such research. Nevertheless, let us refer to some of such research here, intentionally focusing on parallel frameworks (leaving the distributed ones for further research, when we have an appropriately-prepared framework available.

In [24], the authors present an implementation of a Java-based agent-oriented system, achieving linear scalability up to 8 cores. Another framework, also implemented in Java, this time devoted for simulation, appears to be able to scale up to 8 cores [25]. In [26] the authors show an implementation of a C++/MPI scalable simulation platforms for spatial simulations of particles movement, achieving linear scalability up to 64 cores. In [27] ParadisEO scalability is tested, reaching linear speedup up to 10 cores (the implementation is realized using C++).

However, to fully leverage the existence of multi- and many-core computers, the community of language developers seem to agree that a paradigm shift is needed [28]. Efficient development of software for many-core architectures will require high level functional languages with immutable variables, no shared state and message-passing concurrency. Some of these relatively old concepts are being adopted in new and existing languages, like C++, Java or Scala. These are also the most basic assumptions of Erlang programming language.

Erlang [29] is a high level functional language which has proven to be an efficient tool for building large-scale systems for multi-core processors. The concurrency model based on lightweight processes seems very well suited for running massively concurrent code on many cores or processors. Therefore, it might seem that Erlang is a good programming choice for parallel computing on many-core architectures. Finding out that it is not that simple cost us a lot of research effort, which we summarize in this paper.

Most of Erlang industrial applications are deployed on computers with up to 24 CPU cores. Scalability of solutions is provided by using clusters of computers – running Erlang in distributed configuration. Actually, the Erlang/OTP team developing the virtual machine does not test the implementation on bigger architectures.[6]

There are few reports on using a single Erlang virtual machine on architectures exceeding 32 physical cores. In [30] the authors present a test suite for measuring different aspects of Erlang applications performance. The exemplary test running on a 64-core machine shows that in most cases the speedup is non-linear and it degrades for high number of cores and schedulers. The problem of Erlang term storage scalability on a computer with 32 physical cores have been considered in [31]. Promising results of using Erlang on a Intel Xeon Phi coprocessor have been shown in [32]. Basic benchmarks show good scalability up to 60 cores, which is the number of physical cores of the coprocessor. However, there are hardly any reports on scaling complex, computationally intensive Erlang applications on many-core architectures.

The computing framework presented in this paper has a counterpart implemented in Scala[7] [33], being a subject of research on the same concurrent implementation style yet using other technology. The comparison of these platform became the main subject of other publications of our research team.

3. Computing models for agent-based metaheuristics

Various models of parallel implementations of evolutionary algorithms have already been proposed [34]. The standard approach (sometimes called a *global parallelization*) consists in distributing selected steps of the sequential algorithm among several processing units. *Decomposition* approaches are based on defining different complex models such as *coarse-grained* and *fine-grained* parallel evolutionary algorithms. There are also methods which use some combination of the models described above (*hybrid* parallel evolutionary algorithms).

Agents play an important role in the integration of artificial intelligence subdisciplines, which is often related to a hybrid design of modern intelligent systems [35]. In some similar applications reported in the literature (see, e.g. [36,37] for a review), an evolutionary algorithm is used by an agent to facilitate the execution of some of its tasks, often connected with learning or reasoning, or to support coordination of some group (team) activity. In other approaches, agents constitute a management infrastructure for a distributed realization of an evolutionary algorithm [38]. A quite similar approach is proposed by Liu et al. [39,40] where agents are situated on the lattice, derivatives of their fitness are encoded in a form of energy function and dedicated operators for reproduction and removal of the individuals are introduced (mimicking cellular evolutionary algorithms [41]). A very interesting example of parallel metaheuristics platform is EvoSpace capable of running as a service on cloud [42]. Metaheuristics are also implemented in P2P networks [43].

In this section, starting from an EMAS [1] being an example of general-purpose agent-based metaheuristic, concurrent, scalable way of implementation and relevant Erlang-based software platform are presented.

3.1. Evolutionary multi-agent system

Evolutionary multi-agent systems are a hybrid meta-heuristic which combines multiagent systems with evolutionary algorithms. The idea consists in evolving a population of agents to improve their ability to solve a particular optimization problem [44,5].

In a multi-agent system no global knowledge is available to individual agents. Agents should remain autonomous and no central authority should be needed. Therefore, in an evolutionary computing system, selective pressure needs to be decentralized, in contrast with traditional evolutionary algorithms. Using agent terminology, we can say that selective pressure is required to emerge from peer to peer interactions between agents instead of being globally-driven.

In a basic algorithm, every agent is assigned with a real-valued vector representing a potential solution to the optimization

[6] Based on a discussion with an OTP member at the Erlang User Conference in Stockholm, 2014.

[7] http://github.com/ParaPhraseAGH/scala-mas.

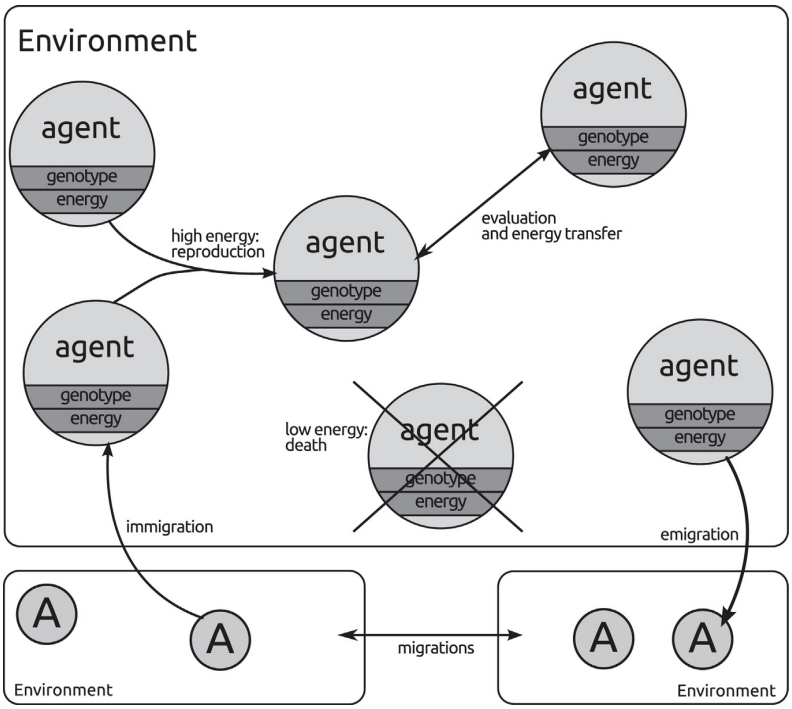


Fig. 1. EMAS structure and principle of work.

problem, along with the corresponding fitness. Emergent selective pressure is achieved by giving agents a single non-renewable resource called energy. Agents start with an initial amount of energy and meet randomly. If their energy is below a threshold, they fight by comparing their fitness – better agents take energy from worse ones. Otherwise, the agents reproduce and yield a new one – the genotype of the child is derived from its parents using variation operators and it also receives some energy from its parents. The system is stable as the total energy remains constant, but the number of agents may vary and adapt to the difficulty of the problem (see Fig. 1).

As in other evolutionary algorithms, agents can be split into separate populations. Such sub-populations, called islands, help preserve diversity by introducing allopatric speciation and can also execute in parallel. Information is exchanged between islands through agent migrations.

It should be noted, that the EMAS computing abilities were formally proven by constructing a detailed Markov-chain based model and proving its ergodicity [45,5], showing EMAS as a general optimization tool.

3.2. Interaction and execution models for agents

In agent-oriented computing systems, agent interactions are one of the crucial aspects of their work. It is easy to predict that parallelizing them can significantly increase the throughput of the system. However, this comes at the cost of increased communication and synchronization. Therefore, an important issue is to choose the appropriate granularity of the entities in the computation.

As agents are defined as autonomous and independent beings, it seems natural to look for further concurrency within a single environment. The question is where to put the boundaries of concurrent execution, as it has consequences on both performance and ease of programming. This section discusses the most common models of execution and interaction in existing agent software [33].

3.2.1. Heavyweight agents

In this model every agent is associated with a thread and communicates through message passing. Some agents may passively wait for incoming messages and react to them. Other agents may actively initiate interactions with other agents. It is difficult to achieve a coordinated life cycle among such agents, since the corresponding threads may be arbitrary interleaved. Therefore, some kind of synchronization between agents still needs to be introduced, usually in terms of a specific communication protocol.

In order to interact with each other, agents need to locate other agents willing to perform the same actions. For example, in an evolutionary multi-agent system, an agent with enough resources to reproduce needs to find another one which also has enough resources. In order to do that, it could ask all other agents in the population. However, such a solution is obviously inefficient, because of the intensity and redundancy of the required communication.

A better approach, introduced in [33], is to use a mediating entity, which we call a *meeting arena*. Every time an agent wants to perform an action, it chooses an appropriate arena based on its energy level in order to meet with other similar agents (see Listing 1). The arena is then able to partition its members in groups of some given arity and mediate the meeting itself (see Fig. 2). The arena code responsible for administering and processing meetings is presented in the Listing 2. It shows the logic executed for each different meeting that is matched in the function clause.

The usage of meeting arenas should bring many benefits, not only in terms of efficiency, as the algorithm itself can be structured more clearly. Agents only need to be given a set of rules, in order to choose an arena on the basis of their state. The actual protocol of agents interactions can then be defined at the level of the appropriate arena.

```
1 behaviour(Agent) when Agent#agent.energy == 0 -> death
2 behaviour(Agent) when Agent#agent.energy > 10 -> reproduction
3 behaviour(Agent) -> fight
```

Listing 1. In every step, agents choose an arena based on their current state.

```
1 meeting({death, Agents}) -> []
2 meeting({reproduction, Agents}) ->
3   lists:flatmap(
4     fun doReproduce/1,
5     inGroupsOf(2, Agents));
6 meeting({fight, Agents}) ->
7   lists:flatmap(
8     fun doFight/1,
9     inGroupsOf(2, Agents));
```

Listing 2. Arenas process partitions of the population and trigger agent meetings.

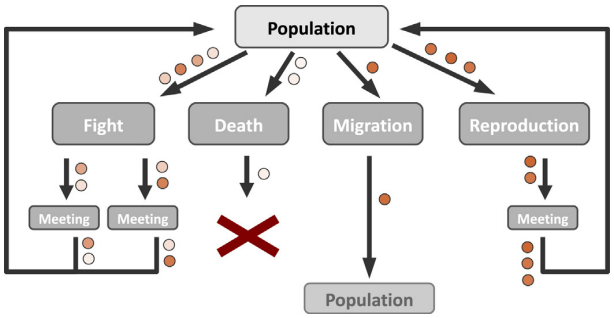


Fig. 2. Meeting arenas allow to group similar agents and coordinate meetings between them.

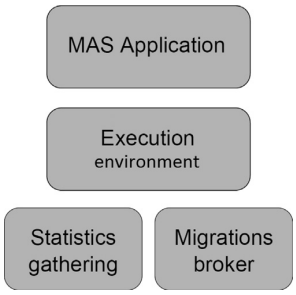


Fig. 3. The architecture of the platform.

Assigning a thread to each agent may feel very natural. In practice, however, the number of agents is often much higher than the number of cores. Performance may then be seriously hindered by frequent context switches, although this overhead may be reduced by sharing a pool of threads among agents. However, this model still involves intensive communication and costly processor cache synchronization. In consequence, the trade-off for such concurrency may not be worthwhile.

3.2.2. Lightweight agents

An opposite approach is to consider agents as parts of the model, but not parts of the implementation. As such, they are simply represented as data structures and processed like in a discrete event simulation.

The execution of an individual agent has to be divided into smaller parts which can be interleaved. These parts, which we will call *actions*, could for example consist of moving to different location or meeting a neighbour. Given its current state, every agent decides which action to perform next (function *behaviour/1* in Listing 3). Then the agents are grouped by the selected action (function *group_by_behaviour/1*) and the actions are performed on pairs or individual agents (function *meeting/1*). Finally the new population is shuffled to change the order of agents before next iteration of the algorithm. All of these four consequential steps are presented in the Listing 3.

The performance of such a model will usually be higher than in the previous one, as more consistent memory access patterns result in more efficient processor usage. Even though the explicit parallelism is reduced, throughput can be improved, because frequent agent interactions no longer need to be synchronized between threads.

Moreover, independent actions can still be executed in parallel by the executor service. This is consistent with the meeting arena concept described above, as actions on common subsets of agents may be grouped together and considered as a single meeting.

4. Framework architecture and EMAS implementation

The platform is thus divided into two layers: the multi-agent application and the execution environment (Fig. 3). The application layer abstracts from the execution and focus on the interactions. It defines

- the types of agents
- their possible states and actions
- the behaviour function mapping states to actions
- the meetings function updating agents subpopulation for a given action.

The execution environment implements the logic of computing of these functions (realizing them in the form of one of the described below execution models) and tying their results together to run the simulations. Several versions of execution models are described below, but they all expose the same API, accepting the types and functions defined at the application layer.

This decoupling allows to design the multi-agent algorithm separately and later choose and tune the execution model to best fit a given problem or hardware configuration.

A broker module is responsible for handling agent migrations between environments, independently of their underlying execution environment. Although this is outside the scope of this paper, the same mechanism also allows the application to handling cross-node migrations in a distributed environment.

```
1  update(Agents) ->
2    Tagged = [{behaviour(A), A} || A <- Agents],
3    Grouped = group_by_behaviour(Tagged),
4    NewAgents = lists:flatmap(fun meeting/1, Grouped)
5    shuffle(NewAgents)
```

Listing 3. The behaviour and meeting functions are applied to derive the next agent population.

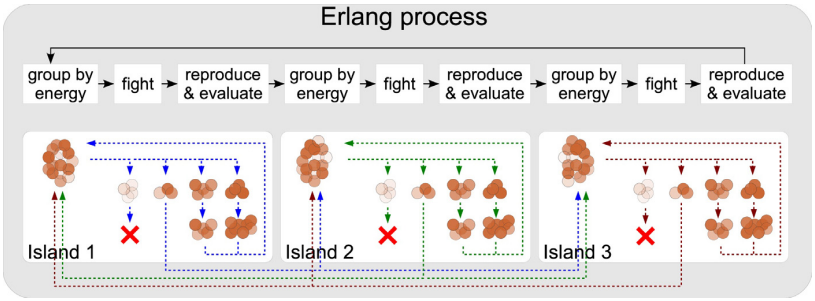


Fig. 4. Outline of the sequential version of EMAS.

The last module is responsible for gathering statistics and metrics from the running application. Several mechanisms are available for different levels of concurrency. In particular, a special support is provided for statistics that are monotonic, such as the number of agents meetings or the best fitness found so far. Such statistics can be computed at any time and recorded asynchronously. They can be periodically *flattened* to yield the value of the statistic at a given time, with minimal synchronization between sources.

As described in Section 3.2, choosing an appropriate execution model for agent interactions is crucial to achieve efficient parallelism in multi-agent systems. Such execution models may differ in terms of granularity and other characteristics, in particular their efficiency for a given application. Decoupling agent behaviour and meetings from the interaction mechanism [33] allows to swap different execution models and compare their performance.

We have designed three different implementations of agent interactions based on our previous experiences gathered during implementation of different types of agent-based computing frameworks:

- hybrid (coarse-grained), following our approach to implement the system as a number of processes working in parallel, communicating among themselves and running the real computing and agent-related tasks inside these processes, following so-called discrete event simulation [46] in order to simplify the interactions among the agents, improving the efficiency and scalability of the whole system,
- concurrent (fine-grained), prepared in order to test the well-known Erlang (and Scala – in another paper [20]) capability to run and communicate immense numbers of lightweight processes,
- SKEL-based (also coarse-grained, based on the SKEL [47] library[8]), in order to test the applicability of the parallel-patterns included there,
- sequential version, implemented for comparison purposes.

[8] SKEL is one of the products of EC FP7 ParaPhrase Project, contract no. 288570, http://paraphrase-ict.eu/.

4.1. Sequential

The sequential version is very simple: it repeatedly applies the behaviour and meeting functions to update the population of agents (Listing 3). Additionally, several such agent populations are kept and in every step agents may migrate between them with low probability. All the populations are then wrapped in a recursive loop within a single Erlang process until some stop criterion is met.

Fig. 4 presents the outline of the sequential implementation. Single Erlang process iterates over the collection of islands. Within each island, solutions are grouped by energy value into three groups: the agents to remove, fight and reproduce. Additionally, a group of agents for migration is selected with low probability from the last two groups. In following steps, the process performs migration (moves the solutions to another agents collection) and meetings between agents. Operation of fighting results in equinumerous set of agents, reproduction causes growth in the number of agents. New solutions are evaluated during this process, which is the most computationally-expensive operation of the whole process.

4.2. Hybrid

The hybrid version is the most straightforward way to parallelize the simplest sequential version. Every agent population is contained in a separate Erlang process and runs in an independent loop. The outline of this implementation is presented in Fig. 5.

Agent migration is realized through message passing. A single, separate process acts as a migration broker and is responsible for forwarding migrating agents according to some topology. In practice, the migration probability is very low (typically a few migrations every second), so this single process does not become a bottleneck but simplifies communication.

4.3. Skeletons

The SKEL-based implementation is the result of refactoring the sequential version by introducing skeletons from the SKEL library. These skeletons are used to emulate loops and parallel patterns in the code. The outline of this implementation is presented in Fig. 6.

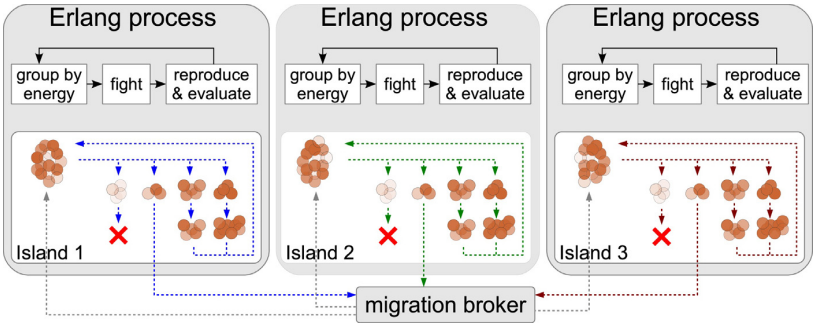


Fig. 5. Outline of the hybrid version of EMAS.

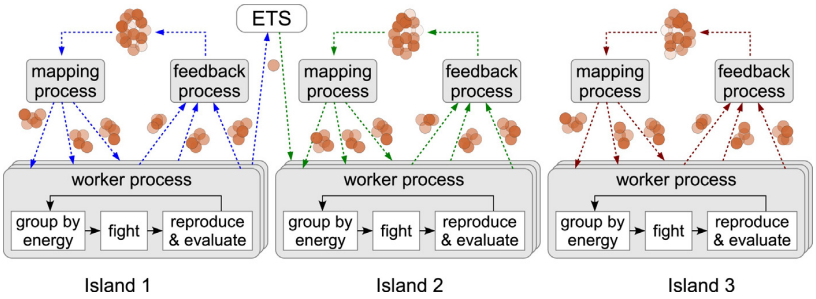


Fig. 6. Outline of the SKEL version of EMAS.

Agent populations are parallelized with the map skeleton and each island operates within a loop. This loop is implemented with a feedback pattern, which basically emulates the classic *while* instruction i.e. repeats its content whilst a certain condition holds true. While the rest of the logic is implemented with skeletons, asynchronous migration had to be written separately. Agents are exchanged between islands through an ETS – Erlang term storage – a shared memory space, which is fast and threadsafe.

The codebase of this implementation is smaller while the final performance is almost as good as in the hybrid model. Also, the structure of the parallel workflow can be easily changed. On the other hand, this structure highly influences the performance, so we had to experiment with several versions until good performance was achieved.

4.4. Concurrent

The concurrent version represents an approach where every agent runs in a separate autonomous Erlang process. As they lack global knowledge, agents only interact through the mediating meeting arenas. In these models the arenas become distinguished entities. They are implemented as Erlang processes (gen_servers) constant for each island. There is a separate arena responsible for each kind of interaction: fight, reproduction, death and migration. The outline of this implementation is presented in Fig. 7.

When agents decide to carry out a certain action e.g. reproduce, they send a message to an appropriate arena. The arena pairs such incoming agents, calculates the interaction and sends back its results. This approach allows to keep very lightweight agents and removes all synchronization barriers from the algorithm, at the cost of more expensive communication.

Every agent population has a separate set of arenas. Migrating an agent is as simple as changing the PIDs of the arenas it communicates with.

5. Scalability of the platform applied for benchmark continuous optimization

In this section, the scalability issues encountered during optimization of the framework applied for continuous optimization are discussed.

All the versions of the algorithm described in the previous section have been carefully tested to evaluate their performance, scalability and compare with each other. To our surprise, initial implementations off all of them did not scale well for large numbers of CPU cores.

The performance metric recorded in our experiments is the number of agent reproductions happening every second. Our previous research [48] indicated that this value is proportional to the amount of all other events in the system and thus is a good indication of the system's overall throughput. We do not present results of fitness value along time, as these are dependent mostly on the problem definition and genetic operators implementation and therefore do not represent the scalability of the platform.

We ran our experiments on the ZEUS supercomputer provided by the PI-Grid[9] infrastructure at the ACC Cyfronet AGH.[10] We used nodes with 4 AMD Opteron 6276 processors each (64 cores per node) and a total of 256 GB of memory per node.

For every experiment, we started the computations and let the system reach a steady state. Then, we recorded throughput during several minutes. This procedure was repeated 30 times to account for operating system variability or dependence on random number generator seed. Finally, these throughput were averaged over time and experiments for a given configuration. We do not provide complete statistical measures because the results were highly repeatable.

[9] http://www.plgrid.pl/en.
[10] http://www.cyfronet.krakow.pl/en/.

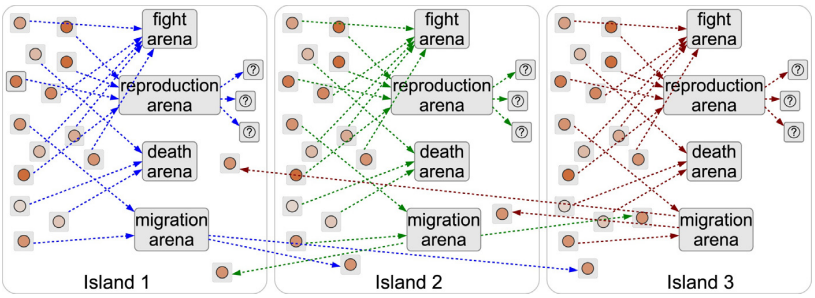


Fig. 7. Outline of the concurrent version of EMAS.

The particular problem considered in the experiments was finding the 100 dimensional Rastrigin function minimum. Configuration of the genetic algorithm in all experiments included 64 evolutionary islands with initial number of agents on each island equal to 100. Initial energy of a new agent (initial or offspring) was 10, while the threshold for performing reproduction was 11, which forced each created agent to win a fight before mating. Operation of fighting transferred at most 10 points of energy. The chance to mutate an offspring was set to 0.75, the mutation influenced 10% of genome within the range of 5%.

Optimizing the Rastrigin benchmark function [49] is a computationally intensive task. By varying the dimension of the problem, we can choose appropriate computation to communication ratios to identify scalability problems related with communication.

Our initial implementations were not very scalable, as throughput for all the variants increased only up to around 8 cores. Above this threshold, only the hybrid version further improved (Fig. 8).

It was not easy to profile the code and look for bottlenecks, as most of common Erlang profiling tools are not designed for HPC programs. We managed to monitor the schedulers' load which was constantly 100% and employed popular Erlang profiling tools for this task. We used *eprof* and *fprof*, which are the main profiling tools shipped with Erlang. The main difference between them is the amount of information they provide and the overhead introduced to the programme, *fprof* is much more significant in both of these traits. We also tried *percept2*, which is a third-party library very popular in the Erlang community. It provides typical profiling information with additional visualizations that make it stand out from the competition. All of those tools showed nothing peculiar in the behaviour of our system and focused on the distribution of computation time between function and different processes.

We also used the *lcnt*[11] tool. This library provides functions to monitor lock contention in the virtual machine and provides valuable information concerning the number of collisions and time spent on each lock.

After running our application with lock monitoring enabled, we discovered that the concurrent model suffers from extensive usage of the *make_ref/0* function. In Erlang, this function is responsible for generating a globally unique identifier and has to be synchronized between schedulers. It is also used by the *gen_server:call/2* function, which was used extensively in our programme. In the concurrent model, arenas have been implemented as *gen_server* behaviours, which use the *gen_server:call/2* function to perform bidirectional communication. Therefore, every time agent-arena communication occurred, the *make_ref/0* function had to be called, which resulted in a big performance loss. Changing communication to use the *gen_server:cast/2* (unidirectional calls) function proved essential to good scalability.

Another significant change was the introduction of the *exometer*[12] library as a global logging system. Thanks to its built-in counters, we managed to lighten the arenas. However, statistics gathering was very difficult due to the distributed and decentralized nature of our algorithm. After several approaches, we decided to use Erlang NIF functions. The native implemented functions (NIF) in Erlang are one of a few ways to introduce C code into an Erlang application. They look like any other function to the caller, however they can be implemented in C enabling significant speed improvements. After employing them for fitness logging we experienced a performance boost and no additional scaling overhead.

The performance improvement of the concurrent version after changes in communication and logging is shown in Fig. 9.

However, no locks were observed in the skel model. Therefore, we experimented with the skeleton structure in order to increase performance. The main challenge was related to agent migrations – it was impossible to implement them through message passing, because skel processes are transparent for the programmer and it is difficult to get their PIDs. Our first approach consisted of adding a synchronization barrier between islands and performing synchronized migrations after each algorithm iteration (Listing 4). This simple solution proved to be not scalable, as the synchronous migration become a bottleneck and slowed down the computation significantly.

In order to solve this problem, we implemented migration separately using ETS tables and removed the synchronization point. In this approach each island iterated in its own loop and asynchronous communication was emulated through writing and reading the ETS tables (Listing 5). The performance improvement of the skel version after reordering skeletons is shown in Fig. 10.

Skel patterns and workflows look very simple for the user, however underneath they involve a lot of message passing through many different processes. In Erlang every outgoing message has to be copied in memory, therefore sending large messages may be computationally expensive. However, this is not the case for large instances (more than 64 bytes) of the Erlang binary data type. Large Erlang binaries are held in a separate space in memory and sent by reference instead of by value. In order to further improve skel version's performance, we moved from a list representation of agents to binary types. We experienced a significant performance and scalability boost, as most of communicational overhead vanished (Fig. 11).

Fig. 12 shows the final scalability of all three implementations. Achieved scalability makes it possible to efficiently utilize a 64 core CPU. These results prove that it is possible to successfully use Erlang for computations on many-core architectures, however getting linear scalability of a particular algorithm is not straightforward.

[11] http://www.erlang.org/doc/man/lcnt.html.

[12] http://github.com/Feuerlabs/exometer.

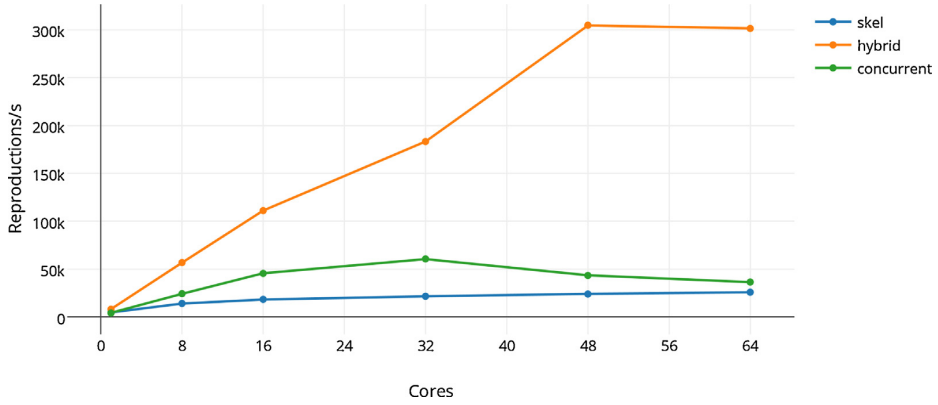


Fig. 8. Throughput of initial implementations. Only the hybrid version scales to some extent.

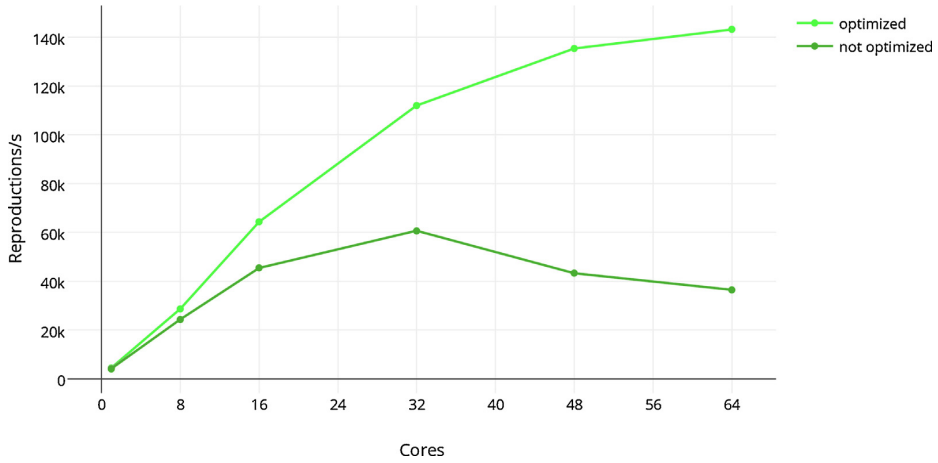


Fig. 9. Throughput of the concurrent model after optimizing both communication and logging.

```
1  Map = {map, [MainWorkflowFun], Workers},
2  Feedback = {feedback, [Map, MigrationFun], StopConditionFun},
3  skel:do([Feedback], [Population]).
```

Listing 4. First approach to skel-based programme implementation. Map skeleton is nested within the Feedback skeleton imposing a barrier after every loop, but enabling easy logging and migration.

```
1  Feedback = {feedback, [MainWorkflowFun], StopConditionFun},
2  Map = {map, [Feedback], Workers},
3  skel:do([Map], [Population]).
```

Listing 5. New skel-based implementation logic. Here the Feedback skeleton is nested in the Map skeleton (inversely to the previous listing) removing the barrier, but enforcing the logging and migration to be implemented differently.

6. Urban traffic management

In order to confirm the scalability features of the developed framework, a more complex, real-life problem has been considered. The computing framework has been applied for optimizing traffic on a simulated urban crossroad. This use case was selected as being a much more practical one than popular and acclaimed, yet quite artificial benchmark functions. Moreover, this problem employs a data-intensive fitness function, again, contrary to a very simple-to-compute Rastrigin function (or similar benchmarks).

Traffic affects all people living in crowded cities, but despite many years of research in this area, most of crossroads are still controlled by simple sequence-based traffic lights. The nature of the problem makes it very hard to be solved both safely and efficiently. Efficient coordination of motion for tens of cars requires complex computations, while high dynamics puts very strong constraints on computations time and robustness. Moreover, the plans created are never executed accurately, therefore some uncertainty must be explicitly accounted for in the planning algorithm. These factors make the classic, domain independent planning methods [35] unsuitable.

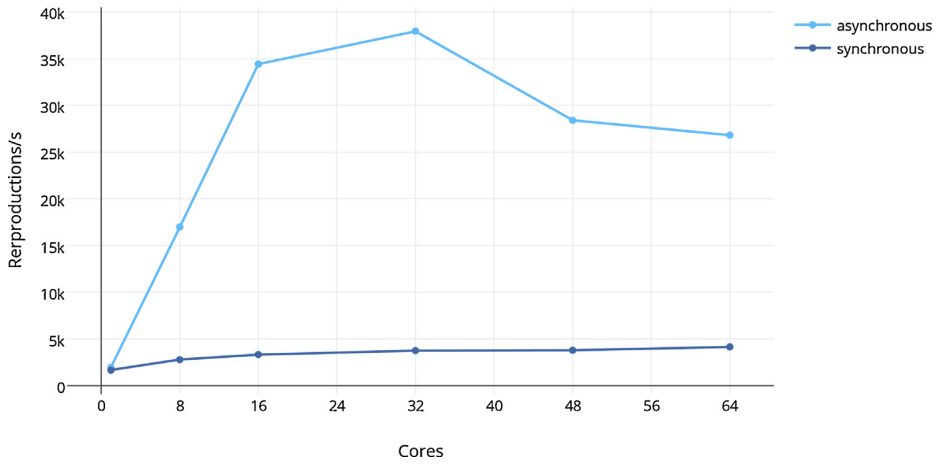


Fig. 10. Throughput of the skel model depending on skeleton structure and agent migration approach: synchronization barrier (synchronous) or ETS tables (asynchronous).

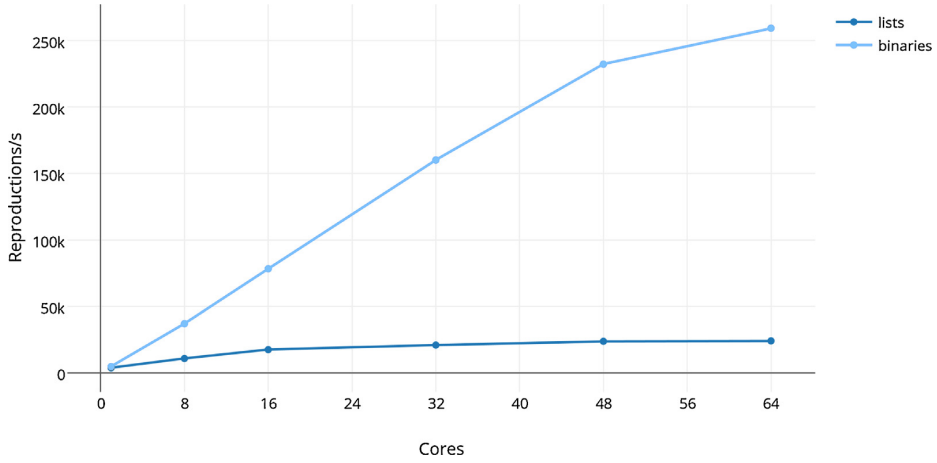


Fig. 11. Performance boost of the skel version after changing agent representation from list (sent by value) to binaries (sent by reference).

Unpredictable changes in the problem parameters can be addressed with approaches defined as *planning under uncertainty*. The methods assume that a planner does not have complete knowledge required to calculate a plan or the knowledge is uncertain. The solutions to this class of problems have to address an issue of uncertainty modeling [50]. An interesting example of a solution for mobile robot motion planning is presented in [51]. The planning algorithm handles sensing and motion uncertainty and optimizes

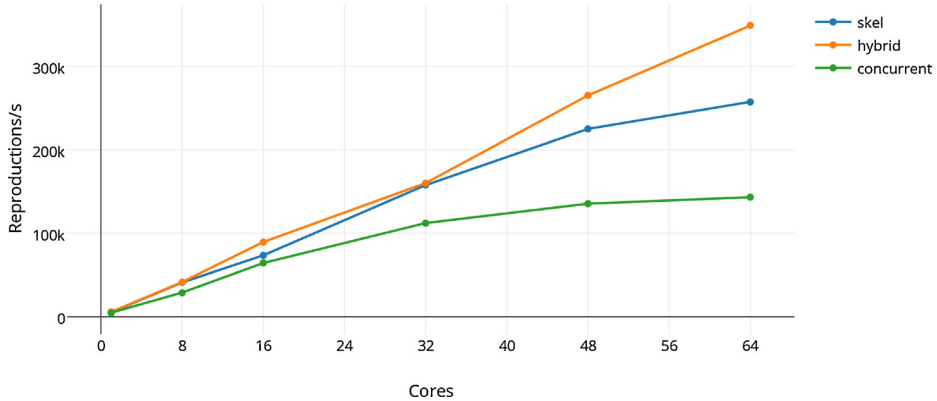


Fig. 12. Performance after all optimizations.

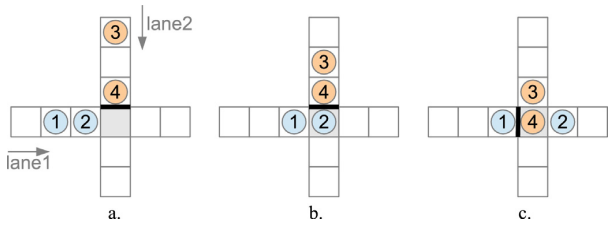


Fig. 13. Three steps of a plan for two lanes crossing.

motion plan with the complexity of $O[n^6]$ in a search space of n dimensions. Such methods cannot be applied for real-time motion planning when a group of independent vehicles is considered.

The problems of managing **entity groups** under uncertainty are also receiving attention, especially in the domain of mobile robotics. Tasks like formation control [52] or motion coordination [53] require robust planning and execution methods. In order to guarantee safety, the solutions tend to simplify the planning algorithm or apply behavioural controllers, which can respond quickly in case of an unexpected execution error. This trade off between robustness and optimality of solutions cannot be overcome as long as the plan is prepared on-demand, when the unexpected situation has already occurred.

Planning in dynamic environments is also studied in the case of scheduling problems, e.g. Job Shop and similar ones, where it is necessary to receive new, unplanned jobs in certain time periods, and/or deal with potential machinery breakdowns. Such approach was usually called a rolling horizon procedure where a rolling time window is introduced and newly arriving jobs are included in the prediction window. Based on the predictions, schedules are prepared (sub-problems of the Job Shop Scheduling Problem are solved) and final schedule is integrated in the current global solutions [54,55]. In these cases, a shifting bottleneck heuristic is used for scheduling and rescheduling [56]. Such heuristics are of course very useful, and usually good-enough for the manual solving of such problems, but in the approach presented here, we would like rather to use a general-purpose evolutionary algorithms (in particular EMAS) to schedule the JSSP, however none of other possible heuristics are excluded and they may be considered in the future.

6.1. Evolutionary crossroads lights management

Traffic management and in particular crossroads lights management problem can be defined as optimization one, and its high level of complexity and potential difficulty of solving (because of its highly dynamic nature) may be approached with general purpose metaheuristics, as evolutionary algorithms. In the case described here, traffic management of certain crossroads is encoded into the fitness function and optimized using EMAS. This is a first step towards further introduction of predictive multi-variant planning that will be one of our future work issues (preparing plans before they are needed, based on certain features of the earlier observed traffic, its participants, environmental conditions, etc.).

In the described case, we assume discrete time and space and we use simplified cars motion model. Each car occupies a singe cell of a road lane and in each time step, it moves to the succeeding cell if the cell is empty. A controlling element is located in the cells where two lanes cross. It can block one lane and enable the other. The plan for controlling the whole crossroad in a single time-step is a set of decisions concerning the controlling elements. The plan for controlling the whole crossroad for a period of time is a sequence of single-step plans. Three steps of a plan executed in a two-lanes crossing are shown in Fig. 13. The plan is (1, 1, 2), which means that during the first two time-steps only cars moving on lane1 can enter the crossing. In the third time-step (situation c) the plan enables motion of cars 3 and 4 moving on lane2, stopping car number 1.

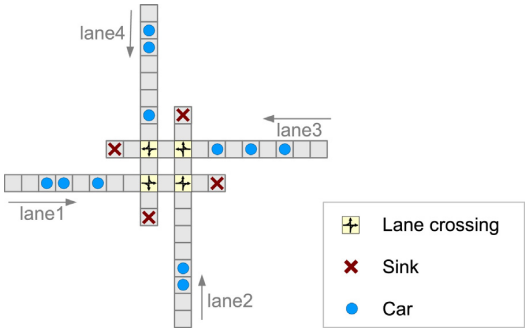


Fig. 14. The crossroad layout used in the experiments.

The efficiency of the plan is calculated as the number or cells covered by all the cars. Calculating the efficiency is a relatively complex task, which requires executing a simulation of cars according to a particular plan. Its complexity is $O(n^2 * t)$, where n is the number of cars and t is the number of time-steps in the plan.

The evolutionary algorithm used for optimizing the plan operates on individuals represented as

$$i = ((x_1^1, \ldots, x_m^1), \ldots, (x_1^t, \ldots, x_m^t)),$$

where m is the number of lanes crossings to control and $x_i^j \in \{1, 2\}$ determines which of the two lanes is blocked in the particular time-step. Each individual is therefore a sequence of $m * t$ bits, which makes variation operators definition straightforward. We use single-point crossover operation and bit-flip mutation. Each variation operator generates one or two new individuals (new plans) which must be evaluated. Multiple variants of possible future situations on the crossroad can be handled using more complex plan evaluation method. Assuming that each car can fail to reach desired location on time, we can generate several possible variants of the situation on the crossroad. Then, the evaluation of a plan must involve calculating the efficiency of the plan in all v variants of situation, increasing the complexity to $O(n^2 * t * v)$. The total fitness should be calculated as a sum of fitness values for all variants. This approach promotes more universal plans, which do not cause significant loss of efficiency when slight differences in the situation occur.

6.2. Scaling the memory-intensive operations

The optimized version of the Erlang computing framework, presented in Section 5, has been applied to the presented traffic planning problem. We used a single crossroad setup, which consisted of four lanes with four independent lane crossings, as presented in Fig. 14. The incoming lanes length was adjusted according to the number of cars, which varied from 10 to 100. Fitness value was calculated as the total number of steps covered by all the cars after 20 time-steps of the plan.

The main difference from the previous use-case is the memory-intensive fitness function. Every fitness evaluation requires performing a entire traffic simulation, which involves intensive memory manipulation. This observation makes the choice of an appropriate data structure very important. Different Erlang built-in data structures can have a crucial influence on the programme scalability and performance.

Erlang ships with several types of key-value stores, the main ones are: *dicts*, *gb_trees*, *ETS tables* and *proplists*. The first two are typical general-purpose key-value data structures differing

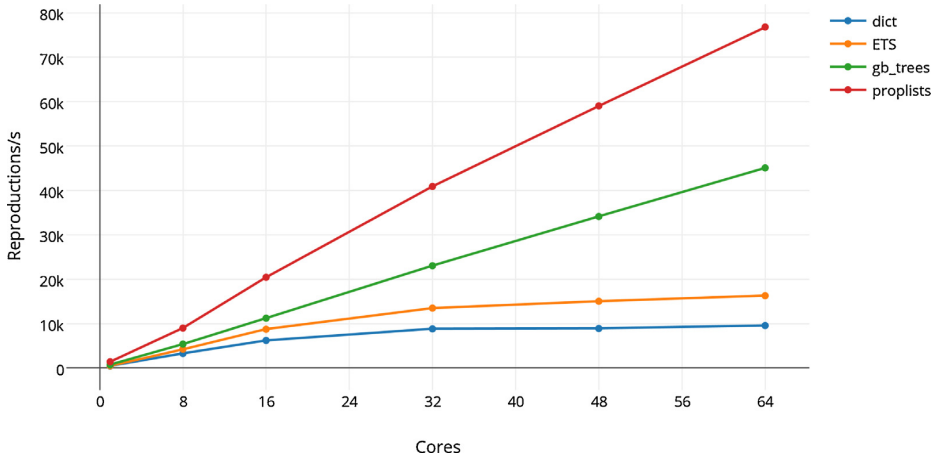


Fig. 15. Performance of different data structures in traffic simulation using the concurrent model.

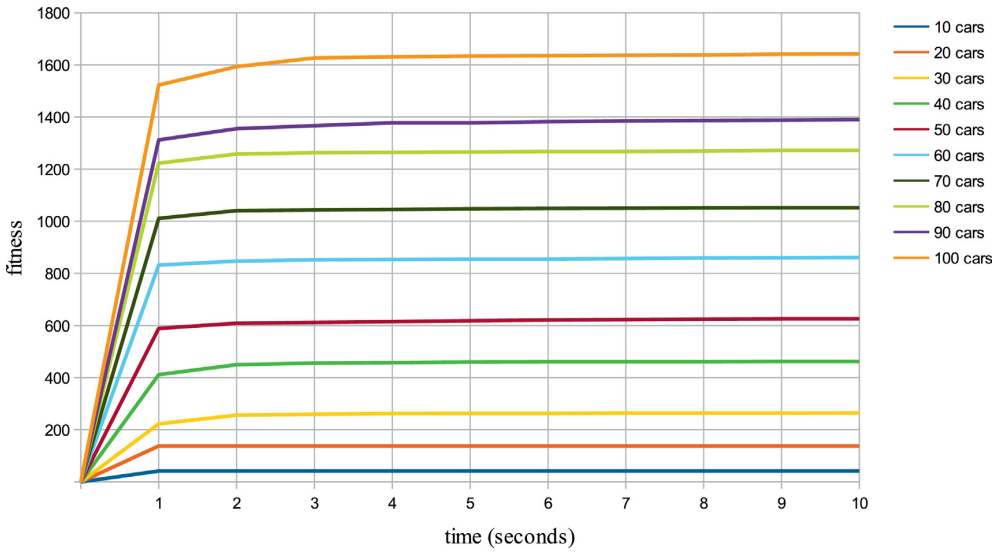


Fig. 16. Fitness convergence for different problem sizes during first 10 s.

in implementation: *dicts* use a hash table and *gb_trees* General Balanced Trees [57]. *ETS tables*, on the other hand, are fast and thread-safe structures for storing large quantities of data and *proplists* are just standard Erlang lists which are fast and lightweight, but with linear access time. Our initial approach used *dicts* as a major container for the simulation data, however we observed very poor performance and scalability.

The hardware setup and the evolutionary algorithm configuration was exactly the same as in case of Rastrigin function optimization described before. Each configuration has been executed 30 times for a time of 3 min in order to obtain statistical significance. Fig. 15 compares the efficiency (median value) of different data structures.

Surprisingly, Erlang *proplists* are clearly the fastest solution for the use case. The reason might be that the structure itself is fairly small (one element per car, not more than 100), therefore the hashing functions used in other data structures are computationally more expensive than iterating over a small list. Another explanation could be that the smaller size and more predictable memory pattern of this data structure makes it more cache-friendly on multi-core hardware. It is also worth noting that *ETS* performed slightly better than *dicts*, but significantly worse than *gb_trees*. Only *proplists* and gb_trees allow linear scalability of the algorithm on a many-core architecture.

6.3. Traffic management results

We have carried out tests to evaluate the speed and convergence of fitness for this particular usecase. The application of the algorithm in the problem of traffic management requires providing good solutions within specified time, therefore the performance improvements are crucial. Median of fitness convergence for different problem sizes computed on 64 cores is shown in Fig. 16.

The results show, that the method converges very fast, reaching stable solutions after 1–3 s, depending on problem size. The quality of found solutions is compared to the results of the fixed-cycles approach in Table 1.

The system managed to calculate solutions better by 8–34%, which is definitely a significant improvement of traffic efficiency. More complex crossroads with several lanes could probably

Table 1
Comparison of traffic efficiency with fixed-cycles approach (traffic lights) and the evolutionary traffic management.

Number of cars:	10	20	30	40	50	60	70	80	90	100
Cycle each step	25	126	194	313	480	787	962	1047	1100	1347
Cycle of 2 steps	25	112	194	396	522	753	962	1047	1164	1364
Cycle of 5 steps	32	98	194	371	501	774	845	1025	1126	1347
Plan after 3 s (median of 30 runs)	42	138	260	456	612	852	1043	1262.5	1367	1626.5
Plan after 3 s (standard deviation)	0.00	0.00	1.27	2.58	2.77	3.36	3.46	3.02	11.40	8.03
Percentage of improvement over best cycle approach	31.25	9.52	34.02	15.15	17.24	8.26	8.42	20.58	17.44	19.24

benefit even more. Presented results constitute a good basis for further research on optimization-based traffic planning, which will require solving problems of representing more realistic motion and control models.

7. Conclusions

Metaheuristics, especially general-purpose ones are crucial methods for solving hard optimization problems. Agent-based metaheuristics turned out to be an effective tool for dealing with such problems, therefore looking for efficient ways of implementing of agent-oriented frameworks becomes an important endeavour. Moreover, in the era of multi-core and many-core architectures, leveraging functional approach and scalable techniques may lead to reaching efficient, easy to use tools that can be adapted to many computing techniques and problems.

Erlang technology is frequently used for building large-scale systems successfully utilizing clusters of computers and multi-core processors in a reliable setting with very small maintenance overhead. However, the most ambitious (industrial) deployments of Erlang applications utilize 24 CPU cores, and even the creators of Erlang do not test the EVM on architectures consisting of more than 32 cores. Some academic research conducted shows results of scaling Erlang up to 64 cores and beyond (using e.g. Intel Xeon Phi architecture), but these reports are based mostly on low-level technology-oriented benchmarks.

Looking for new applications of the Erlang-based frameworks (like parallel computing), encourages to undertake new challenges to efficiently utilize existing supercomputing hardware – thus the approach to efficiently scale such a framework on 64 CPU core machine.

In this paper we presented the initial assumptions regarding the structure and the behaviour of the implemented system (population-based agent-oriented computing framework) pointing out the potential scalability problems and devising three versions of the framework. The procedure of profiling and refactoring led us to good scalability, beyond the acclaimed 24 CPUs. Starting from good scalability only on 8 CPU cores, the search for bottlenecks, monitoring of schedulers' load with predefined tools and search for locks in the running code were performed. As a result of this research, the existing framework was appropriately modified, and eventually the assumed scalability was reached. The performance was verified on both benchmark Rastrigin function and a real-life traffic optimization problem.

The main conclusion of the presented work is that it is possible to implement a computationally intensive applications in Erlang, which scales up to 64 cores, however, achieving this is not a straightforward process. We are convinced, that our experience gathered in the course of this research may be utilized by other researchers and developers preparing highly scalable computation-intensive frameworks or applications based on the Erlang technology. In the future we plan to extend our knowledge by coping with computing in a heterogeneous hardware environment using the available computing frameworks.

The comparison of our framework and the achieved results using Erlang to other frameworks found in the literature is quite difficult as other frameworks are usually implemented having other goals: e.g. management of the computing process, instead of realization of the computing itself. One good comparison would be to cite the results achieved by us in the case of other technologies: namely we are in the course of implementing of similar frameworks in Scala and Haskell, and they currently scale linearly up to 12 cores.

In the future, from the technological point of view, we are planning to try to further scale our platform using Intel Xeon Phi architecture (preliminary runs allowed to achieve almost linear scalability up to 244 cores), and from the substantial point of view, we are going to further explore the multi-variant planning problem, treating the results presented in this paper as a good basis and encouragement for its efficient implementations.

Acknowledgements

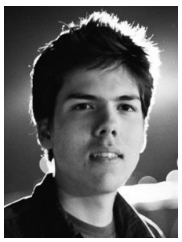
The research presented in the paper was conducted using PL-Grid Infrastructure (http://www.plgrid.pl/en). The research presented in the paper has received founding from the Polish National Science Centre under grant DEC-2011/01/D/ST6/06146. The authors would like to express heartfelt gratitude to a member of the OTP team, Lukas Larsson, for his advices on profiling Erlang code.

References

[1] K. Cetnarowicz, M. Kisiel-Dorohinicki, E. Nawarecki, The application of evolution process in multi-agent world (MAW) to the prediction system, in: M. Tokoro (Ed.), Proc. of the 2nd Int. Conf. on Multi-Agent Systems (ICMAS'96), AAAI Press, 1996.
[2] A. Byrski, M. Kisiel-Dorohinicki, E. Nawarecki, Agent-based evolution of neural network architecture, in: M. Hamza (Ed.), Proc. of the IASTED Int. Symp.: Applied Informatics, IASTED/ACTA Press, 2002.
[3] G. Dobrowolski, M. Kisiel-Dorohinicki, E. Nawarecki, Some approach to design and realisation of mass multi-agent systems, in: R. Schaefer, S. Sedziwy (Eds.), Advances in Multi-Agent Systems, Jagiellonian University, 2001.
[4] M. Kisiel-Dorohinicki, G. Dobrowolski, E. Nawarecki, Agent Populations as Computational Intelligence, in: Neural Networks and Soft Computing: Proceedings of the Sixth International Conference on Neural Networks and Soft Computing, Zakopane, Poland, June 11–15, 2002, Physica-Verlag HD, Heidelberg, 2003, pp. 608–613.
[5] A. Byrski, R. Schaefer, M. Smołka, Asymptotic guarantee of success for multi-agent memetic systems, Bull. Pol. Acad. Sci. – Tech. Sci. 61 (1) (2013).
[6] L. Siwik, R. Dreżewski, Agent-based multi-objective evolutionary algorithms with cultural and immunological mechanisms, in: W.P. dos Santos (Ed.), Evolutionary Computation, In-Teh, 2009, pp. 541–556.
[7] A. Byrski, M. Kisiel-Dorohinicki, Immunological selection mechanism in agent-based evolutionary computation, in: M.A. Klopotek, S.T. Wierzchon, K. Trojanowski (Eds.), Intelligent Information Processing and Web Mining: Proceedings of the International IIS: IIPWM' 05 Conference, Gdansk, Poland, Advances in Soft Computing, Springer Verlag, 2005, pp. 411–415.
[8] K. Wróbel, P. Torba, M. Paszyński, A. Byrski, Evolutionary multi-agent computing in inverse problems, Comput. Sci. 14 (3) (2013) 367 https://journals.agh.edu.pl/csci/article/view/67.
[9] A. Byrski, Tuning of agent-based computing, Comput. Sci. 14 (3) (2013) 491 https://journals.agh.edu.pl/csci/article/view/91.
[10] J. Jeffers, J. Reinders, Intel Xeon Phi Coprocessor High-Performance Programming, Elsevier Science, 2013.
[11] A. Varghese, B. Edwards, G. Mitra, A. Rendell, Programming the adapteva epiphany 64-core network-on-chip coprocessor, in: Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, 2014, pp. 984–992.
[12] EZchipSemiconductor, Highest core-count arm processor optimized for high performance networking applications, http://www.tilera.com/products/?ezchip=585&spage=686 (04 2015).

[13] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2013.
[14] W. Turek, Erlang as a high performance software agent platform, Adv. Methods Technol. Agent Multi-Agent Syst. 252 (2013) 21.
[15] F. Bellifemine, A. Poggi, G. Rimassa, JADE: A FIPA2000 compliant agent development environment, in: Proceedings of the Fifth International Conference on Autonomous Agents, AGENTS' 01, ACM, New York, NY, USA, 2001, pp. 216–217, http://dx.doi.org/10.1145/375735.376120.
[16] A. Pokahr, L. Braubach, W. Lamersdorf, Jadex: Implementing a BDI-infrastructure for JADE agents, EXP – Search Innov. (Special Issue on JADE) 3 (1) (2003) 76–85.
[17] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, G. Balan, Mason: a multiagent simulation environment, Simulation 81 (7) (2005) 517–527, http://dx.doi.org/10.1177/0037549705058073.
[18] M. North, N. Collier, J. Ozik, E. Tatara, C. Macal, M. Bragen, P. Sydelko, Complex adaptive systems modeling with repast simphony, Complex Adapt. Syst. Model. 1 (1) (2013) 3, http://dx.doi.org/10.1186/2194-3206-1-3 http://www.casmodeling.com/content/1/1/3.
[19] K. Pietak, M. Kisiel-Dorohinicki, Agent-based framework facilitating component-based implementation of distributed computational intelligence systems, in: Transactions on Computational Collective Intelligence X, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 31–44.
[20] D. Krzywicki, W. Turek, A. Byrski, M. Kisiel-Dorohinicki, Massively concurrent agent-based evolutionary computing, J. Comput. Sci. 11 (2015) 153–162, http://dx.doi.org/10.1016/j.jocs.2015.07.003 http://www.sciencedirect.com/science/article/pii/S1877750315300041.
[21] M. Kazirod, W. Korczynski, A. Byrski, Agent-oriented computing platform in python, in: 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), vol. 3, 2014, pp. 365–372, http://dx.doi.org/10.1109/WI-IAT.2014.190.
[22] O. Gutknecht, J. Ferber, Madkit: a generic multi-agent platform, in: Proceedings of the Fourth International Conference on Autonomous Agents, AGENTS' 00, ACM, New York, NY, USA, 2000, pp. 78–79, http://dx.doi.org/10.1145/336595.337048.
[23] S. Cahon, N. Melab, E.-G. Talbi, Paradiseo: a framework for the reusable design of parallel and distributed metaheuristics, J. Heuristics 10 (3) (2004) 357–380.
[24] Agent-based parallel computing in java proof of concept, Tech. Rep. TR-UNL-CSE-2001-1004, University of Nebraska–Lincoln, 2001.
[25] F. Cicirelli, L. Nigro, An agent framework for high performance simulations over multi-core clusters, in: AsiaSim 2013: 13th International Conference on Systems Simulation, Singapore, November 6–8, 2013, Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 49–60.
[26] B. Cosenza, G. Cordasco, R. De Chiara, V. Scarano, Distributed load balancing for parallel agent-based simulations, in: 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2011, 2011, pp. 62–69.
[27] S. Cahon, E. Talbi, N. Melab, Paradiseo: a framework for parallel and distributed biologically inspired heuristics, in: in: Parallel and Distributed Processing Symposium, 2003. Proceedings, 2003, p. 9.
[28] D. Thomas, Functional programming-crossing the chasm? J. Object Technol. 8 (5) (2009) 45–48.
[29] F. Cesarini, S. Thompson, Erlang Programming, O'Reilly Media, Inc., 2009.
[30] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, I.E. Venetis, A scalability benchmark suite for Erlang/otp, in: Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop, Erlang' 12, ACM, New York, NY, USA, 2012, pp. 33–42, http://dx.doi.org/10.1145/2364489.2364495.
[31] K. Sagonas, K. Winblad, More scalable ordered set for ETS using adaptation, in: Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang, Erlang' 14, ACM, New York, NY, USA, 2014, pp. 3–11, http://dx.doi.org/10.1145/2633448.2633455.
[32] S. Zheng, X. Long, J. Yang, Using many-core coprocessor to boost up Erlang VM, in: Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang, Erlang' 13, ACM, New York, NY, USA, 2013, pp. 3–14, http://dx.doi.org/10.1145/2505305.2505307.
[33] D. Krzywicki, L. Faber, A. Byrski, M. Kisiel-Dorohinicki, Computing agents for decision support systems, Future Gener. Comp. Syst. 37 (2014) 390–400.
[34] E. Cantú-Paz, A survey of parallel genetic algorithms, Calculateurs Paralleles, Reseaux et Systems Repartis 10 (2) (1998) 141–171.
[35] S. Russel, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 2003.
[36] R. Sarker, T. Ray, Agent-Based Evolutionary Search, 1st Edition, vol. 5 of Adaptation, Learning and Optimization, Springer, 2010.
[37] S.-H. Chen, Y. Kambayashi, H. Sato, Multi-Agent Applications with Evolutionary Computation and Biologically Inspired Technologies, IGI Global, 2011.
[38] P. Uhruski, M. Grochowski, R. Schaefer, Multi-agent computing system in a heterogeneous network, in: Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC 2002), IEEE Computer Society Press, Warsaw, Poland, 2002, pp. 233–238.
[39] J. Liu, Y. Tang, Y. Cao, An evolutionary autonomous agents approach to image feature extraction, IEEE Trans. Evol. Comput. 1 (2) (1997) 141–158, http://dx.doi.org/10.1109/4235.687881.
[40] J. Liu, H. Jing, Y. Tang, Multi-agent oriented constraint satisfaction, Artif. Intell. 136 (1) (2002) 101–144, http://dx.doi.org/10.1016/S0004-3702(01)00174-6 http://www.sciencedirect.com/science/article/pii/S0004370201001746.
[41] E. Alba, B. Dorronsoro, Cellular Genetic Algorithms, Springer, 2008.
[42] M. Garca-Valdez, J. Guervs, F. Fernndez de Vega, Unreliable heterogeneous workers in a pool-based evolutionary algorithm, in: A.I. Esparcia-Alczar, A.M. Mora (Eds.), Applications of Evolutionary Computation, vol. 8602 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 726–737.
[43] J. Jimnez Laredo, D. Lombraa Gonzlez, F. Fernndez de Vega, M. Garca Arenas, J. Merelo Guervs, A peer-to-peer approach to genetic programming, in: S. Silva, J. Foster, M. Nicolau, P. Machado, M. Giacobini (Eds.), Genetic Programming, vol. 6621 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 108–117.
[44] A. Byrski, R. Drezewski, L. Siwik, M. Kisiel-Dorohinicki, Evolutionary multi-agent systems, Knowl. Eng. Rev. 30 (2015) 171–186.
[45] A. Byrski, R. Schaefer, Formal model for agent-based asynchronous evolutionary computation, in: IEEE Congress on Evolutionary Computation, 2009. CEC' 09, 2009, pp. 78–85, http://dx.doi.org/10.1109/CEC.2009.4982933.
[46] J. Banks, J. Carson, B. Nelson, D. Nicol, Discrete-Event System Simulation, Prentice Hall, 2005.
[47] V. Janjic, A. Barwell, K. Hammond, Using Erlang skeletons to parallelise realistic medium-scale parallel programs, in: Proceedings of the Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems.
[48] D. Krzywicki, J. Stypka, P. Anielski, W. Turek, A. Byrski, M. Kisiel-Dorohinicki, et al., Generation-free agent-based evolutionary computing, Procedia Comput. Sci. 29 (2014) 1068–1077.
[49] J. Digalakis, K. Margaritis, An experimental study of benchmarking functions for evolutionary algorithms, Int. J. Comput. Math. 79 (4) (2002) 403–416.
[50] J. Mula, R. Poler, J. Garcí a-Sabater, F. Lario, Models for production planning under uncertainty: a review, Int. J. Prod. Econ. 103 (1) (2006) 271–285.
[51] J. Van Den Berg, S. Patil, R. Alterovitz, Motion planning under uncertainty using iterative local optimization in belief space, Int. J. Robot. Res. 31 (11) (2012) 1263–1278.
[52] T. Balch, C. Arkin, Behavior-based formation control for multi-robot teams, in: IEEE Transactions on Robotics and Automation, 1997, pp. 926–939.
[53] W. Turek, K. Cetnarowicz, W. Zaborowski, Software agent systems for improving performance of multi-robot groups, Fundam. Inf. 112 (1) (2011) 103–117.
[54] J. Fang, Y. Xi, A rolling horizon job shop rescheduling strategy in the dynamic environment, Int. J. Adv. Manuf. Technol. 13 (3) (1997) 227–232.
[55] B. Wang, Q. Li, Rolling horizon procedure for large-scale job-shop scheduling problems, in: IEEE International Conference on Automation and Logistics, 2007, 2007, pp. 829–834, http://dx.doi.org/10.1109/ICAL.2007.4338679.
[56] M. Pinedo, Planning and Scheduling in Manufacturing and Services, Springer, 2009.
[57] A. Andersson, General balanced trees, J. Algorithms 30 (1999) 1–28.

Wojciech Turek, PhD, obtained his PhD in 2010 at AGH University of Science and Technology in Cracow. He works in the area of multi-robot systems, multi-robot planning, autonomous and agent-based systems, concurrent and parallel programming, mostly in functional languages.



Jan Stypka is a MSc student at AGH University of Science and Technology in Cracow, he is interested in parallel and distributed programming and social networks.

Daniel Krzywicki is a PhD student at AGH University of Science and Technology in Cracow, he is interested in agent-based computing and parallel programming in functional languages.

Piotr Anielski obtained MSc in 2013 at AGH University of Science and Technology in Cracow, he is interested in parallel and distributed programming mostly in functional languages.

Kamil Pietak obtained MSc at AGH University of Science and Technology in Cracow, he is interested in agent-based frameworks, software engineering, DSLs and component-based systems.

Aleksander Byrski obtained his PhD in 2007 and DSc (habilitation) in 2013 at AGH University of Science and Technology in Cracow. He works as an assistant professor at the Department of Computer Science of AGH-UST. His research focuses on multi-agent systems, biologically-inspired computing and other soft computing methods.

Marek Kisiel-Dorohinicki obtained his PhD in 2001 and DSc (habilitation) in 2013 at AGH University of Science and Technology in Cracow. He works as an assistant professor at the Department of Computer Science of AGH-UST. His research focuses on intelligent software systems, particularly utilizing agent technology and evolutionary algorithms, but also other soft computing techniques.

3.4. Execution model based on parallel skeletons

The next execution model is based on parallel skeletons. Algorithmic skeletons are an approach which consists in considering programs as a composition and parameterization of higher-order patterns. Such patterns are called skeletons as they define structure, but need to be completed with details [30]. This approach may be summarized as an analysis of a program in order to find patterns that can be replaced with an alternative, more effective implementation, without changing the behavior of the program. For example, consider a function that takes an array at an input, and transforms each element independently to create an output array. An implementation that transforms each element in turn can be converted into an implementation that performs all the transformations in parallel on multiple processors, without changing the behavior of the function itself. If we consider patterns of parallelism, skeletons can be seen as a functional approach to parallelism. Just like referential transparency allows compilers to optimize synchronous functional code, a major topic of study on skeletons is how to use them to rewrite existing code to introduce parallelism without changing the semantics of the program.

The implementation of the third model is thus the effect of applying this type of refactoring, or program transformation, in the first model. Some identified patterns were replaced by parallel implementations from a skeleton library called Skel [31].

The main skeletons used in the implementation of this execution model were Feedback, Map and Farm (Figure 3.6). A single element representing the whole population would loop through the Feedback pattern until a stop condition is met (such as elapsed time or a number of iterations). Within the feedback, the population is shuffled and then partitioned according to behavior using a Map skeleton. For each partition, agents are grouped in pairs and the meetings function is applied to each pair in parallel using the Farm skeleton. Eventually, the results of meetings and partitions are recombined to form the new population.

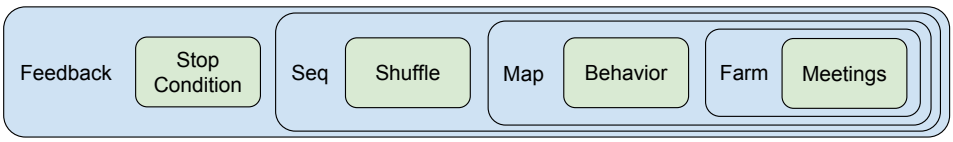


Figure 3.6. In the skeleton-based execution model, a single population is repeatedly processed within a Feedback skeleton until a stop condition is met. At each step, the population is shuffled and then partitioned according to the behavior function by a Map skeleton (The Seq skeleton is just function composition). Then, every partition is divided in groups and for each group the meeting function is computed. The meetings are made parallel with the use of the Farm skeleton. Eventually, the results of meetings and partitions are merged to form the new population.

The resulting model is naturally amenable to parallelism. However, a major limitation of the technology used to implement parallel skeletons was that they were designed to process streams of independent data (e.g. analyzing a stream of images with parallelism at different levels of abstraction). It could not be

used to make subsequent elements interact with each other. In our case, this severely limited the range of skeletons which could be used, as the whole population needed to be the lone element in the flow. As such, there is no more actual concurrency than in the synchronous model.

The following publication provides more information about the Skel library, along with the skeletons used in this model. It also details the model's implementation and experimental evaluation.

Computer Science • 17 (1) 2016 http://dx.doi.org/10.7494/csci.2016.17.1.83

Jan Stypka
Piotr Anielski
Szymon Mentel
Daniel Krzywicki
Wojciech Turek
Aleksander Byrski
Marek Kisiel-Dorohinicki

PARALLEL PATTERNS FOR AGENT-BASED EVOLUTIONARY COMPUTING

Abstract *Computing applications such as metaheuristics-based optimization can greatly benefit from multi-core architectures available on modern supercomputers. In this paper, we describe an easy and efficient way to implement certain population-based algorithms (in the discussed case, multi-agent computing system) on such runtime environments. Our solution is based on an Erlang software library which implements dedicated parallel patterns. We provide technological details on our approach and discuss experimental results.*

Keywords agent-based computing, functional programming, parallel patterns

Citation Computer Science 17 (1) 2016: 83–98

1. Introduction

In the era of multi-core hardware, it is crucial to efficiently and effectively use the possibilities offered by available computing equipment. Over the years, various techniques and tools, such as MPI, have been introduced to construct distributed and parallel systems. They were usually based on imperative and object-oriented programming paradigms. However, it has now become clear that the intrinsic features of functional programming provide a clear advantage in constructing parallel programs. In multi-core environments, it is far easier to program in languages such as Erlang[1] or Scala[2] than in conventional, imperative languages.

In this paper, we consider a functional approach to the implementation of a specific class of computational intelligence systems. Most of the metaheuristic approaches to solving optimization problems (like evolutionary algorithms, particle swarm optimization, immunological algorithms) have potential for parallelism, as they usually consist in processing a large number of individuals. Therefore, provided that the interactions of these individuals are appropriately defined, sequential implementations can be easily replaced with structural parallel alternatives [10]. As an example, parallel evolutionary algorithms are based on the decomposition of a population of individuals into so-called evolutionary islands, which are assigned to particular computing nodes. In agent-based approaches the same happens to agents, which may be distributed among computing nodes [8, 9].

This process seems easy from a conceptual point of view but some practical problems often arise. For example, classical systems implemented using synchronous communication methods (different flavours of remote procedure calls, as e.g. RMI in Java [2]) or asynchronous ones (JMS in Java [1]) require users to design appropriate failure protocols in order to achieve resiliency. Dedicated techniques such as load balancing must also be employed in order to map particular parts of the system into computing nodes, based on the nodes characteristics. Other technological problems may be wrapped up in questions such as “who should start the computing process?”, “who should gather the results?”, “will it become a single point of failure?”, “how to reliably and efficiently communicate with parts of the system?”.

Another important question to be answered is “who should implement these above-mentioned mechanisms?”. If the answer is: the system developer, another question arises: “will the solution be reliable?” or even “should the design of a computing system be focused on technical problems?”.

Fortunately, a number of dedicated software frameworks are now easily available, supporting asynchronous, reliable communication and resilience, among other features. Moreover, technologies such as Erlang, Scala and Akka[3] not only offer the above mentioned features, but also allow to easily use available multi-core and multi-

[1] http://www.erlang.org/
[2] http://www.scala-lang.org/
[3] http://akka.io/

processor machines. Based on such technologies, the designer and developer can truly focus on the nature of the system, not going into excessive technical details.

Such techniques, however, often offer low-level solutions, regarding e.g. concurrency and parallel programming features. It is often more effective for developers to use a high level set of parallel programming patterns in order to speed up the development process, reduce the number of potential bugs and create more flexible and layered implementation. This concept became the basis for the *skel* library, an Erlang tool implementing a pattern-based parallel programming model [5, 7]. That model assumes that a program can be expressed as a workflow constructed of different patterns. The workflow is then supposed to be automatically mapped to available hardware.

In this paper, we focus on presenting an application of the skel library, designed for metaheuristic-based computing, developed in the course of the ParaPhrase FP7 project [15]. We first present a review of related work, along with the relevant computational use-case: Evolutionary Multi-Agent System (EMAS) [8]. We also describe different features of available agent-based computing platforms. Next, we highlight the principles of work of the skel library, then we introduce the actual implementation of agent-based EMAS metaheuristic [9]. Finally, we show experimental results and discuss the scalability of our solution, along with concluding remarks.

2. Parallel and agent-based optimisation metaheuristics

Various models of parallel implementations of evolutionary algorithms have already been proposed [10]. The standard approach (sometimes called a *global parallelization*) consists in distributing selected steps of the sequential algorithm among several processing units. *Decomposition* approaches are based on defining different complex models such as *coarse-grained* and *fine-grained* parallel evolutionary algorithms. There are also methods which use some combination of the models described above (*hybrid* parallel evolutionary algorithms).

Agents play an important role in the integration of artificial intelligence subdisciplines, which is often related to a hybrid design of modern intelligent systems [22]. In most similar applications reported in the literature (see, e.g. [23, 11] for a review), an evolutionary algorithm is used by an agent to support the realization of some of its tasks, often in connection with learning or reasoning, or to support the coordination of some group activity. In other approaches, agents form a management infrastructure for a distributed realization of an evolutionary algorithm [24].

Evolutionary multiagent systems are a hybrid meta-heuristic which combines multiagent systems with evolutionary algorithms. The idea consists in evolving a population of agents to improve its t ability to solve a particular optimization problem [8, 9].

In a multi-agent system no global knowledge is available to individual agents. Agents should remain autonomous and no central authority should be needed. Therefore, in an evolutionary computing system, selective pressure needs to be decentral-

ized, in contrast with traditional evolutionary algorithms. Using agent terminology, we can say that selective pressure is required to emerge from peer to peer interactions between agents instead of being globally-driven.

In EMAS, emergent selective pressure is achieved by giving agents a single non-renewable resource called energy. Agents with high energy are more likely to reproduce, agents with low energy more likely to die. The algorithm is designed to transfer energy from better to worse agents without central control.

In a basic implementation, every agent is assigned with a real-valued vector representing a potential solution to the optimization problem, along with the corresponding fitness.

Agents start with an initial amount of energy and meet randomly. If their energy is below a death threshold, they die. If it is above some reproduction threshold, they reproduce and yield new agents – the genotype of the children is derived from their parents using variation operators and some amount of energy is also inherited. If neither of these two conditions is met, agents fight in tournaments by comparing their fitness values resulting in better agents sapping energy from the worse ones (Fig. 1).

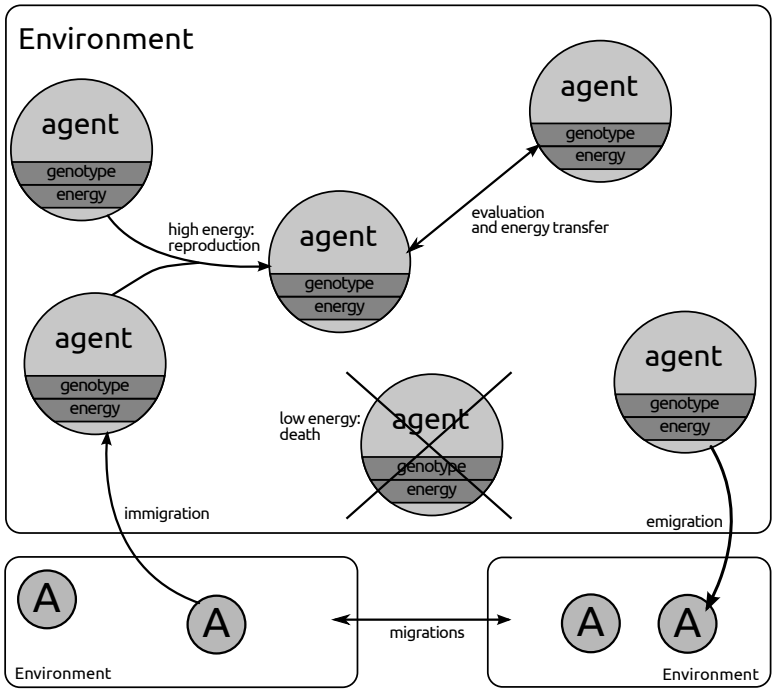


Figure 1. EMAS structure and principle of work.

The system is stable as the total energy remains constant, but the number of agents may vary and adapt to the difficulty of the problem – small numbers of agents with high energy or large numbers of agents with low energy. The number of agents can also be dynamically changed by varying the total energy of the system.

As in other evolutionary algorithms, agents can be split into separate populations. Such islands help preserve diversity by introducing allopatric speciation and can also execute in parallel. Information is exchanged between islands through agent migrations.

EMAS computing abilities were formally proven by constructing a detailed Markov-chain based model and proving its ergodicity [9]. These results show that EMAS is indeed a general optimization tool.

3. Agent-oriented frameworks for computational systems

There are several interesting agent platforms with different purposes. Some of them focus on compliance with the FIPA standard (Foundation for Intelligent Physical Agents), e.g. JADE [3]. Others go in the opposite direction, constructed in a more lightweight way, being better suited for large simulations, e.g., MASON [18]. Some of them provide a large set of built-in features like support for visualization or GIS, e.g. Repast Simphony [19]. Considering aspects of distribution and concurrency, two platforms will be elaborated in deep: Jadex and MaDKit.

Jadex [6] introduces a concept of "active components" — components that are acting as providers and consumers of services and which are active entities with autonomy similar to agents. They communicate with each other through service calls. This system is a good example of a complete distributed and concurrent agent-based platform [21].

The way in which agents in Jadex are implemented results in transparent distribution and concurrency. Services may use remote asynchronous calls instead of local ones. Each service has its own proxy that is responsible for receiving and scheduling calls. On the technical side, remote calls use asynchronous messages between remote management system components. They are encoded using codecs (e.g., binary, XML) and then trasmitted through streams (using any possible transports, e.g., HTTP, TCP). Codecs can also provide advanced functions like encryption or compression.

In MaDKit agents are organized into groups and have some defined roles. The whole platform is centralized around the agent-group-role (AGR) model. Using it, developers build *organizations* which consist of interacting *groups* and *roles* [14].

MaDKit has two important concepts that ease the introduction of distribution and concurrency: micro-kernels and agent-based services. The former is the name of a reduced platform core that executes only the most basic functions: control of groups and roles, lifecycle management of agents, local messaging. More advanced functions must be provided by agents and this is the latter concept in which agents provide the rest of platform services, e.g., distributed message passing, migration. As a result, the platform is extensible and flexible. Additionally, groups can span multiple platform nodes.

The above-mentioned systems are general-purpose tools. For specific applications, efficiency improvements can be achieved by simplifying assumptions concerning system granularity or communication. As such, Jadex and MADKit became

an inspiration for several dedicated agent-based computing frameworks targeted at population-based computing.

The AgE computing framework is an open-source project developed at the Intelligent Information Systems Group of AGH-UST and a starting point for further considerations. AgE is a framework for the development and the run-time execution of distributed agent-based simulations and computations.

In AgE, a computation is decomposed into agents responsible for performing some part of the algorithm. Agents are structured into a tree according to the Composite design pattern [13]. It is assumed that all agents at the same level are being executed in parallel. To increase performance, top level agents can be distributed amongst different nodes along with all their children.

Agents, however, are not atomic assembly units, but they are further decomposed into functional units according to the Strategy design pattern [13]. Strategies represent problem-dependent algorithmic operators and may be switched without otherwise changing the implementation of the agent. Stateless strategy instances may be shared between agents as they provide various services to agents or others strategies.

With the use of the environment, agents can communicate with their neighbours via messages or queries. They may also ask their neighbours to perform specific actions.

In a distributed model, agents are located in so-called workplaces, which are assigned to computing nodes. Workplaces facilitate inter-agent communication and migration between nodes. The workplaces may be implemented according to phase-simulation or can be event-driven [20].

There are several AgE implementations, the most noteworthy are based on Java[4], Python[5] and Erlang[6].

A functional agent-based execution model is a new approach to the design of agent-based computing frameworks [16].

In the platforms and frameworks described before, agent-based systems are usually implemented using an object-oriented or a component-based approach. As such, their design follows the domain of the implemented problem, i.e. a number of interacting individuals, embedded in an environment, being able to perceive and interact among themselves and with the environment they are located in.

However, in the case of computing systems, a number of simplifications can lead to simpler implementations, fully compatible with functional programming languages. Such a functional approach allows to naturally use concurrent and distributed features of such languages and leads to a more efficient execution of a multi-agent system. [17].

In this approach, agents willing to perform similar actions are grouped in separate entities called *arenas*, following the Mediator design pattern [13]. Agents choose and

[4] `http://age.agh.edu.pl`
[5] `https://github.com/maciek123/pyage`
[6] `http://paraphrase.agh.edu.pl`

join an arena depending on their state. Arenas split incoming agents into groups of certain cardinality and trigger the actual actions. Every kind of agent behavior is represented by a separate arena (e.g. in the case of EMAS there are arenas for meeting, reproduction and migration).

The dynamics of the multi-agent system are fully defined by two functions. The first function represents agent behavior and chooses an arena for each agent (mapping step). The second function represents meeting logic and is applied in every arena (reducing step). This approach is similar to the MapReduce model and has the advantage of being very flexible, as it can be implemented in both a centralized and synchronous way or a decentralized and asynchronous one, as we show further below.

4. Skel – general purpose tool for parallelization

An efficient parallel implementation of a complex algorithm is typically a challenging and time-consuming task. It requires significant effort to maximize speedup using software tools for parallel hardware such as operating system threads, shared memory and synchronization mechanisms. In such implementations, the logical structure of the algorithm or the problem is often coupled to the physical architecture of hardware. This is a significant disadvantage, as the decision on how to make a computation parallel should depend on the problem and its size. Moreover, an implementation created for a particular machine is often suboptimal on a different computer architecture. Therefore, coupling the algorithm with the hardware is inflexible and hardware-dependent.

The Skel library was designed to efficiently solve these issues with a different programming model for parallel algorithms. The library is a result of the ParaPhrase FP7 EU project [15]. The project defines a new methodology, based on parallel patterns, for the design and implementation of parallel applications on heterogeneous hardware architectures. A pattern describes a parallel computation by highlighting the functional behavior instead of the implementation details. The patterns are composed by a programmer into algorithmic *skeletons*.

A skeleton is represented as a directed graph of nodes, each of which defines a parallel computational behavior. Thus, a skeleton tree corresponds to a specific pattern of computation, in which the number of nodes and the data distribution policies are explicitly specified. The details related to the implementation on a specific target architecture are hidden. As shown in Figure 2, a parallel application designed as a composition of parallel patterns is mapped to the available hardware resources, and it may be dynamically re-mapped to meet application needs and hardware availability. Moreover, the application can easily be restructured using a refactoring tool such as PaRTE [4] in order to change or improve the used parallel patterns.

The basic parallel patterns of Skel library are:

- Pipe – a sequence of stages, where the output of one stage is an input for the next stage. A single data item is executed in each stage in turn, but separate data items may be executed in different stages in parallel.

- Farm – embarrassingly parallel computations in which every data item can be computed independently of others.
- Map and Reduce – split collective data structure into parts, perform operations on them in parallel and aggregates the results.
- Feedback – a skeleton equivalent of a loop, feeds its output in its input until a stop condition is met.

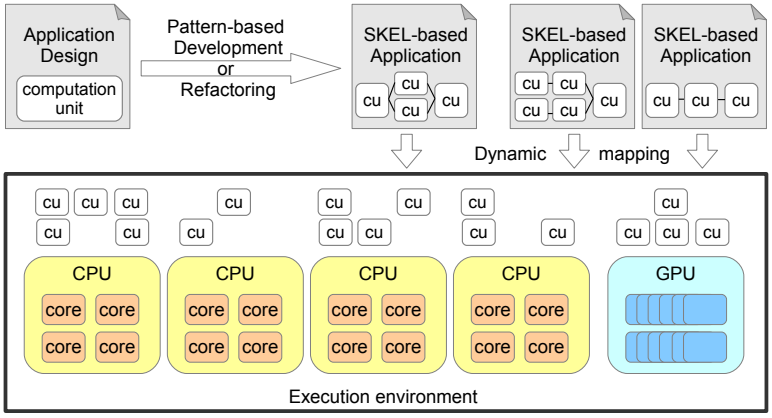


Figure 2. Parallel program execution schema. Application written using Skel as a graph of patterns is dynamically mapped to available hardware.

The Skel library is implemented in Erlang. It is based on typical Erlang mechanisms and provides higher level skeleton abstractions. It accepts a description of the skeleton workflow (which is the application skeleton graph) and an input data stream and processes them to produce the output data stream. The output stream represents the results produced by the parallel execution of the skeleton graph on the input stream items.

This library allows to use parallel hardware with a minimum effort from the programmer. Single pieces of computation, provided as Erlang functions, are composed into a skeleton within a few lines of code. All the problems of process pooling, data management and efficient hardware mapping are solved transparently.

5. Skel-based EMAS implementation

A general algorithm conducted in one of EMAS evolutionary islands may look as follows:

1. Allow each of the agents to conduct a subsequent step of its work.
2. Gather signatures of actions to be performed by the agents: e.g. reproduce, die, migrate.
3. Perform the actions in the order of notification: e.g. produce an offspring based on two agents wanting to reproduce and transfer appropriate amount of energy

from parents, remove a dying agent and distribute its remaining energy among other agents, migrate an agent between the islands.

4. Unless a stop condition is met, return to step 1.

At the same time, a general algorithm of one step conducted in one of EMAS agents may look as follows:

1. With small probability, decide to migrate and notify the evolutionary island accordingly.
2. If the energy level is higher than some reproduction threshold, notify the evolutionary island accordingly.
3. If the energy level is lower than some death threshold, notify the evolutionary island accordingly.
4. Otherwise, meet another agent, compare the fitness values and exchange some energy.

Assuming the existence of several evolutionary islands, the most obvious parallelization strategy is to represent each island as a separated thread or even as a process. Another solution is to introduce parallel execution of the particular types of operations within a single island. Meetings for energy transfer, reproduction and migration are independent and can be executed in parallel. Moreover, even each agent may be implemented completely asynchronously.

Depending on the complexity of the operations to be performed, different types of parallelism may be more efficient. Therefore, it is advantageous to be able to express the multi-agent algorithm in terms of high-level functions and leave out execution details. These high-level functions can be later combined to match a specific problem size and the available hardware resources. The Skel library provides exactly the required mechanisms to achieve this.

The Skel-based EMAS implementation is composed of several simple skeletons nested within each other. It enables a high-level approach as well as easy code development and maintenance.

The main skeleton that enables continuous program iteration is the *feedback* skeleton. It contains a workflow describing one algorithmic cycle and a condition that has to be fulfilled in order for the program to continue the execution. The definition of the main algorithm loop with a time-based stop condition is shown in Listing 1.

Listing 1. The feedback loop of the algorithm.

```
1  StopCondition = fun(_Agents) -> os:timestamp() < EndTime end.
2  Skeleton = {feedback, [MainWorkflow], StopCondition}.
3
4  FinalPopulation = skel:do([Skeleton], [InitialPopulation]).
```

The main workflow embedded in the *feedback* skeleton is a *pipeline* consisting of three main functions (see Listing 2). These operations are executed sequentially

in the first *seq* skeleton of the pipeline. Concrete definitions of the aforementioned functions are shown in Listing 3.

Listing 2. The main workflow of the algorithm.

```
1  MainWorkflow = {pipe, [{seq, GroupAgents},
2                         {map, [{seq, UpdateAgents}], Workers},
3                         {seq, Shuffle}]}.
```

The first function (*GroupAgents*), is responsible for choosing an action for every agent, performing migration between islands and eventually grouping agents with similar behaviors (actions) on the same islands. Agents choose some action (reproduction, fight, death) depending on their state (amount of energy). Agents can also choose to migrate with some low probability.

The second function (*UpdateAgents*) is where all the evolutionary operations are performed and it is parallelized with the *map* skeleton with a predefined number of workers. Each worker processes one agent group at a time applying an appropriate meeting function until all of the groups have been handled.

For every kind of behavior (reproduction, fight, death), a specific meeting function is called. Fights are tournaments in which agents compare fitness and the loser transfers some of its energy to the winner. Reproduction uses classical evolutionary variation operators to derive offspring from existing agents. Death meetings simply yield an empty list to remove the incoming agents from the population.

The third function's purpose is to shuffle the final agent list, so that the interactions in future generations happen between random individuals.

Listing 3. Particular stages of the algorithm.

```
1  GroupAgents = fun (Agents) ->
2              AgentsWithAction = lists:map(ChooseAction, Agents),
3              Migrated = lists:map(Migrate, AgentsWithAction),
4              GroupByAction(Migrated)
5                 end,
6
7
8  UpdateAgents = fun({{Island, Behavior}, Agents}) ->
9                     NewAgents = Meetings({Behavior, Agents}),
10                    [{Island, A} || A~<- NewAgents]
11                end,
12
13 Shuffle = fun(Agents) ->
14                shuffle(lists:flatten(Agents))
15           end.
```

The basic logic and parallel structure of the algorithm can be expressed in approximately 50 lines of code. Even including all the evolutionary operations as well

as logging and other monitoring code, the total volume does not exceed few hundred lines, which is significantly compact.

Thanks to skeletons provided by the Skel library, the implementation is very simple as well as easy to read and maintain. The program is parallelized automatically which reduces boilerplate code and improves readability and clarity of the source files.

6. Experimental results and comparison

6.1. Problem definition

The evaluation focuses on solving a discrete optimization problem, namely finding Low Autocorrelation Binary Sequences, an NP-hard combinatorial problem with a very simple formulation and many applications in telecommunication, meteorology, physics and chemistry [12]. The problem consists in finding a binary sequence $S = \{s_0, s_1, \ldots, s_{L-1}\}$ with length L where $s_i \in \{-1, 1\}$ which minimizes the energy function $E(S)$:

$$C_k(S) = \sum_{i=0}^{L-k-1} s_i s_{i+k} \qquad E(S) = \sum_{k=1}^{L-1} C_k^2(S).$$

6.2. Test organisation

We ran our experiments on the ZEUS supercomputer provided by the Pl-Grid[7] infrastructure at the ACC Cyfronet AGH[8]. We used nodes with 2 Intel Xeon X5650 processors each (12 cores per node) and a total of 24 GB of memory per node. In consecutive experiments, different numbers of cores were used.

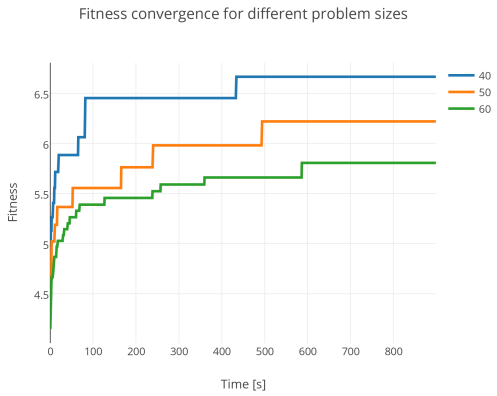
We performed experiments for several CPU configurations and problem sizes. We assessed the weak and strong scalability of our solution by varying problem sizes and used cores. Every experiment was run for 30 minutes and repeated 30 times for statistical significance. The results below are averaged over these runs.

6.3. Experiment results

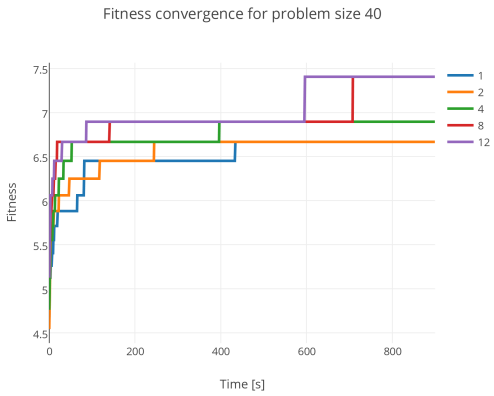
Figure 3(a) shows fitness plots for different problem sizes that have all been run on 1 core. There is no surprise here, the harder the problem, the more time our program needs to improve the solution. Figures 3(b)–(d), on the other hand, show how fitness value converges for different CPU core configurations. One can see significant improvement while adding more computing cores on all problem sizes. Furthermore the difference becomes more visible for larger problems, and the average final fitness values for each experiment are shown in Table 1.

To assess the scalability of the system, we recorded the intensity of interactions in the system, represented by the number of agent reproductions happening every

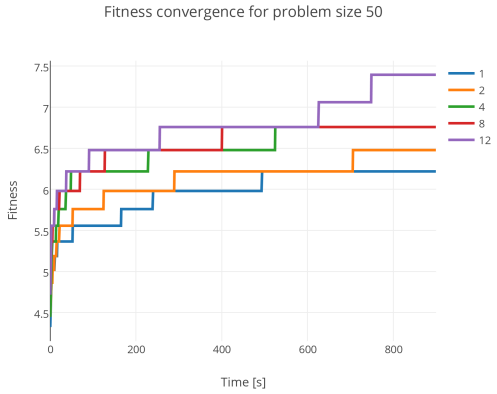
[7] `http://www.plgrid.pl/en`
[8] `http://www.cyfronet.krakow.pl/en/`



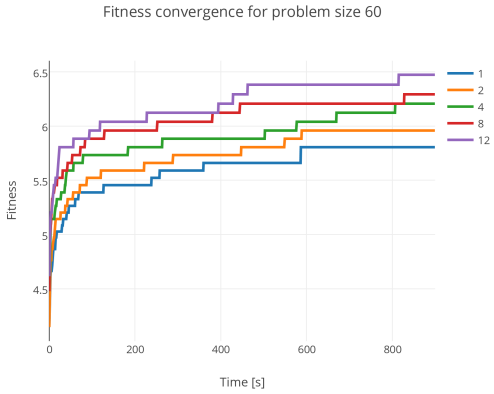
(a) Fitness convergence for different problem sizes on 1 CPU core.



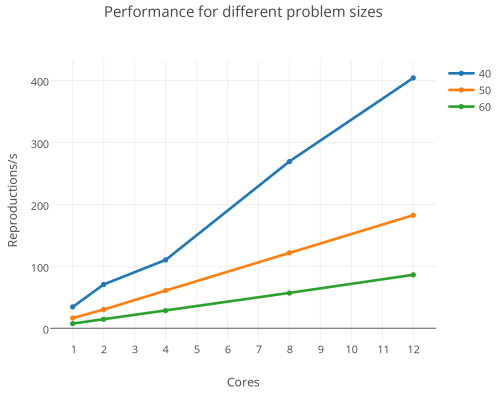
(b) Fitness convergence for problem size 40 on different cores.



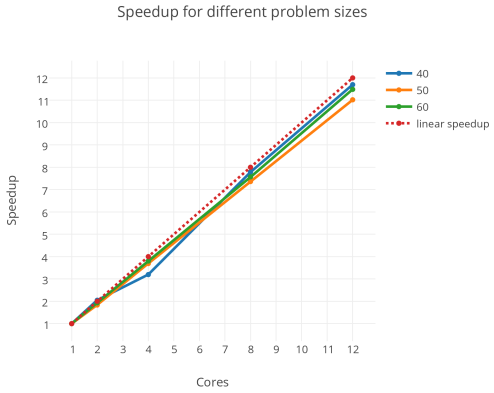
(c) Fitness convergence for problem size 50 on different cores.



(d) Fitness convergence for problem size 60 on different cores.



(e) Reproductions per second for different problem sizes.



(f) Speedup.

Figure 3. Scalability and efficiency of the computation using Skel.

second (Fig. 3e). We can also normalize these values to derive speedup (Fig. 3f), along with an "ideal speedup" reference line. As we can see, scalability is virtually linear for all problem sizes.

Table 1

Average fitness values and their standard error at the end of the experiments, for different problem sizes and number of cores.

Cores	Problem size					
	40		50		60	
1	6.6936	0.0318	6.2240	0.0683	5.8479	0.0474
2	6.7913	0.0523	6.4907	0.0879	5.9742	0.0533
4	6.9127	0.0444	6.7665	0.0676	6.1592	0.0492
8	7.1537	0.0505	6.9047	0.0968	6.3291	0.0578
12	7.2201	0.0449	7.2273	0.1068	6.5007	0.0552

7. Conclusions

Population metaheuristics (e.g. evolutionary or agent-based) are a natural candidate for implementation on parallel computing hardware. A traditional implementation of such systems, using e.g. MPI, is a difficult and error-prone task.

Fortunately, a number of functional technologies, such as Scala or Erlang, can help in an efficient implementation of such systems by changing the perspective. Instead of coupling the algorithm to the underlying hardware, programmers can focus on the problem domain and design multi-agent systems while abstracting from their actual runtime execution.

In this paper, we show how to design an Evolutionary Multi-Agent System in terms of such high-level functions and use parallel patterns and skeletons from the *skel* library in order make the algorithm more efficient on multi-core hardware. However, the algorithm can be easily adapted to different hardware by changing structure of skeletons.

The most important feature of the proposed implementation model is its simplicity. The basic logic and parallel structure of the algorithm can be expressed in approximately 50 lines of code. Our results show that the implemented system was able to efficiently utilize all tested configurations. The algorithm also scales well with the introduction of skeleton parallelism, as increasing the number of cores allows to reach better optimisation results faster.

Future work includes tackling more difficult problems and comparing our results with ones provided by different software platforms.

Acknowledgements

The research presented in the paper was partially supported by the European Commission FP7 through the project ParaPhrase: Parallel Patterns for Adaptive Heteroge-

neous Multicore Systems, under contract no.: 288570 http://paraphrase-ict.eu. *The research presented in this paper received partial financial support from AGH University of Science and Technology statutory project no. 11.11.230.124. The research presented in the paper was conducted using PL-Grid Infrastructure* http://www.plgrid.pl/en.

References

[1] Specification of Java Remote Method Invocation. https://jcp.org/en/jsr/detail?id=368.

[2] Specification of the Java Message Service. http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/spec/rmiTOC.html.

[3] Bellifemine F., Poggi A., Rimassa G.: JADE – A FIPA-compliant agent framework. In: *Proceedings of PAAM*, vol. 99, pp. 97–108, London, 1999.

[4] Bozó I., Fördős V., Horpácsi D., Horváth Z., Kozsik T., Kőszegi J., Tóth M.: Refactorings to Enable Parallelization. In: *Trends in Functional Programming*, pp. 104–121, Springer, Berlin, 2015.

[5] Bozó I., Fordós V., Horvath Z., Tóth M., Horpácsi D., Kozsik T., Köszegi J., Barwell A., Brown C., Hammond K.: Discovering parallel pattern candidates in erlang. In: *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pp. 13–23, ACM, 2014.

[6] Braubach L., Lamersdorf W., Pokahr A.: Jadex: Implementing a BDI-infrastructure for JADE agents. *Exp*, vol. 3(3), pp. 76–85, 2003.

[7] Brown C., Danelutto M., Hammond K., Kilpatrick P., Elliott A.: Cost-directed refactoring for parallel Erlang programs. *International Journal of Parallel Programming*, vol. 42(4), pp. 564–582, 2014.

[8] Byrski A., Dreżewski R., Siwik L., Kisiel-Dorohinicki M.: Evolutionary Multi-Agent Systems. *The Knowledge Engineering Review*, vol. 30(02), pp. 171–186, 2012.

[9] Byrski A., Schaefer R., Smołka M., Cotta C.: Asymptotic guarantee of success for multi-agent memetic systems. *Bulletin of the Polish Academy of Sciences: Technical Sciences*, vol. 61(1), pp. 257–278, 2013.

[10] Cantú-Paz E.: A Survey of Parallel Genetic Algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, vol. 10(2), pp. 141–171, 1998.

[11] Chen S.H., Kambayashi Y., Sato H.: *Multi-Agent Applications with Evolutionary Computation and Biologically Inspired Technologies*. IGI Global, Hershey, Pennsylvania, 2011.

[12] Gallardo J.E., Cotta C., Fernández A.J.: Finding low autocorrelation binary sequences with memetic algorithms. *Applied Soft Computing*, vol. 9(4), pp. 1252–1262, 2009.

[13] Gamma E., Helm R., Johnson R., Vlissides J.: *Design patterns: elements of reusable object-oriented software*. Pearson Education, Harlow, UK, 1994.

[14] Gutknecht O., Ferber J.: The madkit agent platform architecture. In: *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, pp. 48–55, Springer, 2001.

[15] Hammond K., Aldinucci M., Brown C., Cesarini F., Danelutto M., González-Vélez H., Kilpatrick P., Keller R., Rossbory M., Shainer G.: The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems. In: *Formal Methods for Components and Objects*, pp. 218–236, Springer, 2013.

[16] Krzywicki D., Byrski A., Kisiel-Dorohinicki M., et al.: Computing agents for decision support systems. *Future Generation Computer Systems*, vol. 37, pp. 390–400, 2014.

[17] Krzywicki D., Stypka J., Anielski P., Turek W., Byrski A., Kisiel-Dorohinicki M., et al.: Generation-free Agent-based Evolutionary Computing. *Procedia Computer Science*, vol. 29, pp. 1068–1077, 2014.

[18] Luke S., Cioffi-Revilla C., Panait L., Sullivan K., Balan G.: Mason: A multiagent simulation environment. *Simulation*, vol. 81(7), pp. 517–527, 2005.

[19] North M.J., Collier N.T., Ozik J., Tatara E.R., Macal C.M., Bragen M., Sydelko P.: Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, vol. 1(1), pp. 1–26, 2013.

[20] Pidd M., Cassel R.A.: Three phase simulation in Java. In: *Proceedings of the 30th conference on Winter simulation*, pp. 367–372, IEEE Computer Society Press, 1998.

[21] Pokahr A., Braubach L., Jander K.: The jadex project: Programming model. In: *Multiagent Systems and Applications*, pp. 21–53, Springer, Berlin, 2013.

[22] Russell S., Norvig P., Intelligence A.: *Artificial Intelligence: A modern approach*. Prentice-Hall, Egnlewood Cliffs, 1995.

[23] Sarker R.A., Ray T.: *Agent-Based Evolutionary Search*, vol. 5. Springer Science & Business Media, 2010.

[24] Uhruski P., Grochowski M., Schaefer R.: Multi-agent computing system in a heterogeneous network. In: *Parallel Computing in Electrical Engineering, 2002. PARELEC'02. Proceedings. International Conference on*, pp. 233–238, IEEE, 2002.

Affiliations

Jan Stypka
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Krakow, Poland, janstypka@gmail.com

Piotr Anielski
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Krakow, Poland, pr.anielski@gmail.com

Szymon Mentel
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Krakow, Poland, mentel.szymon@gmail.com

Daniel Krzywicki
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Krakow, Poland, krzywic@agh.edu.pl

Wojciech Turek
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Krakow, Poland, wojciech.turek@agh.edu.pl

Aleksander Byrski
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Krakow, Poland, olekb@agh.edu.pl

Marek Kisiel-Dorohinicki
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Krakow, Poland, doroh@agh.edu.pl

Received: 8.02.2015
Revised: 27.06.2015
Accepted: 29.06.2015

3.5. Execution model based on adaptive dataflows

The final execution model is based on adaptive dataflows, where agents are implemented as elements flowing in a variable-rate reactive stream [1]. The major difference compared to parallel skeletons is that because of non-blocking backpressure, the rate of the stream can be locally compacted or expanded to allow neighboring agents to interact.

The architecture of this execution model is divided into two parts. The first is a looping graph which is responsible for iterating the algorithm (Figure 3.7a). Agents are consumed from an initial population, transformed through a Step stage and fed back into the input. When the initial source is depleted, the graph will continuously drain from the feedback loop. In such recursive streams, care must be taken to ensure the liveliness of the algorithm and the resource boundedness of the system. In our case, this is guaranteed by the fact that the total energy in the multi-agent system bounds the possible number of agents in the stream, and an intermediate buffer in the feedback loop can be sized accordingly.

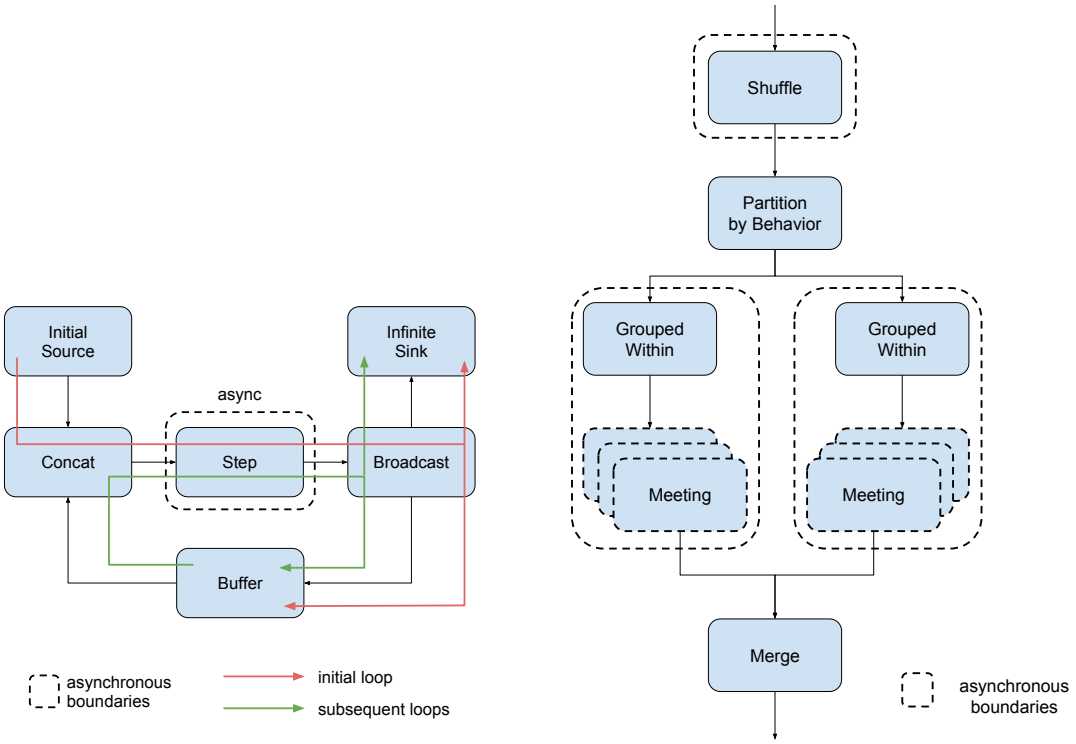
The second part of the architecture is the Step flow (Figure 3.7b). The flow of agents is partitioned by computing the behavior function for each agent. Every partition flows through a substream where subsequent agents are grouped according to the arity of the behavior. This grouping is made possible by the variable rate of reactive streams. The *Grouped Within* stage has a different input and output rate; it will emit a single output containing the last n inputs, or all the inputs observed within a timeout period, whichever happens first. This timeout behavior can lead to uncontrolled non-determinism, but is necessary to ensure the liveliness of the system, for example if only a single agent chooses a particular behavior.

Within each substream, the meetings function is applied on every group. The output of each meeting may consist in modified input agents (fight and reproduction), new agents (reproduction) or no output at all (death). Again, this is made possible by the ability to decouple input and output rates of the stages. Finally, the outputs of all the meetings are merged into a single output. Multiple meetings for the same behavior may be happening in parallel (as defined by a parallelism factor), but the results are kept ordered within a substream. However, as the *Grouped Within* stage already introduces non-determinism, each substream executes in a separate asynchronous context (possibly a different thread or node) for greater efficiency.

Except for the slight non-determinism forced by liveliness requirements, the rest of the stream processing is deterministic. In general, stronger stochastic properties are necessary for metaheuristics such as EMAS [32]. To this purpose, there is an additional *Shuffling* stage in the *Step* flow which is responsible for changing the order of agents in the stream.

It is not trivial to shuffle a (possibly infinite) stream, where by definition we do not know all the elements at any given time. In the following publication, I describe a technique inspired from *Reservoir Sampling* [33]. The stage maintains an internal buffer of the input elements in has seen so far. When an

[1] http://www.reactive-streams.org/



(a) The looping graph (b) The Step flow

Figure 3.7. In the execution model based of adaptive dataflows, agents are modeled as elements flowing in a reactive stream. The first part of the stream is a looping graph which introduces a feedback loop and allows iteration. The Step flow includes a shuffling stage which introduces randomness in the order of stream elements using a technique similar to reservoir sampling. It also includes subflows for different behaviors, with a grouping stage which uses non-blocking backpressure to locally compact the stream and allow subsequent elements to interact.

output element is requested by downstream, it is chosen from within the buffer according to some policy. In the paper, I examine several policies, such as random, min-max or an annealed combination of those.

This execution model combines the benefits of the previous ones. It is possible to have high concurrency like in the actor-based model, however this concurrency can be explicitly controlled for using an appropriate shuffling policy. It is also possible to have high parallelism and scalability, like in the skeleton-based model. I also show that the previous execution models can be simulated by using specific shuffling policies. As such, the dataflow execution model is the most general one and previous execution models can be simulated as specific instances of adaptive dataflows.

The following publication describes in more depth the working details of the implementation of this model. In particular, it illustrates how this model generalizes the other ones. The publication also includes a detailed experimental analysis of the different parameters of the model for two classical optimization problems. As of this writing, the publication has been accepted for publication in the journal "Concurrency and Computation: Practice and Experience".

Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

Concurrent agent-based evolutionary computations as adaptive dataflows

Daniel Krzywicki | Łukasz Faber | Roman Dębski*

Department of Computer Science, AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Kraków, Poland

Correspondence
*Department of Computer Science, AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Kraków, Poland. Email: rdebski@agh.edu.pl

Present Address
Department of Computer Science, AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Kraków, Poland.

Abstract

This paper introduces a new formal description of the execution model for agent-based computing systems, in the form of an adaptive dataflow decoupled from the domain-specific semantics of the computation. We show that the execution models studied in previous work can be unified in this common model. The parameters of the model, such as queuing policies and granularity of the data in the flow are analysed. Several queueing alternatives are benchmarked to demonstrate how they affect the efficiency of the computation. Using the example of a multi-agent evolutionary optimisation problem solver, the new approach is shown to outperform the classic one. This proposed model is well suited to functional languages and can be easily mapped onto different classes of hardware – from simple, single-core computers to distributed environments.

KEYWORDS:
execution model, multi-agent system, evolutionary algorithm, simulated annealing, functional programming, reactive streams

1 | INTRODUCTION

The semantics of an agent-based computation can be defined independently from the underlying execution model of agent interactions [1]. Such a computation, which can be specified functionally, can be then "plugged" into different execution models (for instance a *sequential model*, a *classic island-model*, a *fine-grained (parallel) model* [2] or a *dataflow* [3]). The choice of the execution model of agent interactions can have an impact on the behaviour of the computation as well as on raw performance [1]. By decoupling the semantics of the computation from the execution model, we can meaningfully compare different execution models and tune them to particular execution environments. In consequence, the computation itself can be easily ported to different machines. This is especially important in the case of massively parallel computers.

The exploitation of massive parallelism was the original motivation for the research into *dataflow* (and parallel data-driven computation in general) [4,3]. In the dataflow execution model, a program is represented by a directed graph in which the nodes correspond to some units of computation (operators) and the directed edges to data dependencies (data flows). This execution model is well suited to functional languages and can be easily mapped into different classes of hardware – from simple, single-core computers to distributed environments.

Modelling multi-agent evolutionary computations as static dataflows was investigated within the ParaPhrase Project [5,6], where the authors proposed a kind of virtual dataflow machine programmed with the use of a set of predefined "operators" (each corresponding to one parallel computing pattern). In this approach, the computation (dataflow) is specified upfront and remains unchanged at runtime and the rate of elements within the flow is constant.

This paper introduces a new execution model for agent-based computing systems[1] based on variable-rate dynamic dataflows. When compared to previous work [5,6,1], the new model has more "degrees of freedom" (e.g. different queuing policies, varying granularity of the data in the flow). As a result, it leaves more space for optimisation both at the programming phase and at runtime. The support for an auto-adaptation is "embedded" in

[1] An implementation of multi-agent evolutionary algorithm is used as an example of such a system.

the model. The proposed model is very general both from the deployment point of view and because of the scope of the computation/simulation tasks it covers. In fact, we demonstrate that this new execution model can be considered as a generalized of the previously mentioned ones.

The main contributions of this paper are:

- a specification of the new, dataflow-based execution model together with the "derivations" of some of its special cases (or projections); for instance, the classic, synchronous, population-based EMAS or the generation-free EMAS,
- the results of (initial) experimental validation of the model, which show that the new approach outperforms the classic EMAS.

The remainder of this paper is organised as follows. Section 2 contains a review of some concurrency-related aspects of agent-based evolutionary computations. Following that, in section 3, the proposed execution model for this class of computations is presented. In section 4, the results of experimental verification of the model are given and discussed. The last two sections (i.e. 5, 6) contain the related work overview and the conclusion of the study.

2 | CONCURRENCY IN AGENT-BASED EVOLUTIONARY COMPUTATIONS

The domain of our work is agent-based evolutionary computations (EMAS) [7], hybrid meta-heuristics which combine elements of multi-agent systems with evolutionary algorithms. They solve an optimisation problem by evolving a population of agents, each owning a *genotype*, which is an encoded candidate solution. The representation of this solution is problem dependent, but will usually be a binary or real-valued vector.

This solution is evaluated to determine the fitness of the agent. The fitness is a number or a vector of numbers in the case of multi-objective optimisation. The optimisation problem consists in finding the candidate solution(s) with the best fitness. Depending on the problem, the best fitness may mean the maximum or the minimal one across all candidate solutions (or the set of vectors such that no other ones are strictly better in all dimensions, in the case of multi-objective optimisation).

In this paper, we assume that the solution represented by an agent stays constant during its lifetime. Fitness evaluation, which is usually an expensive operation, will therefore be executed only once for each agent. Agents execute different actions, communicate among themselves and interact with the environment and should be autonomous and share no global knowledge [8]. Therefore, in contrast to traditional evolutionary algorithms, selective pressure is designed to *emerge* from peer to peer interactions between agents instead of being managed globally.

Such selective pressure is introduced by assigning agents with a piece of non-renewable resource, called energy [7] which drives the behavior of the agent. "Good" behavior is rewarded with additional energy whilst "bad" behavior results in energy being taken away. In this paper, we assume a very simple strategy: "being good" means having better fitness.

So, the rules for managing energy are as follows:

- Agents in the first generation are given some initial energy.
- Agents receive some energy from their parents when created.
- If the energy of an agent is below some threshold, it fights with another agent by comparing their fitness values – the better agent takes energy from the worse one.
- Agents with sufficient energy can reproduce and yield new agents. The genotype of the children is derived from their parents using variation operators.
- When the energy of an agent drops to zero, it is removed from the system.

This strategy introduces a feedback loop in the system which results in emergent selective pressure. In contrast to traditional evolutionary algorithms, the number of agents may vary over time. As energy is discrete and cannot be infinitely subdivided, the maximal amount of agents is bounded by the total energy in the system.

What remains to be defined in such a model is the execution model of agent interactions, i.e., how we choose which agents fight or reproduce with each other and when they do it. This choice may impact the semantics of the algorithm as much as the rules governing energy.

For example, a computation where every agent always interacts with a restricted set of neighbours will yield different results than a computation where every pair of agents can meet – in the first case there is a higher chance that multiple, genetically diverse subpopulations coexist, which is a phenomenon called *allopatric speciation* and can be useful for example in multimodal optimization [9,10].

Another aspect of the model of interactions is how they are related causally during the computation. In other words, how concurrent are the interactions as perceived by the agents. One possible approach is to define a total order on interactions, such that the consequences of every interaction between a pair of agents is immediately visible to all other agents in subsequent interactions. A different way is to have a partial order of interactions, such that multiple interactions are *conceptually* executed in isolation and their consequences are eventually propagated to other agents. Note that the latter example does not imply parallelism. The choice is foremostly about how information propagates in the multi-agent system.

However, if agent interactions can be executed in parallel, the system as a whole can progress faster. As such, different execution models are amenable to various degrees of parallelism, as described in Section 2.2.

2.1 | The traditional approach to concurrency in evolutionary algorithms

Traditionally, evolutionary algorithms have focused more on high-level parallelism rather than on the concurrency of the individuals. For example, the island model consists in splitting a population of individuals into several sub-populations which can be evolved in parallel [11]. However, the algorithm within each island is sequential, and at each step a new population is born out of the previous one as a whole.

There exists several methods of choosing members for the new population. The population size usually remains constant, and new individuals may be chosen from a pool of offspring while disregarding the previous generation in a strategy denoted as (μ, λ) [12]. Offspring can also compete with individuals from the previous population in a strategy denoted $(\mu+\lambda)$. Overall, there is no concurrency at the individual level, as the population is transformed successively through discrete generations (Figure 1 a).

Considering individuals as agents and introducing emergent selection through energy exchange allows for greater flexibility. Generations no longer exist conceptually, as agents of different age may meet as long as they manage to keep their energy and live long enough.

In practice, however, initial implementations of agent-based evolutionary algorithms were still discrete-step simulations. In every step, agents would be randomly paired with each other in order to interact. New agents may be created as a result of such interactions, and these agents were added to the population in the next step. There were no generations at the individual level, but the population was still transformed through successive discrete steps. In contrast with a $(\mu + \lambda)$ strategy, the transformation function was not explicitly defined but emergent instead, and the population size could vary (Figure 1 b) There was still no concurrency at the agent level, only non-determinism through random shuffling.

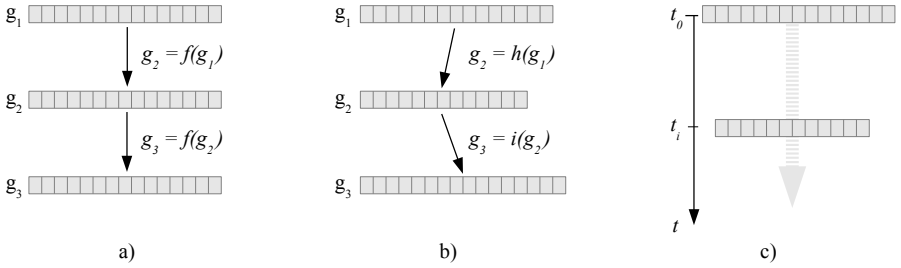


FIGURE 1 Different approaches to concurrency in evolutionary algorithms. Traditional algorithms usually have no concurrency and the population is transformed successively through discrete generations using a globally controlled selection mechanism (a). Initial EMAS implementations allow the use of an emergent selection mechanism which may be different at each step and allow the population size to vary, but there are still discrete generations (b). As described in Section 2.2, meeting arenas allow the introduction of actual concurrency resulting in the cohabitation of individuals of multiple generations, which are no longer discrete but evolve continuously [2] (c).

2.2 | Meeting arenas as a way to abstract the concurrency of agents interactions

In [2] we have proposed a design which allows to define a multi-agent computation such as EMAS in a way which is portable across different models of agent interactions. As such, it becomes possible to meaningfully compare different such execution models, both in terms of performance and scalability, as in how they may affect the semantics of a particular algorithm.

We use the metaphor of *meeting arenas* to convey the intuition of this pattern. Based on their state, agents select an action they are willing to perform, such as fighting, reproduction, etc. Then, agents conceptually move to an arena where they can meet other agents willing to perform the same action. In other words, meeting arenas allow to split a flow of incoming agents into groups of coherent behavior. Each kind of agent behavior is represented by a separate arena. Depending on the type of the behavior, agents are grouped together within arenas and interactions can proceed (see Figure 2).

Therefore, the semantics of the MAS algorithm are fully determined by two functions. The first one consists in *agent behavior*, which chooses the arena to meet on based on the state of an agent. The second corresponds to the *meeting operation* which is computed at every arena for groups of agents. Listing 1 illustrates such functions for the simple EMAS described above.

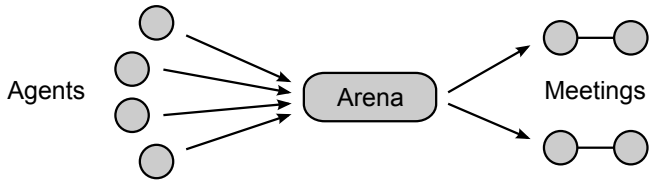


FIGURE 2 The *meeting arenas* pattern [2] allows to decouple the semantics of a multi-agent simulation from its execution model, and explore different models of varying concurrency. It consists of dividing agents into groups of similar behavior and performing meetings between them to yield new or modified agents. As such, it is similar to the *MapReduce* programming model.

```
1  sealed trait Behavior
2  case object Death extends Behavior
3  case object Reproduction extends Behavior
4  case object Fight extends Behavior
5
6  trait GeneticBehavior {
7    def shouldReproduce(energy: Int) = energy >= reproductionThreshold
8
9    def behaviorFunction(agent: Agent) = agent.energy match {
10     case 0 => Death
11     case x if shouldReproduce(x) => Reproduction
12     case _ => Fight
13   }
14 }
15
16 trait GeneticMeetings {
17
18   def fightStrategy: Seq[Agent] => Seq[Agent] = ???
19   def reproductionStrategy: Seq[Agent] => Seq[Agent] = ???
20
21   def meetingsFunction(behavior: Behavior, agents: Seq[Agent]) = behavior match {
22     case Death =>
23       Seq.empty
24     case Fight =>
25       agents.grouped(2).flatMap(fightStrategy).toSeq
26     case Reproduction =>
27       agents.grouped(2).flatMap(reproductionStrategy).toSeq
28     }
29 }
```

Listing 1 The behavior function chooses the behavior of an agent based on its current state. Agents exhibiting the same behavior will be grouped together so that they can interact as defined by the meetings function. For a given behavior and a group of agents exhibiting that behavior, the meeting functions yields a new group of agents. The output group may contain new agents (which are added to the system) or skip some input agents (which are then removed from the system).

This approach resembles the *MapReduce* programming model [13]. The agent behavior partitions the agents population into meeting arenas just as in the *mapping phase*. The meeting logic transforms the population and aggregates it back as in the *reduce phase*.

What is left to define is how and when these functions are combined and that constitutes the execution model of agents interactions. A simple such a model is shown in Listing 2, where a population is iteratively transformed by repeated application of the behavior and meetings functions.

Note that there is no parallelism in this example, yet agent interactions are *conceptually* concurrent to some extent, in the sense that all meetings are computed independently and their consequences are only visible in the next iteration.

```
1  class SequentialModel(behaviorFunction: BehaviorFunction, meetingFunction: MeetingFunction) {
2
3    def run(steps: Int, initialPopulation: Seq[Agent]): Seq[Agent] = {
4      Stream.iterate(initialPopulation)(runStep).apply(steps)
5    }
6
7    def runStep(population: Seq[Agent]]): Seq[Agent] = {
8      population
9        .shuffled
10       .groupBy(behaviorFunction)
11       .flatMap(meetingFunction)
12       .toSeq
13   }
14 }
```

Listing 2 In a sequential execution model, the population is transformed step by step by first shuffling the agents, then grouping them according to their behavior, then computing the meetings function on each group of similar behavior, and finally combining the results.

Several more complex execution models have been explored so far, each with different limitations.

An execution model based on the composition of parallel skeletons has been described in[14]. The *Farm* skeleton was used to partition the population according to agent behavior and processed each partition in parallel as a meeting arena. The model proved linearly scalable, but was semantically equivalent to the sequential one: discrete generations of agents were implied by the iterative transformation of a population (Figure 1 b). From the point of view of the concurrency of agent interactions, it is as if there was a synchronization barrier forcing agents to wait for each other after every interaction. This was a consequence of the design of the underlying skeletons which were based on fixed-rate stream processing.

In turn, an actor-based concurrent execution model has been introduced in[1], where both agents and meeting arenas were modelled as actors communicating through message passing. As such, agent interactions were truly concurrent and there is no implicit generational synchronization – instead, the population was changing continuously as agents interacted with each other (Figure 1 c). When applied to an optimization problem, this execution model proved significantly more efficient in terms of the number of fitness function evaluations needed to reach a solution, demonstrating that the choice of the execution model can impact the semantics of the multi-agent computation. However, the concurrent properties of the algorithm could not be easily reasoned about, as they depended on scheduling internals of the underlying actor dispatcher. Moreover, that model showed to be less scalable on larger numbers of cores as the context switching of actors by the dispatcher incurred a performance overhead. Finally, stream-like patterns emerged, as data appeared to be repeatedly sent between the same actors. This suggested that such a fine-grained level of concurrency might not be needed to preserve the interesting properties of the model.

3 | NEW EXECUTION MODEL FOR AGENT-BASED EVOLUTIONARY COMPUTATIONS

In this paper, we capitalize on previous research and introduce a data-flow execution model which combines the advantages of skeleton and actor-based execution models. This model also makes it precisely control the ordering of agent interactions and use computing resources more efficiently. It is also shown to be a generalization of the previous approaches, as these can be simulated as special cases of the new execution model.

This new approach is based on variable-rate streaming based on actors. The building blocks described below have been implemented using the *Akka Streams*[2] library, which allows to program resource-bounded, non-blocking, back-pressured, asynchronous data streams compliant with the *Reactive Streams*[3] standard.

[2]https://akka.io/docs/
[3]https://www.reactivemanifesto.org/

In Akka Streams, a stream is described as a graph of connected stages. Some stages are *Sources* which emit elements, some are *Sinks* which consume elements, possibly with a side-effect. Stages with one input and one output port are called *Flows*. The simplest graph is a Source connected to a Sink, possibly through several flows. However, stages can have any number of input and output ports, allowing to define arbitrary computing graphs.

The rate of elements may be different between stages. In particular, stages can emit zero, one or more output elements for every input element. Akka Streams use pull-based back-pressure to adapt publishers offer to subscribers demand. In other words, elements will be emitted as fast as possible, as long as there is pending demand. This allows to dynamically switch between a pull and a push approach to maximise throughput while guaranteeing boundedness. Stages are synchronous by default and asynchronous boundaries can be explicitly defined by the user. As much as possible, Akka Streams combines the execution of stages within asynchronous boundaries to minimise thread-switching overhead.

An Akka Streams graph is defined declaratively and can be *materialized* into a set of actors which will start exchanging data along the edges of the graph. Materialization produces a value, which is usually an asynchronous result which will be available once the stream is completed or shut down (see Listing 3).

```
1 val wordsSource: Source[String, NotUsed] = Source(List("some", "words", "to", "count"))
2 val coutingFlow: Flow[String, Int, NotUsed] = Flow[String].fold(0) { case (i, _) => i + 1 }
3 val sink: Sink[Int, Future[Int]] = Sink.last[Int]
4
5 val graph: RunnableGraph[Future[Int]] = wourdsSource.via(coutingFlow).toMat(sink)(Keep.right)
6 val futureSum: Future[Int] = graph.run
7 futureSum.onSuccess(sum => println(sum))
```

Listing 3 The basic building blocks in Akka Stream are Sources, Flows and Sinks. They can be composed declaratively into graphs which are an immutable description of a stream. Graphs with no loose input or output edges can be run, instantiating an actor-based stream and materializing some value. In this example, we have a finite source which will emit a sequence of words, a flow which will count the number of inputs and emit a sum, and a sink which will materialize the last (and unique) value it eventually receive. Materializing this particular graph yields a future which will be completed with the count of words once the stream is done consuming the source. Note the variable rate in this example: the countingFlow will only eventually emit a single element once it has consumed all input elements.

The building blocks described in the remaining of this section are also assembled into such a graph. When the graph is run, it starts the computation and materialize a future value which will eventually be completed with the results of the computation (see Listing 4). In practice, the computation will also side-effect by emitting intermediate logs and statistics to disk.

3.1 | Looping graph

In order to iterate an evolutionary algorithm within a stream, we need to introduce a feedback loop in the stream so that output elements may be inserted back.

Reactive streams do not need to be linear, but can form arbitrary graphs. Some stages may have multiple inputs or outputs and can be connected to form cycles. Care must be taken to ensure both liveliness and boundedness in such streams. In our case the boundedness is guaranteed because of the finite energy in the computation, which dictates a finite amount of agents.

Figure 3 shows the graph used to iterate the computation. An infinite sink continually requests element from the graph. Every element consumed by the sink is also broadcast into an explicit *buffer*. The request from the broadcast propagates through the *step* stage to a *concatenation* stage which will first consume elements from the initial source before switching to the feedback buffer. For every input element, the step stage can emit zero, one or many output elements.

Therefore, the initial population will be drained from the initial source, transformed through the *step* stage into a new population and pushed in the buffer. Then, elements from the buffer will be continually pulled, transformed and pushed again. The size of the buffer is configured to be bounded by the maximum possible number of agents, as determined by the total energy in the computation.

3.2 | Discrete Generations

Synchronous Step

The looping graph can be used to execute a traditional, synchronous agent-based evolutionary computation (classic EMAS). In such a scenario, there will only be one element looping through the graph – the whole population as a collection of agents.

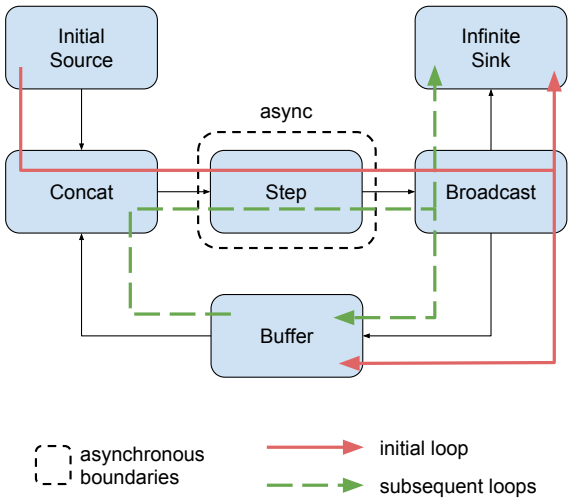


FIGURE 3 Looping graph. The stream used to iterate the computation in a loop. The infinite sink just discards elements. The broadcast stage sends every input element into the infinite sink and into a buffer. The buffer size must be configured to be able to keep the maximum amount of agents, as determined by the total energy in the system. The initial source is first drained through the step stage into the buffer, which is then continuously pulled through the step stage back into itself. Different step stages will be explored further below.

```
1  val source: Source[Agent, NotUsed] = Source(initialPopulation)
2  val step: Flow[Agent, Agent, NotUsed] = ???
3  val sink: Sink[Agent, Future[Option[Agent]]] = Sink.fold(Option.empty[Agent]) {
4    case (None, agent) => Some(agent)
5    case (Some(bestSoFar), agent) => Some(max(bestSoFar, agent))
6  }
7
8  def loopingGraph[A, Mat](
9    source: Source[A, _],
10   step: Flow[A, A, _],
11   sink: Sink[A, Mat],
12   bufferSize: Int
13 ): RunnableGraph[(Mat, KillSwitch)] = ???
14
15 val (best, killSwitch): (Future[Agent], KillSwitch) = loopingGraph(source, step, sink).run
16 scheduleOnce(someDuration) {
17   killSwitch.shutdown()
18 }
19 best.onComplete(println)
```

Listing 4 A simplified example of how a description of the computation is constructed by specifying the initial source, the intermediate step and the final sink. Materializing the resulting graph starts an infinite stream and returns two values: a Future corresponding to the best solution found by the end of the computation, and a switch to externally terminate the stream.

The step flow synchronously transforms one population into a new one, as shown in Listing 5. The population is first shuffled, then partitioned according to agent behavior. Finally, every partition is transformed using the meetings function.

```
1 def step = Flow[Seq[Agent]].map { population =>
2   population
3     .shuffled
4     .groupBy(behaviorFunction)
5     .flatMap(meetingsFunction)
6 }
```

Listing 5 A simple synchronous step flow, where a population of agents is transformed into a new one by applying the behavior and meeting functions (as in the classic EMAS).

Asynchronous Step

The step flow can be parallelised while maintaining discrete generations. The required design can be decomposed in two parts as follows.

First, we use a flow to transform a stream of individual agents, as shown in Figure 4 . The flow of input agents is partitioned into sub-streams according to the behavior function.

On each sub-stream, agents are then grouped within a time and size windows. A size window of n means that we wait for n input elements $a_1, \ldots, a_n$ to arrive before emitting a single element output element consisting of $(a_1, \ldots, a_n)$. A time window of t means that if only $m < n$ input elements have been received so far and t time have passed since the last one was received, we emit $(a_1, \ldots, a_m)$ anyway.

Then, every emitted group is transformed in parallel using the meetings function. In order to satisfy the boundedness property, the number of parallel meetings for each sub-flow is bounded by a parallelism parameter.

The meetings corresponding to each behavior may modify agents, split them into more agents or remove existing ones from the flow. The agents resulting from all the meetings are eventually propagated into the output.

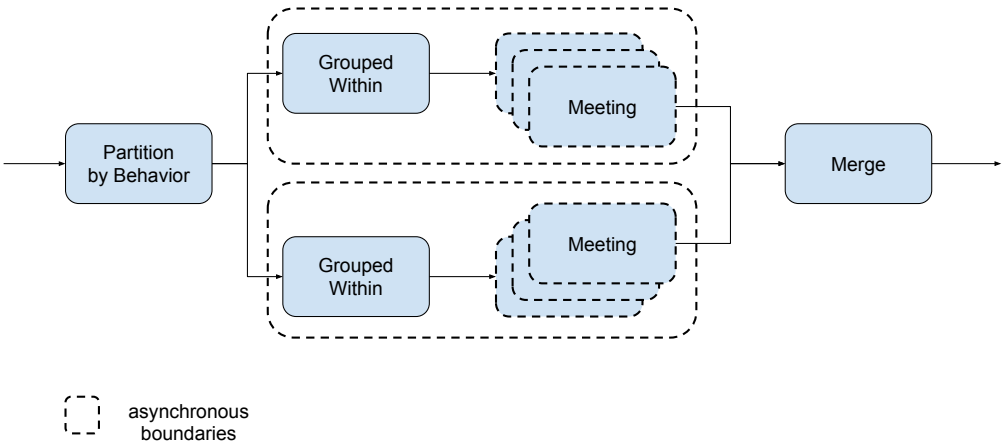


FIGURE 4 Arenas Flow: Modeling meeting arenas as a flow in the stream. Agents are partitioned into sub-flows according to their behavior. In each sub-flow, agents are grouped within a time or size window specific to the behavior. Every group then performs a meeting, which may yield some output agents. The number of output agents from a meeting may be different from the number of input agents. Agents resulting from all meetings are then merged into the output of the flow.

The second step in the design is lifting such a flow of agents into a flow of population which we could insert into a discrete generational loop. The solution is illustrated in Figure 5 . After every input element, we start a *sub-stream*, which allow to define transformation of elements in terms of streams. In the sub-stream, we transform the input population into a stream of constituent agents, forward them through the meeting arenas flow and accumulate the results using a fold stage.

The fold stage will emit a single element when its upstream is completed, i.e. all original agents have passed through the arenas flow. This output element will simply be the new population. For every input population, we will thus start a sub-stream which will produce a single output population, and we concatenating the outputs of such subsequent sub-streams into our output.

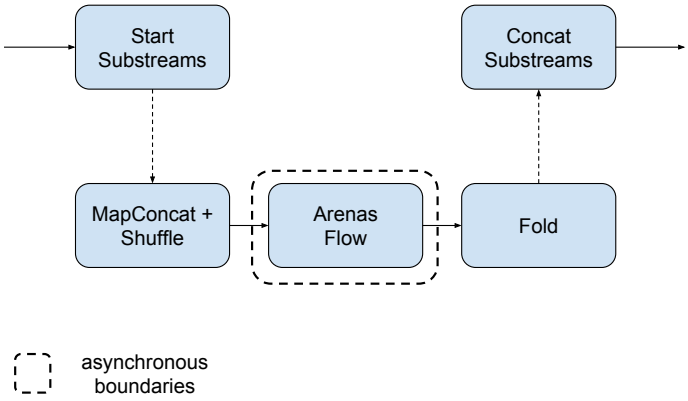


FIGURE 5 Sequential Flow. For every input population element, we start a new substream, where we decompose the population into a stream of agents, transform that stream through the arenas flow, then combine the results into a new population.

We need to use sub-streams so as to know when the stream of agents from a given population is over, so that the fold stage may emit the new population. As we will see in Section 3.4, the same semantics of a generational synchronisation can also be achieved in a different and more general way.

3.3 | Continuous generations

The solution above is similar to the *farm* pattern, analysed in previous work on parallel skeleton patterns [15], where an input element is partitioned and each partition is transformed in parallel before being recombined.

However, the main difference compared with previous work is that the use of reactive streams allows us to change the rate of elements and consider the partitioning of a stream of agents, instead of the partitioning of a collection of agents.

Therefore, the next approach consists in dropping the concept of explicit populations and consider only a stream of agents in the looping graph. Generations are no longer explicitly synchronized, agents are transformed continuously and agents from different generations can meet freely.

One problem with a purely streaming solution is that it is mostly deterministic. In other words, it is similar to a First In First Out queue, from which we would pop some agents from the beginning, transform them and push the resulting ones in the end. In most cases, agents would keep interacting with the same "neighbours". In general, stronger stochastic properties are recommended in evolutionary algorithms to ensure exploratory characteristics [16].

Introducing stochasticity in a stream is not trivial, as we no longer have the whole population under hand to be able to shuffle it. The looping graph may contain a buffer, but this buffer will probably never contain the whole population at any given moment, as some agents may be in internal buffers thorough intermediate stages.

The solution used in this paper is based on a technique called "Reservoir Sampling" [17]. Reservoir sampling is an algorithm for selecting random or representative elements from a stream of unknown size, possibly infinite. The algorithm maintains an internal pool of chosen elements. Each new stream element replaces one in the internal pool with some probability, chosen so that when the stream is finished or interrupted, all elements so far have had the same probability of ending up in the final pool.

We use a similar technique to introduce stochasticity in the stream of agents. As illustrated in Figure 6 , we introduce a *shuffling buffer* stage in the step flow, ahead of the meeting arenas. The shuffling buffer maintains an internal pool of input elements. When pulled, it chooses the output element based on some policy, effectively changing the order of elements in the stream. This buffer is distinct from the one in the looping graph, as it has different purpose, semantics and capacity.

The use of a shuffling buffer abstraction introduces a powerful degree of freedom to the algorithm, as many different strategies can be considered. In fact, as we will see below, all previous models of execution can be considered special cases of such a shuffling buffer.

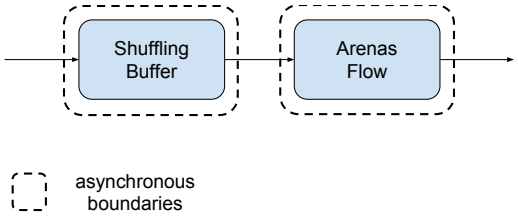


FIGURE 6 Continuous Flow. Agents flow through a shuffling buffer, which potentially outputs them in a different order that they arrived, e.g. by using reservoir sampling with an internal pool. Then, agents continue through the Arena flow described in Figure 4 .

3.4 | Shuffling Buffer

The following section explores several shuffling buffers strategies implemented and benchmarked in this paper.

Random Buffer

The simplest shuffling strategy is a buffer of fixed capacity, which chooses a random element when pulled. Therefore, elements will leave in a different order that they entered, so the stream will effectively be shuffled.

It is not a true permutation as we would have to consume the whole stream to have a non-zero probability of seeing any potential input element as the first output one.

However, elements have a non-zero probability of staying indefinitely long in the buffer. The bigger the size of the buffer, the longer elements will stay in average, and the more shuffled the stream will seem. However, too big buffer can immobilize a significant part of the population and decrease overall throughput.

Max Buffer

This strategy is similar to the previous one in that we immobilize a part of the stream in an internal buffer. However, the element chosen as the output is always the one which maximizes a given metric, in this case the fitness value associated with the agent.

Annealed Buffer

This strategy behaves like a random buffer with probability p, and like a max buffer with probability $1 - p$. The value of p decreases every second according to the formula $p_i = p_{i-1} \cdot 2^{\frac{-1}{n}}$. In other words, p will be halved every n seconds, called the half-life time of the buffer. A max heap with fixed time top and random removal is used as the backing data structure of this buffer.

The random buffer strategy is actually a special case of an annealed buffer strategy with infinite half-life time, while the max buffer a special case with $\lim \to 0$ half-life time.

Barrier Buffer

The goal of that buffer is to simulate a generational evolutionary algorithm by introducing an explicit synchronisation barrier. The buffer is configured with the total energy injected in the multi-agent system.

This buffer alternates between two states: open and closed. When closed, it accepts incoming agents but does not emit any in the output. When the total energy of the agents in the buffer equals the expected total energy in the system, the buffer opens. When open, it does not accept any incoming agents, but emits agents in the output until it become empty, when it closes again.

Therefore, agents from different generations are never mixed together. This buffer demonstrates that a continuous flow of agents is a more general model, as the discrete generations are just a special case.

The use of this buffer allows us to compare the characteristics specific to the semantics of discrete generations, while controlling for the overhead specific to the overall stream-based architecture.

4 | EXPERIMENTAL RESULTS

In this section we describe the optimisation problems used to test our model and we present numerical results of these tests. The main purpose of the experiments was to evaluate the impact of the new execution model and its parameters on the semantics of the algorithm and also to validate that the new execution model can simulate previous ones.

4.1 | Optimisation Problem

As metaheuristics often have multiple parameters and include many operators, it is difficult to separate the effect of their different constituents on the observed properties of the algorithm. In our case, we want to distinguish the properties of the different shuffling buffers and their impact on the performance of the optimisation process. In order to minimise the effect of the choice of the evolutionary operators, we chose two optimisation problems which have been well studied and for which good operators and their parameters are well known. As such, we applied our multi-agent algorithm to the following two test problems: searching for the minimal values of the Rastrigin function (Eq. 1) and of the Ackley function (Eq. 2) – two common benchmarking functions used to compare evolutionary algorithms [18].

$$f(\mathbf{x}) = 10n + \sum_{i=1}^{n}(x_i^2 - 10\cos(2\pi x_i)) \tag{1}$$

$$f(\mathbf{x}) = -20\,exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n}x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n}cos(2\pi x_i)\right) + 20 + e \tag{2}$$

These functions are highly multimodal with one global minimum equal to 0 at $\bar{x} = 0$ (Fig. 7)[4]. In the computational experiments we used a problem

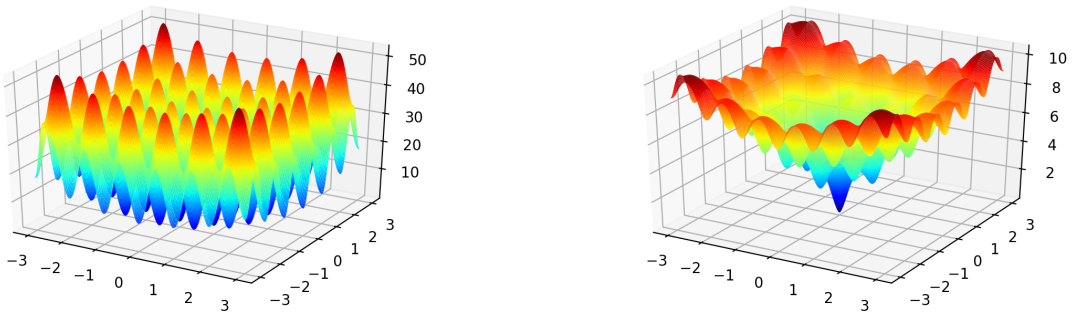


FIGURE 7 Test functions (for $n = 2$): Rastrigin's function (on the left) and Ackley's function (on the right).

size $n = 100$ in a domain being the hypercube $[-50, 50]^{100}$ for the Rastrigin function, and the hypercube $[-33, 33]^{100}$ for the Ackley function. These are the typical domains used when using these test functions.

4.2 | Methodology

The computations were run on AMD Opteron 6276 nodes using 12 cores and 2 GB of memory. The duration of each experiment was set to 3 hours, and each one was repeated 30 times in order to obtain statistically significant results. The results below are averaged over these 30 runs. Unless otherwise specified, the graphs show the change of these average values over time together with 95% confidence intervals.

[4] One particular property of Rastrigin's function is that values lower than 1 correspond to the basin of attraction of the global optimum.

Metrics

The models were evaluated with respect to two criteria: the efficiency and effectiveness of the optimisation. We measured the effectiveness of the algorithm by recording the best fitness found so far at any given time. In addition, we measured the efficiency by recording the number of fitness evaluations performed so far at any given time. We combined those metrics to compute the best fitness found after a given number of evaluations.

The total number of evaluations that are performed vary over different runs. In order to compare meaningful averages, the plots of fitness over evaluation count are truncated at the smallest number reported at the end across all runs of a given configuration.

A constant equal to 10^{-16} has been added to all fitness values plotted below in order to visualise the global optimum on a logarithmic scale. In other words, a fitness value equal to 10^{-16} means the optimisation did reach the global optimum.

Configurations

We benchmarked two buffer strategies among those described in Section 3.4: "annealed buffer" and "barrier buffer". The other variants can be considered as special cases of the first one.

The "barrier buffer" corresponds to a sequential evolutionary algorithm with discrete generations (i.e. the classic EMAS). This strategy has no relevant parameters.

The "annealed buffer" is a concurrent strategy with continuous generations. It has a p probability of selecting a random individual as the next output and a $1 - p$ probability of selecting the best one. p is initially equal to 1 and decreases exponentially as described below.

An annealed buffer is configured with the following two parameters:

buffer size – the size of the shuffling buffer,

half-life time ($t_{\frac{1}{2}}$) – the time in seconds after which the probability p is halved. At the extremes, this buffer behaves like a *random buffer* when $\lim p \to \inf$, and a *max buffer* when $\lim p \to 0$

We have run the annealed buffer with the following combinations of parameters:

- three sizes of the buffer: 10, 40, 100;
- four values of the half-life time parameter $t_{\frac{1}{2}}$: 60, 900, 3600 seconds and one version with deactivated decay (marked as *inf* in figures and tables). The last configuration is semantically equivalent to a *random buffer*.

4.3 | Results and Discussion

Figures 8 and 9 show a summary of all possible configurations with the annealed buffer, while figure 10 shows results for the barrier buffer.

The results for the annealed buffer are detailed in the appendix. For the Rastrigin function, Figures A1 to A7 present the results along with 95% confidence intervals organised along the buffer size dimension (Figures A1 to A3) and along the half-life dimension (Figures A4 to A7). In a similar way, for the Ackley function, Figures A10 to A16 present the results along with 95% confidence intervals, organised along the buffer size dimension (Figures A10 to A12) and along the half-life dimension (Figures A13 to A16).

The total number of evaluations performed by the end of the computation differs across runs and configurations and is summarized in Tables 1 (for Rastrigin function) and 2 (for Ackley function). Median, maximum value and quartiles are represented as relative to the mean.

Mean time series are only meaningful over the ranges of evaluation numbers for which we have data from all the runs of a given configuration. Therefore, the "best" and "worst" runs of the annealed buffer are represented in Figures A8 to A9 (for the Rastrigin function) and Figures A17 to A18 (for the Ackley function).

The best and worst runs are defined as follows:

- when all runs reached zero, the best run is the one which reached it the first, and conversely for the worst run.
- when some run reached zero, the best is defined as above. The worst run is the one which had the worst (largest) fitness value at the end of the computation.
- when no run reached zero, the worst is defined as above. The best run is the one which had the best (lowest) fitness value at the end of the computation.

The first conclusions we can draw from Figures 8 and 10 is that the barrier buffer performs the worst by reaching average final values of the order of 10^{-6} for the Rastrigin function. It is followed by the annealed buffer configured with no actual decay (which corresponds to a random buffer) which reaches average final values of the order of 10^{-10}. Similarly, for the Ackley function, these values are close to 10^{-4} and 10^{-5}.

For the Rastrigin function, other configurations of the annealing buffer perform much better, reaching the global optimum (represented as 10^{-16}) for almost half of the configurations, and reaching 10^{-12} in the worst ones. We can see in Figures A1 to A7 that the difference between the

annealed buffer and both the barrier and random one is statistically significant. In the case of the Ackley function algorithm does not reach global minimum but it still is significantly better than the barrier buffer version.

Interestingly, the annealed buffer performs better the longer the half-life time. As we can see in Figures A1 to A3 , this difference becomes more significant the bigger the size of the buffer.

Our interpretation is that it results from the computational complexity of the annealed buffer implementation, which is a max heap. The amortised cost of removing a random element is smaller than the cost of removing the maximum element. At higher half-life times, we end up removing random elements relatively more often, which increases overall throughput. This explanation is further supported by the evidence that the difference in throughput is more pronounced for bigger buffer sizes (Figures A4 to A6).

buf	$t_{\frac{1}{2}}$	$\frac{\text{Min}}{\text{Mean}}$	$\frac{\text{1stQu.}}{\text{Mean}}$	$\frac{\text{Median}}{\text{Mean}}$	*Mean*	$\frac{\text{3rdQu.}}{\text{Mean}}$	$\frac{\text{Max}}{\text{Mean}}$
10	60	0.8498	0.9665	1.0029	98393543.8	1.0339	1.1351
10	900	0.8739	0.9521	0.9832	101290512.3	1.0333	1.1458
10	3600	0.7301	0.9658	1.0047	113997906.9	1.0523	1.1657
10	inf	0.4536	0.9020	0.9857	166991465.1	1.0871	1.3819
40	60	0.8937	0.9671	0.9918	61422435.7	1.0358	1.1257
40	900	0.9110	0.9509	0.9883	65373658.8	1.0485	1.1119
40	3600	0.8313	0.9419	1.0051	83298855.1	1.0380	1.1510
40	inf	0.8450	0.9582	1.0063	156417948.2	1.0447	1.2062
100	60	0.8688	0.9718	1.0128	52452528.5	1.0399	1.0879
100	900	0.8679	0.9552	0.9973	56551136.2	1.0610	1.1520
100	3600	0.8884	0.9719	1.0026	73827134.5	1.0441	1.1042
100	inf	0.7747	0.9370	1.0185	150022933.5	1.0406	1.2708
barrier (classic EMAS)		0.9683	0.9759	0.9977	65354200.0	1.0217	1.0440

TABLE 1 Statistics about the total number of fitness function evaluations by the end of the computation for each configuration for the **Rastrigin function**. The mean is shown as an absolute value and all other statistics – as relative to the mean. The first two columns represents the parameters of a given configuration: buffer size and $t_{\frac{1}{2}}$. The last row shows results for the barrier buffer.

buf	$t_{\frac{1}{2}}$	$\frac{\text{Min}}{\text{Mean}}$	$\frac{\text{1stQu.}}{\text{Mean}}$	$\frac{\text{Median}}{\text{Mean}}$	*Mean*	$\frac{\text{3rdQu.}}{\text{Mean}}$	$\frac{\text{Max}}{\text{Mean}}$
10	60	0.7237	0.9811	1.0259	80342620.4	1.0657	1.1382
10	900	0.7930	0.9630	0.9958	87852715.5	1.0539	1.1649
10	3600	0.7060	0.9274	1.0401	90264469.0	1.1170	1.2070
10	inf	0.5760	1.0233	1.0508	133273703.7	1.0857	1.2441
40	60	0.9395	0.9706	0.9985	51211047.4	1.0205	1.1265
40	900	0.8250	0.9450	1.0128	54880279.1	1.0465	1.1903
40	3600	0.7600	0.9645	1.0072	70134972.1	1.0434	1.1555
40	inf	0.5395	0.9927	1.0666	130541931.0	1.1053	1.2026
100	60	0.9290	0.9699	0.9938	43627533.0	1.0430	1.0796
100	900	0.7947	0.9315	1.0186	46583632.7	1.0843	1.1891
100	3600	0.7471	0.9064	1.0509	60920639.6	1.0696	1.2396
100	inf	0.5205	0.9415	1.0578	132090258.7	1.1231	1.2440
barrier (classic EMAS)		0.9684	0.9870	1.0007	62029013.0	1.0132	1.0234

TABLE 2 Statistics about the total number of fitness function evaluations by the end of the computation for each configuration the **Ackley function**. The mean is shown as an absolute value and all other statistics – as relative to the mean. The first two columns represents the parameters of a given configuration: buffer size and $t_{\frac{1}{2}}$. The last row shows results for the barrier buffer.

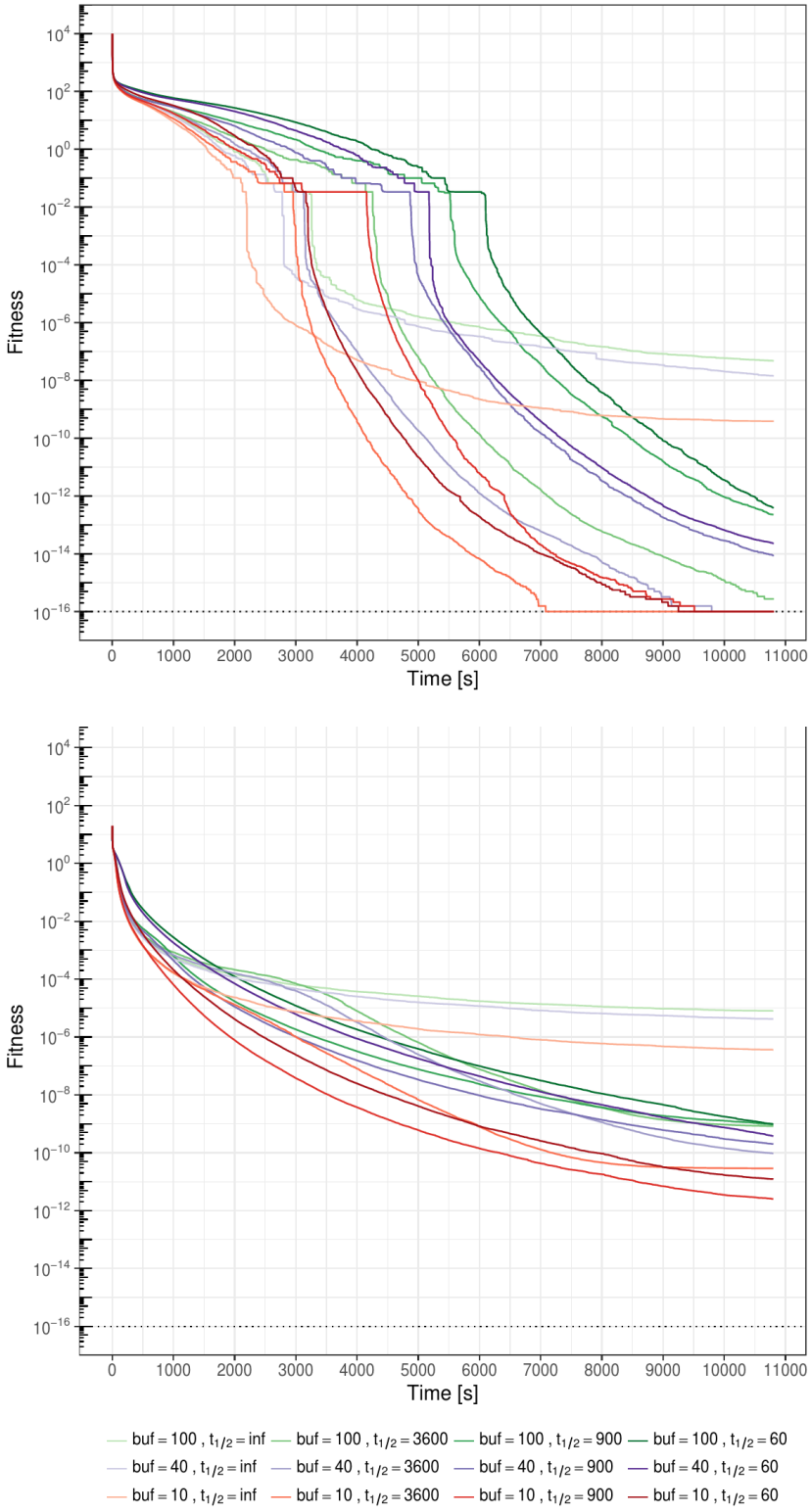


FIGURE 8 Best fitness as a function of time for the **annealed buffer**. The upper chart shows results for Rastrigin function and the bottom one – for Ackley function. The **buf** parameter is the buffer size, $t_{1/2}$ the half-life value in seconds. A 10^{-16} constant has been added to results to visualize the global optimum. Confidence intervals are omitted for clarity.

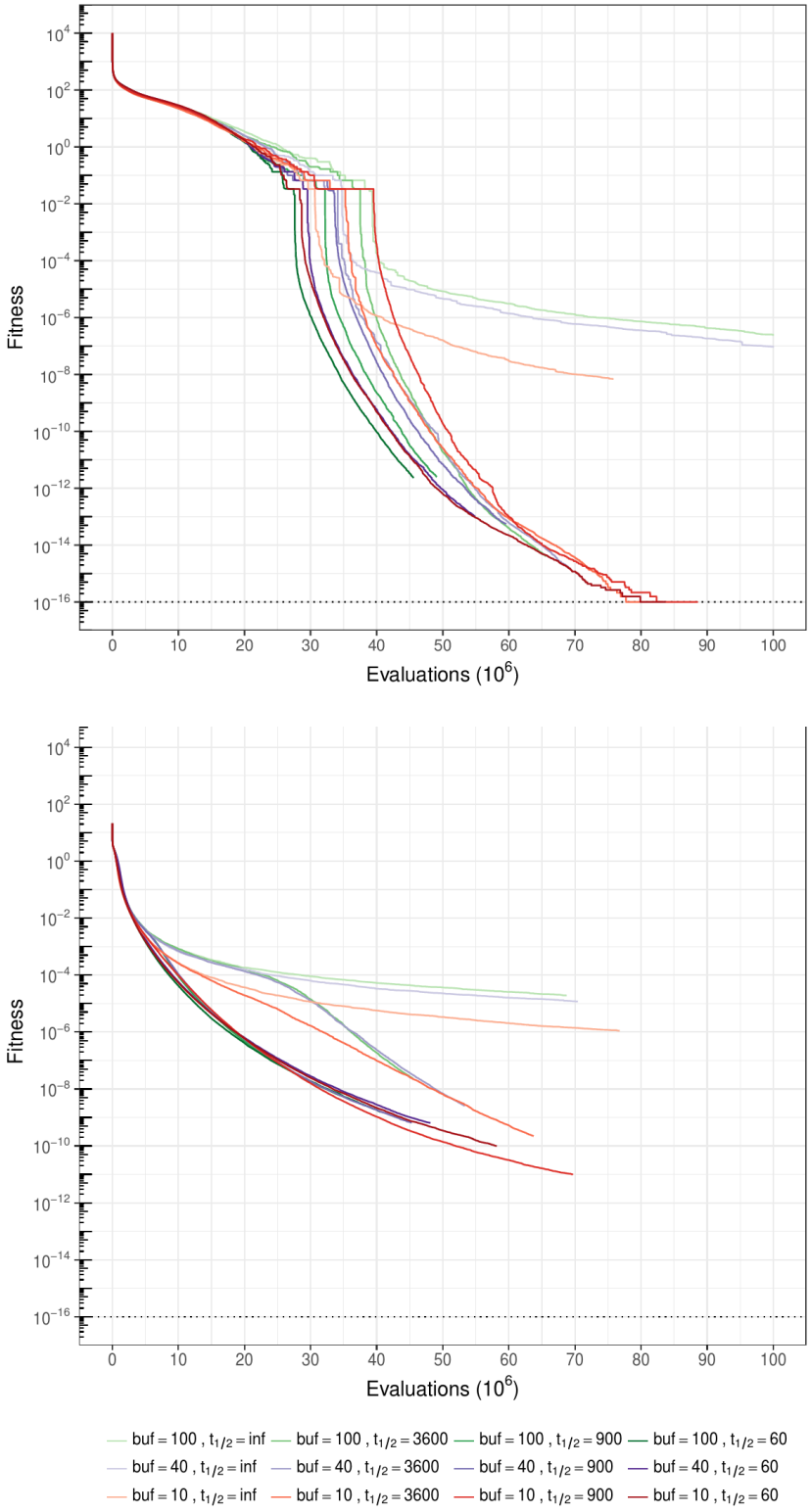


FIGURE 9 Best fitness as a function of the number of evaluations for the **annealed buffer**. The upper chart shows results for Rastrigin function and the bottom one – for Ackley function. The **buf** parameter is the buffer size, $t_{1/2}$ the half-life value in seconds. A 10^{-16} constant has been added to results to visualize the global optimum. Confidence intervals are omitted for clarity.

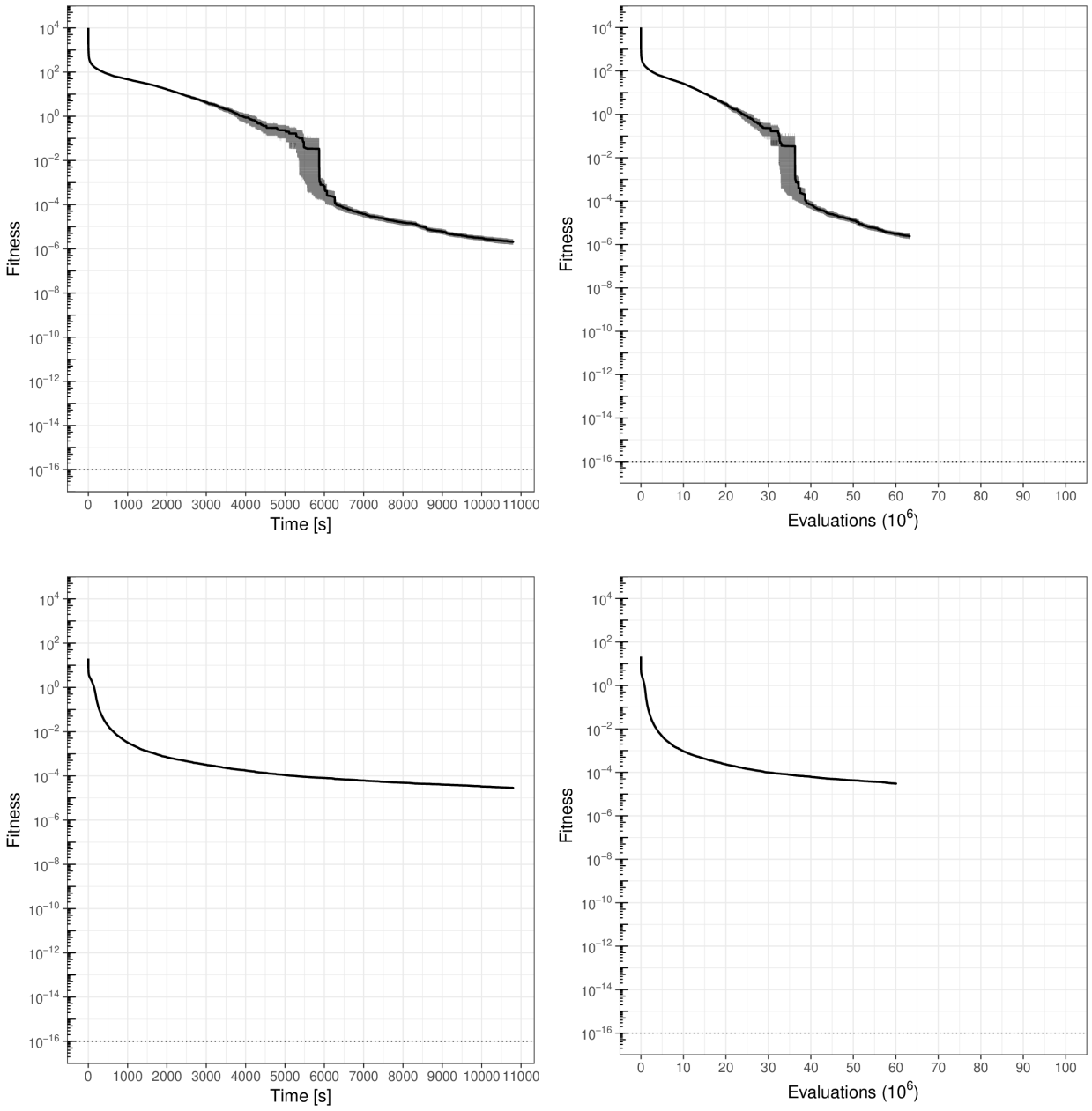


FIGURE 10 Best fitness as a function of time (on the left) and of evaluation number (on the right) in the **barrier buffer** (classic EMAS). Upper charts shows results for Rastrigin function and bottom ones – for Ackley function. The black line represents the mean value and the lighter ribbon a 95% confidence interval.

Tables 1 and 2 shows statistics about the final number of fitness evaluations performed on average by the end of each computation respectively for Rastrigin and Ackley functions. We can notice that in all cases the longer the half-time $t_{\frac{1}{2}}$, the larger the mean number of evaluations. The total number of evaluations also decreases along with growing buffer size. In general, configurations with fully random buffer behaviour ($t_{\frac{1}{2}} = \inf$) were able to perform more evaluations and had larger spread of evaluations count. These numbers are therefore coherent with our conclusions above about how parameter values impact raw performance.

Beyond throughput, we can draw more conclusions by considering the best fitness found after a given number of fitness function evaluations (Figure 9).

The different configurations of the annealed buffer (with the exception of the infinite half-life time) behave very similarly – there is no significant difference as evidenced by Figures A1 to A7 . In turn, the difference between these configurations and the barrier and random one is even more significant when considered that way.

The choice of the size of the buffer turns out to be insignificant with regard to the properties of the algorithm (as evidenced by Figures A4 to A6). However, the value of half-life time can be significant (Figures A1 to A3).

As we can see, the choice of the shuffling buffer, and therefore of the concurrency semantics, turns out to have a significant impact on the characteristics of the overall algorithm. The shuffling strategy turns out to be an important degree of freedom of the stream-based model we propose and its choice is paramount. The results above provide directions for a further exploration of possible strategies.

5 | RELATED WORK

Simulated annealing (SA) is a stochastic search (optimisation) method inspired by the annealing process in metallurgy in which a solid is cooled until its structure is eventually frozen at a minimum energy [19]. In this analogy the (physical) material states correspond to points (candidate solutions) in the search space, the energy of a state – to the cost ("goodness") of a point (solution), and the temperature represents a control parameter. Several methods to improve the basic SA algorithm have been proposed, for instance Cauchy annealing [20], simulated reannealing [21], generalized SA [22], and SA with known global value [23].

Classic evolutionary algorithms (such as the simple genetic algorithm [24], evolution strategies [12] etc.) are generally perceived as universal optimisation metaheuristics (cf. theory of Vose [25]). They employ a simple model of evolution consisting of repeating, for subsequent generations, the same process of selecting parents and producing offspring using variation operators. Although this approach has proved effective in many optimisation problems (e.g. [26]) it reveals some limitations when trying to parallelise the base algorithm (especially if it is to be adapted to massively-parallel computers). This is why only simple approaches to the parallelisation of such algorithms have been proposed (e.g. the master-slave model or parallel evolutionary algorithm [27]).

A multi-agent system (MAS) is a loosely coupled network (system) of agents that work together to solve problems that are beyond the individual capabilities (or knowledge) of each agent [28,29,30]. In MAS no global knowledge is available to individual agents [8]; agents should remain autonomous and no central authority should be needed. An overview of multi-agent systems (given from different perspectives) can be found in [31] and [28]. Examples of popular agent-oriented frameworks are Jadex [32], Jade [33] or MadKit [34]).

Evolutionary Multi-Agent Systems (EMAS) [35] may be treated as general-purpose optimisation systems [36,37,38,16] which utilize key ideas from agent-based simulations and evolutionary algorithms. In EMAS a population of agents (in the form of genotypes) evolve to improve their ability to solve a particular problem. Selection in an EMAS is designed so that agents with good behaviour become more likely to reproduce. Many variants of this mechanism have been proposed (an overview can be found in [7]).

EMAS implementations (e.g. the AgE platform[5]) were applied to different optimisation problems (global, multi-criteria, multi-modal, in continuous and discrete spaces) and the results showed superior performance in comparison to classic heuristics (see, for instance, [7,39,40,41,42]).

Generation-free EMAS [2] bring evolutionary algorithms closer to their biological origins by removing generations (imposed by step-based implementation) from the algorithm. This approach introduces more concurrency and asynchronicity to the EMAS model and, as a result, also such concepts as parallel ontogenesis. Agents may initiate actions at any possible time giving a nearly continuous-time simulation.

Massively-concurrent EMAS's are best suited for closed, fine-grained concurrent systems, with a large number of lightweight (and often homogeneous) agents [43,44,2,45]. In such a class of systems, the popular agent development platforms (e.g. Jade, Magentix) may be not suitable as they are limited to several thousands of simultaneous agents on a single computer [43].

Functional multi-agent systems address the issue mentioned above (i.e. an inefficient use of multi-core CPUs in the shared-memory concurrency model). The most mature example of such a system (platform) is eXAT (*erlang eXperimental Agent Tool*) (see [46] and also [47]). eXAT overcomes the basic limitations of popular Java-based solutions as its agents are based on Erlang lightweight processes, and millions of such processes can be created on a single computer. Other Erlang-based solutions are discussed in [48] and [2]. Haskell-based multi-agent systems are presented in [49] and [50]. Examples of a Scala-based implementations are shown in [44] and [1].

Dataflow is a broad term used in several computing related domains including hardware and software architectures (see, for instance, [4,51]), concurrency/execution models (see, for instance, [52,3]) and programming languages (see, for instance, [53,54]). In the *dataflow execution model*, a program is represented by a directed graph that shows explicitly (by directed arcs) data dependencies. The initial motivation for the research into dataflow

[5] https://age.agh.edu.pl

(and parallel data-driven computation in general) was the exploitation of massive parallelism (see, for instance, above mentioned [4,3]). Since then, this model has received much attention. A review of this research field can be found, for instance, in [53].

6 | CONCLUSION

We have introduced a formal description of the execution model for agent-based computing systems as an adaptive dataflow decoupled from the domain-specific semantics of the computation. We have also shown that the execution models studied in previous works (for instance, the classic, synchronous, population-based EMAS or generation-free EMAS) can be unified in this common model. In addition, we have analysed the parameters of the model, such as queuing policies and granularity of the data in the flow. Following that, we have benchmarked several queuing alternatives and demonstrated their effect on the efficiency of the computation. Using the example of a multi-agent evolutionary optimisation problem solver we have shown that the new approach outperforms the classic one. This model we have described is "natural" for functional languages and can be easily mapped into different classes of hardware – from simple, single-core computers to distributed environments.

Future research work could concentrate on:

- comparing (experimentally) the effectiveness of the proposed algorithm/approach to the wider range of alternative ones (for instance, to the other optimisation heuristics or other multi-agent based simulation environments),
- studying the algorithm in other optimisation problems (both discrete and continuous ones),
- studying the scalability of this new model,
- experimenting with more complex dataflows (for instance, the ones implementing the concept of islands and migrations of agents),
- experimenting with some more special configurations (cases) of the model; in fact, the whole parameter space can be checked/searched,
- experimenting with different variants of adaptation (for instance, adaptive queuing policies and/or adaptive dataflow topologies can be considered).

ACKNOWLEDGEMENT

The research presented in this paper was partially supported by the Polish Ministry of Science and Higher Education under the AGH University of Science and Technology grant (statutory project) no. 11.11.230.124 and by PL-Grid infrastructure[6] at the ACC Cyfronet AGH[7]

References

1. Krzywicki Daniel, Turek Wojciech, Byrski Aleksander, Kisiel-Dorohinicki Marek. Massively concurrent agent-based evolutionary computing. *Journal of Computational Science.* 2015;11:153–162.
2. Krzywicki D., Stypka J., Anielski P., et al. Generation-free Agent-based Evolutionary Computing. *Procedia Computer Science.* 2014;29(0):1068 - 1077. 2014 International Conference on Computational Science.
3. Karp Richard M, Miller Rayamond E. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics.* 1966;14(6):1390–1411.
4. Gurd John R., Kirkham Chris C., Watson Ian. The Manchester prototype dataflow computer. *Communications of the ACM.* 1985;28(1):34–52.
5. Hammond K., Aldinucci M., Brown C., et al. The ParaPhrase project: Parallel patterns for adaptive heterogeneous multicore systems. In: 1, vol. 7542: Springer LNCS 2013 (pp. 218-236).
6. Stypka Jan, Anielski Piotr, Mentel Szymon, et al. Parallel patterns for agent-based evolutionary computing. *Computer Science.* 2016;17(1):83.

[6] http://www.plgrid.pl/en
[7] http://www.cyfronet.krakow.pl/en/

7. Byrski Aleksander, Dreżewski Rafał, Siwik Leszek, Kisiel-Dorohinicki Marek. Evolutionary multi-agent systems. *The Knowledge Engineering Review.* 2015;30(2):171–186.
8. Jennings N. R., Sycara K., Wooldridge M.. A Roadmap of Agent Research and Development. *Journal of Autonomous Agents and Multi-Agent Systems.* 1998;1(1):7-38.
9. Mahfoud Samir W. A comparison of parallel and sequential niching methods. In: 1, vol. 136: :143; 1995.
10. Krzywicki Daniel. Niching in evolutionary multi-agent systems. *Computer Science.* 2013;14.
11. Alba Enrique, Tomassini Marco. Parallelism and evolutionary algorithms. *IEEE transactions on evolutionary computation.* 2002;6(5):443–462.
12. Schwefel Hans-Paul, Rudolph Gunter. Contemporary Evolution Strategies. In: :893-907; 1995.
13. Dean Jeffrey, Ghemawat Sanjay. MapReduce: simplified data processing on large clusters. *Communications of the ACM.* 2008;51(1):107–113.
14. Turek Wojciech, Stypka Jan, Krzywicki Daniel, et al. Highly scalable Erlang framework for agent-based metaheuristic computing. *Journal of Computational Science.* 2016;17:234–248.
15. Hammond Kevin, Aldinucci Marco, Brown Christopher, et al. Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In: 1, vol. 7542: :218Springer; 2013.
16. Byrski Aleksander, Schaefer Robert. Formal model for agent-based asynchronous evolutionary computation. In: :78–85IEEE; 2009.
17. Vitter Jeffrey S. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS).* 1985;11(1):37–57.
18. Bäck Thomas, Schwefel Hans-Paul. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation.* 1993;1(1):1–23.
19. Kirkpatrick Scott, Gelatt Jr C Daniel, Vecchi Mario P. Optimization by simulated annealing. In: World Scientific 1987 (pp. 339–348).
20. Szu Harold H, Hartley Ralph L. Nonconvex optimization by fast simulated annealing. *Proceedings of the IEEE.* 1987;75(11):1538–1540.
21. Ingber Lester. Very fast simulated re-annealing. *Mathematical and computer modelling.* 1989;12(8):967–973.
22. Tsallis Constantino, Stariolo Daniel A. Generalized simulated annealing. *Physica A: Statistical Mechanics and its Applications.* 1996;233(1-2):395–406.
23. Locatelli Marco. Convergence and first hitting time of simulated annealing algorithms for continuous global optimization. *Mathematical Methods of Operations Research.* 2001;54(2):171–199.
24. Goldberg D. E.. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley; 1989.
25. Vose M.. *The Simple Genetic Algorithm: Foundations and Theory*. Cambridge, MA, USA: MIT Press; 1998.
26. Dębski Roman, Dreżewski Rafał, Kisiel-Dorohinicki Marek. Maintaining population diversity in evolution strategy for engineering problems. In: Nguyen Ngoc Thanh, Borzemski Leszek, Grzech Adam, Ali Moonis, eds. *New Frontiers in Applied Artificial Intelligence*, 1, vol. 5027: :379–387Springer Berlin Heidelberg; 2008.
27. Cantú-Paz E.. A Survey of Parallel Genetic Algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis.* 1998;10(2):141-171.
28. Stone Peter, Veloso Manuela. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots.* 2000;8(3):345–383.
29. Russell S. J., Norvig P.. *Artificial Intelligence: A Modern Approach*. Prentice Hall; 3rd edition ed.2009.
30. Niazi Muaz, Hussain Amir. Agent-based computing from multi-agent systems to agent-based models: a visual survey. *Scientometrics.* 2011;89(2):479.
31. Wooldridge M.J.. *An Introduction to Multiagent Systems.* John Wiley & Sons; 2009.
32. Pokahr Alexander, Braubach Lars, Jander Kai. *The Jadex Project: Programming Model.*. 2013.

33. Bellifemine Fabio, Poggi Agostino, Rimassa Giovanni. JADE: A FIPA2000 Compliant Agent Development Environment. In: AGENTS '01:216–217ACM; 2001; New York, NY, USA.

34. Gutknecht Olivier, Ferber Jacques. The madkit agent platform architecture. *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*. 2001;.

35. Cetnarowicz K., Kisiel-Dorohinicki M., Nawarecki E.. The application of evolution process in multi-agent world (MAW) to the prediction system. In: Tokoro M., ed. *Proc. of the 2nd Int. Conf. on Multi-Agent Systems (ICMAS'96)*, AAAI Press; 1996.

36. Byrski A., Schaefer R., Smołka M.. Asymptotic Guarantee of Success for Multi-Agent Memetic Systems. *Bulletin of the Polish Academy of Sciences—Technical Sciences*. 2013;61(1).

37. Byrski Aleksander, Kisiel-Dorohinicki Marek. Agent-Based Model and Computing Environment Facilitating the Development of Distributed Computational Intelligence Systems. In: Allen Gabrielle, Nabrzyski Jarosław, Seidel Edward, Albada GeertDick, Dongarra Jack, Sloot PeterM.A., eds. *Computational Science – ICCS 2009*, Lecture Notes in Computer Science, vol. 5545: Springer Berlin Heidelberg 2009 (pp. 865-874).

38. Schaefer Robert, Byrski Aleksander, Kolodziej Joanna, Smolka Maciej. An Agent-based Model of Hierarchic Genetic Search. *Comput. Math. Appl.*. 2012;64(12):3763–3776.

39. Korczyński Wojciech, Byrski Aleksander, Dębski Roman, Kisiel-Dorohinicki Marek. Classic and agent-based evolutionary heuristics for shape optimization of rotating discs. *Computing and Informatics*. 2017;36(2):331–352.

40. Pisarski Sebastian, Rugała Adam, Byrski Aleksander, Kisiel-Dorohinicki Marek. Evolutionary Multi-Agent System in Hard Benchmark Continuous Optimisation. In: Esparcia-Alcázar AnnaI., ed. *Applications of Evolutionary Computation*, Lecture Notes in Computer Science, vol. 7835: Springer Berlin Heidelberg 2013 (pp. 132-141).

41. Byrski Aleksander. Tuning of agent-based computing. *Computer Science*. 2013;14(3)):491–512.

42. Wrobel Krzysztof, Torba Pawel, Paszynski Maciej, Byrski Aleksander. Evolutionary Multi-Agent Computing in Inverse Problems. *Computer Science (AGH)*. 2013;14(3):367–384.

43. Turek Wojciech. Erlang as a High Performance Software Agent Platform. *Advanced Methods and Technologies for Agent and Multi-Agent Systems*. 2013;252:21.

44. Manate B., Munteanu V.I., Fortis T.-F.. Towards a Scalable Multi-agent Architecture for Managing IoT Data. In: :270-275; 2013.

45. Byrski Aleksander, Dębski Roman, Kisiel-Dorohinicki Marek. Agent-based computing in an augmented cloud environment. *International Journal of Computer Systems Science & Engineering*. 2012;27(1):7–18.

46. Di Stefano Antonella, Santoro Corrado. Supporting agent development in Erlang through the eXAT platform. In: Springer 2005 (pp. 47–71).

47. Piotrowski M., Turek W.. Software Agents Mobility Using Process Migration Mechanism in Distributed Erlang. In: Erlang '13:43–50ACM; 2013; New York, NY, USA.

48. Díaz Á.F., Earle C.B., Fredlund L-A. eJason: An Implementation of Jason in Erlang. In: Dastani Mehdi, Hübner J.F., Logan Brian, eds. *Programming Multi-Agent Systems*, Lecture Notes in Computer Science, vol. 7837: Springer Berlin Heidelberg 2013 (pp. 1-16).

49. Frank Andrew U., Bittner Steffen, Raubal Martin. Spatial and Cognitive Simulation with Multi-agent Systems. In: Montello DanielR., ed. *Spatial Information Theory*, Lecture Notes in Computer Science, vol. 2205: Springer Berlin Heidelberg 2001 (pp. 124-139).

50. Grigore Claudia, Collier Rem. Supporting agent systems in the programming language. In: :9–12IEEE Computer Society; 2011.

51. Vo Huy T, Osmari Daniel K, Summa Brian, Comba João LD, Pascucci Valerio, Silva Cláudio T. Streaming-Enabled Parallel Dataflow Architecture for Multicore Systems. In: 1, vol. 29: :1073–1082Wiley Online Library; 2010.

52. Akidau Tyler, Bradshaw Robert, Chambers Craig, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*. 2015;8(12):1792–1803.

53. Johnston Wesley M, Hanna JR, Millar Richard J. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*. 2004;36(1):1–34.

54. Dennis Jack B. First version of a data flow procedure language. In: :362–376Springer; 1974.

APPENDIX

A THE IMPACT OF THE PARAMETERS OF THE MODEL ON THE OPTIMIZATION PROCESS

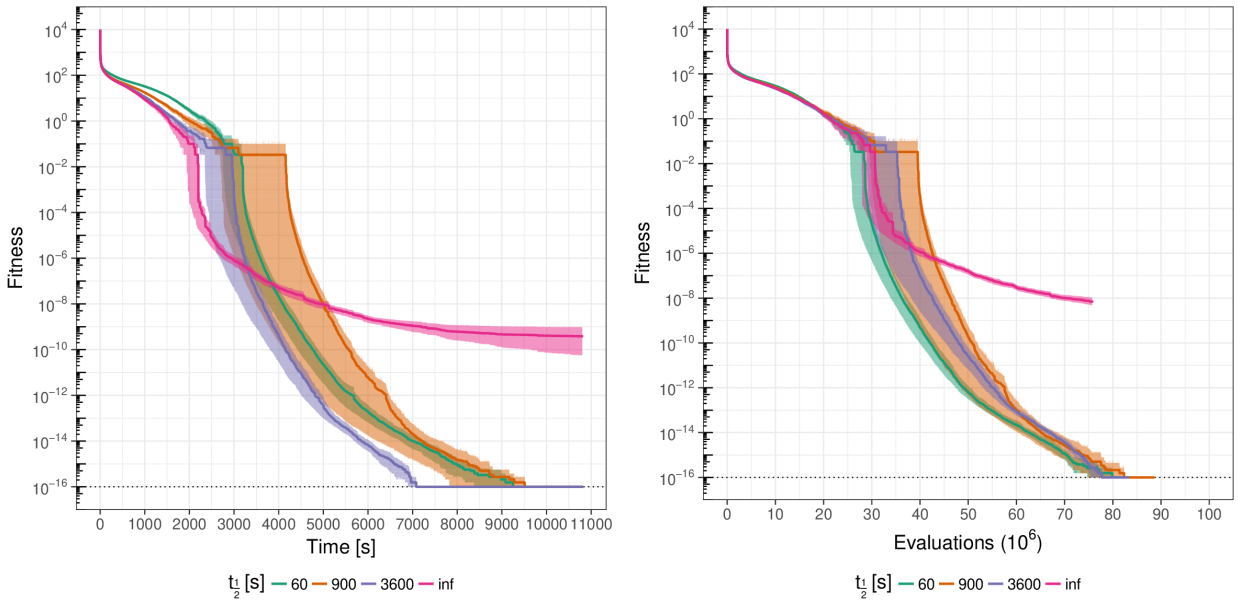


FIGURE A1 Rastrigin function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where the buffer size was equal to **10**. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

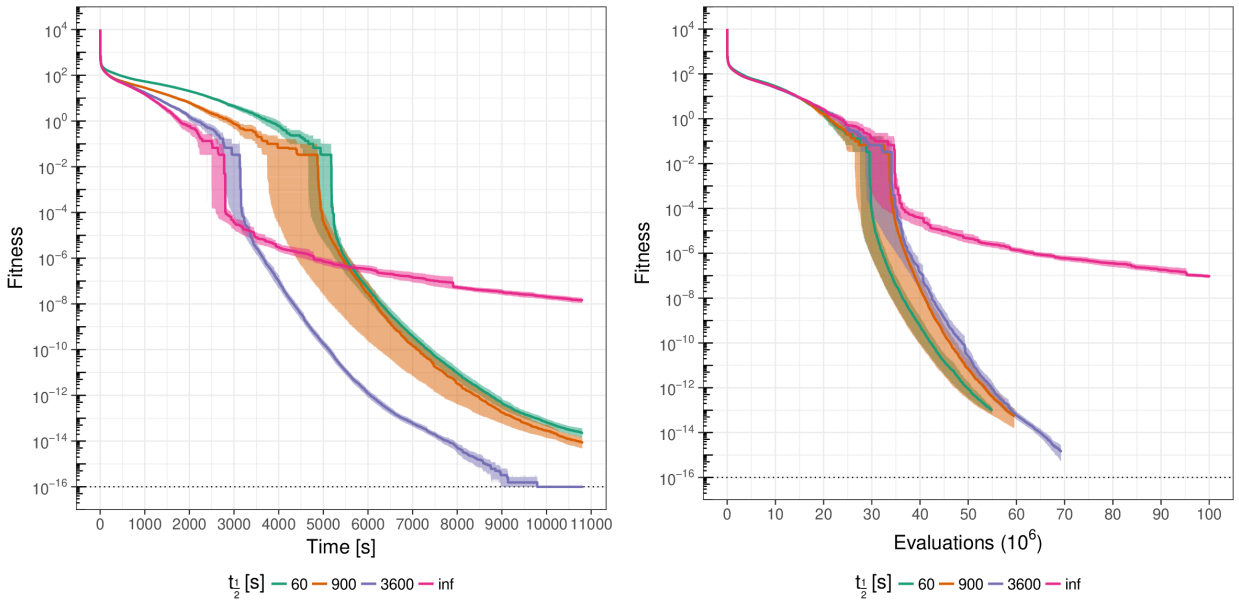


FIGURE A2 Rastrigin function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where the buffer size was equal to **40**. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

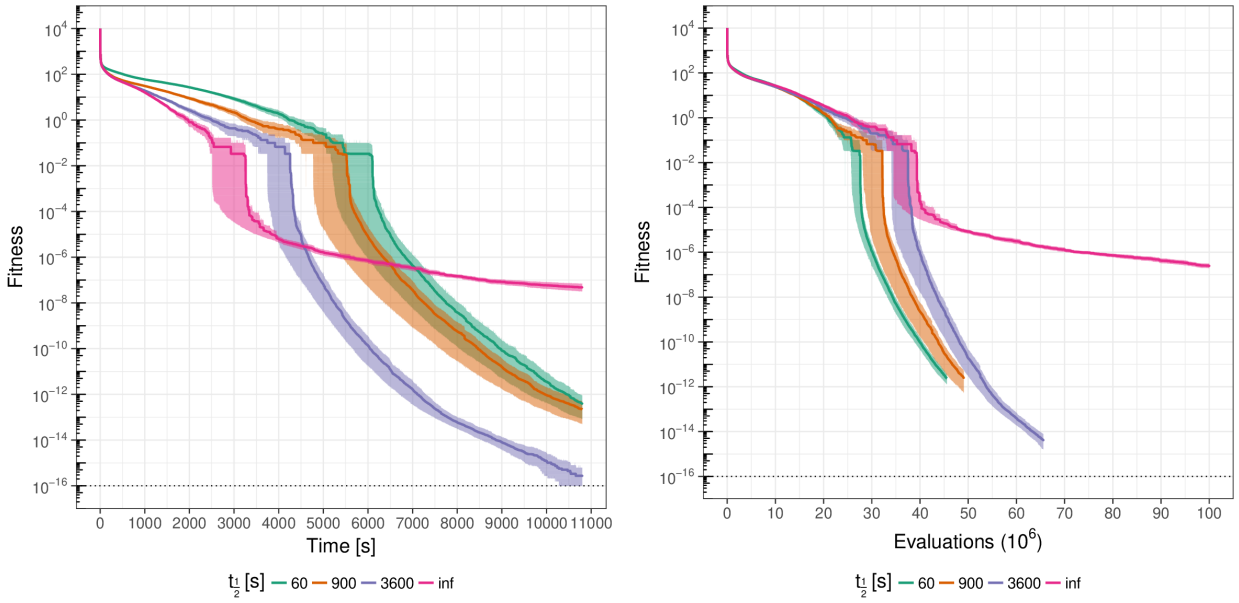


FIGURE A3 Rastrigin function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where the buffer size was equal to **100**. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

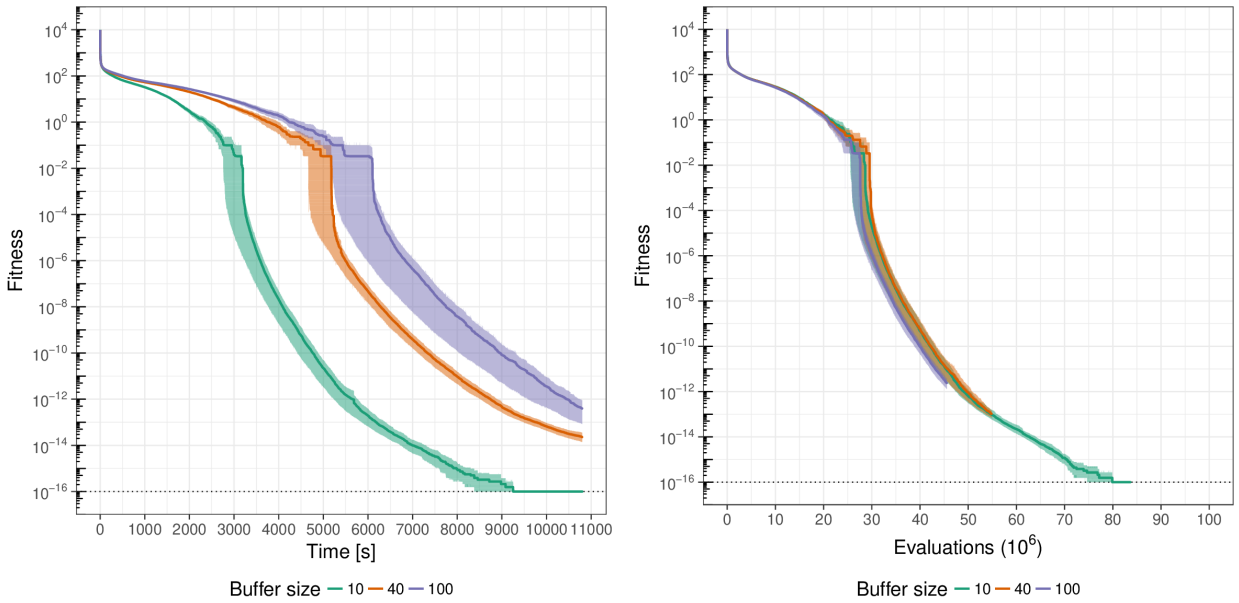


FIGURE A4 Rastrigin function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where $t_{\frac{1}{2}} = 60$s. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

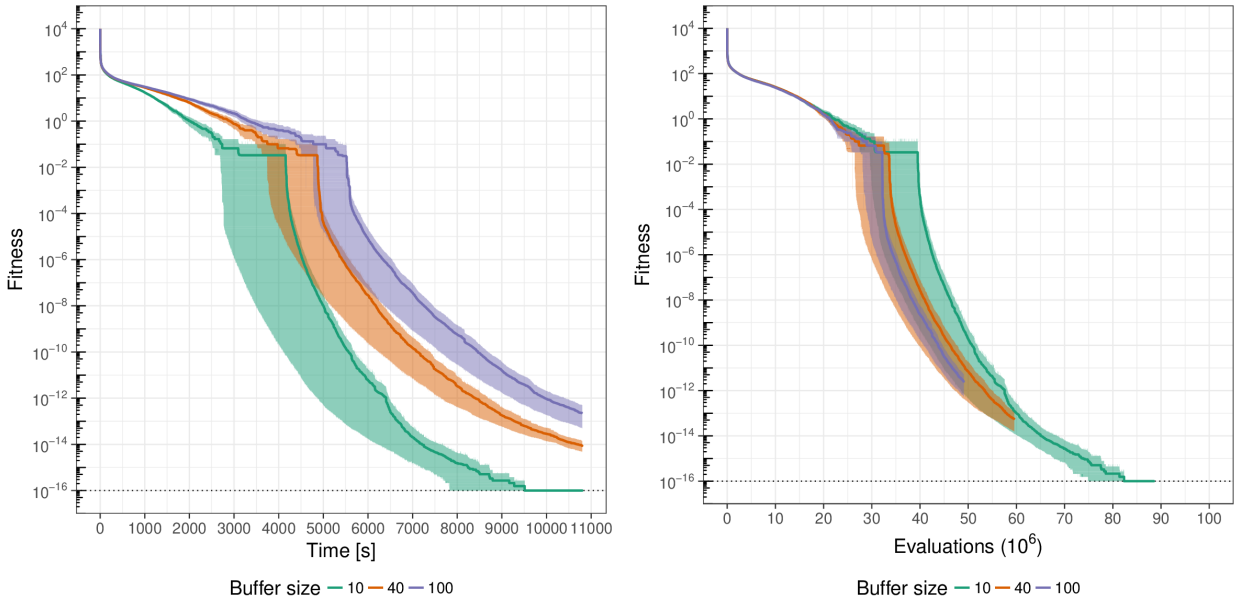


FIGURE A5 Rastrigin function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where $t_{\frac{1}{2}} = 900$s. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

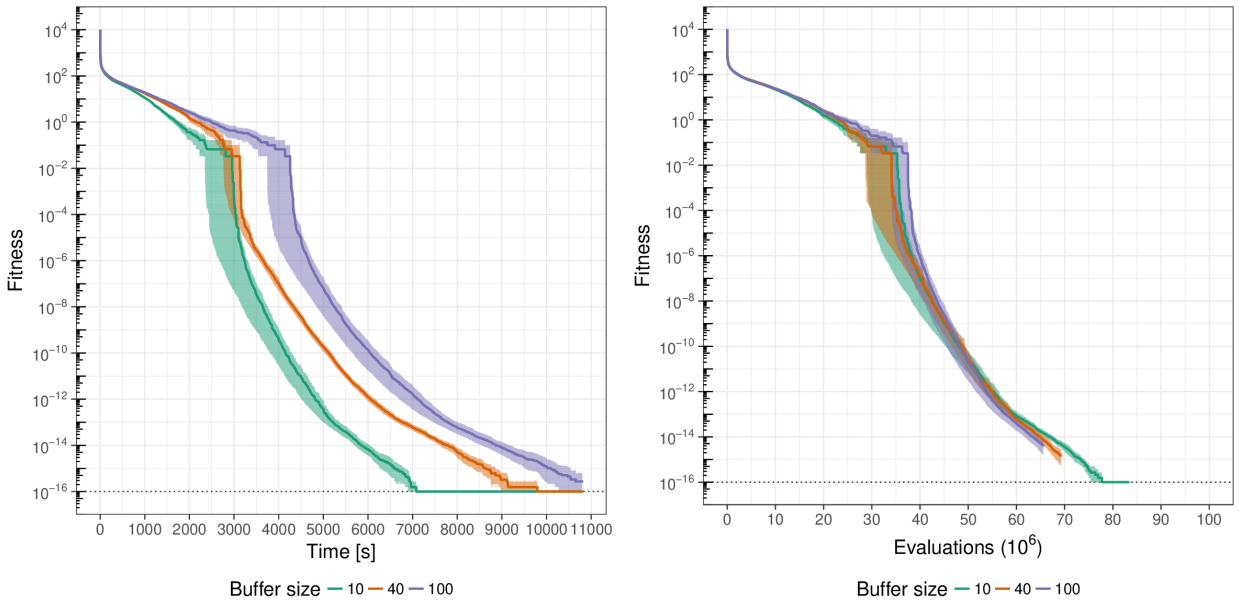


FIGURE A6 Rastrigin function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where $t_{\frac{1}{2}} = 3600s$. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

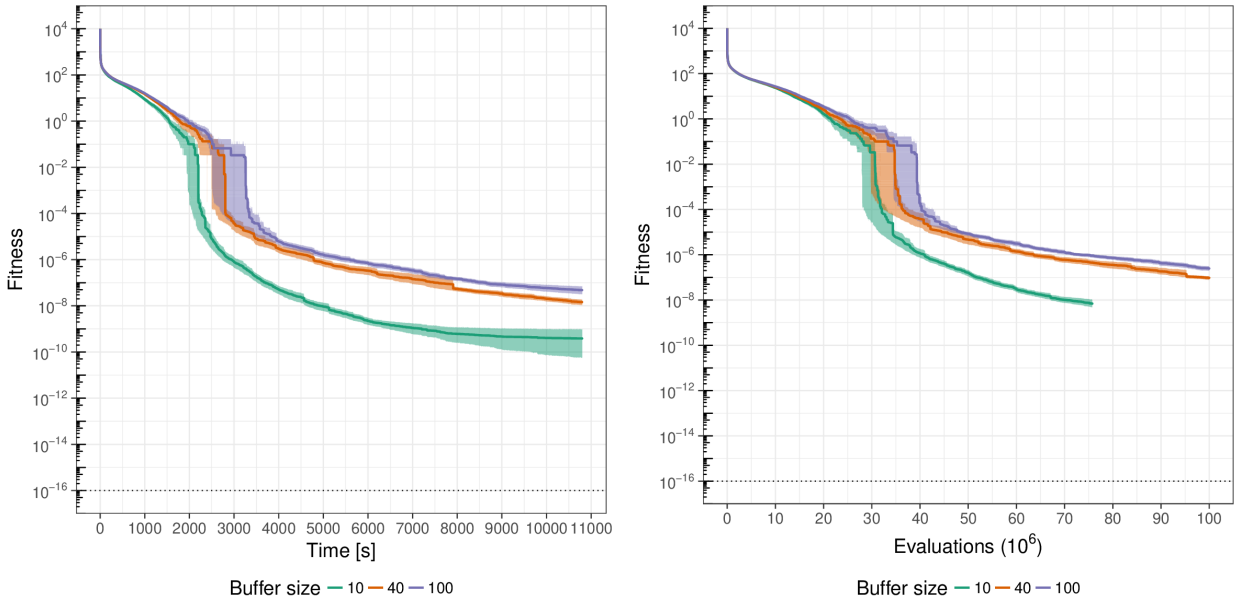
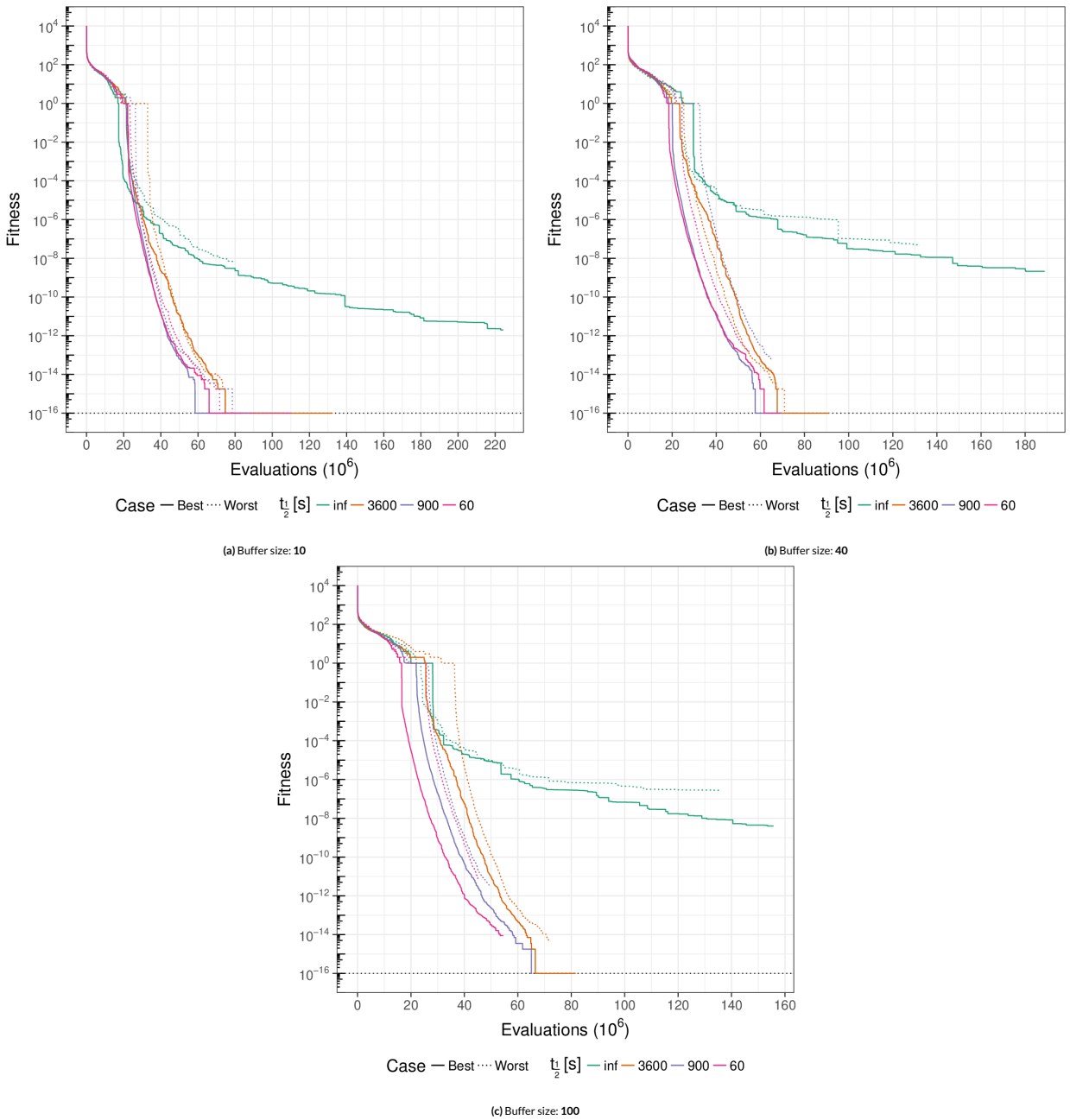


FIGURE A7 Rastrigin function. Best fitness value as a function of time (on the left) and evaluation number (on the right), for configurations of the annealed buffer where the $t_{\frac{1}{2}}$ parameter was disabled (therefore equivalent to a random buffer). A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.



(a) Buffer size: **10**

(b) Buffer size: **40**

(c) Buffer size: **100**

FIGURE A8 Rastrigin function. Fitness as a function of the number of evaluation for the best and worst runs of the annealed buffer, for different buffer sizes. A 10^{-16} constant has been added to results to visualize the global optimum.

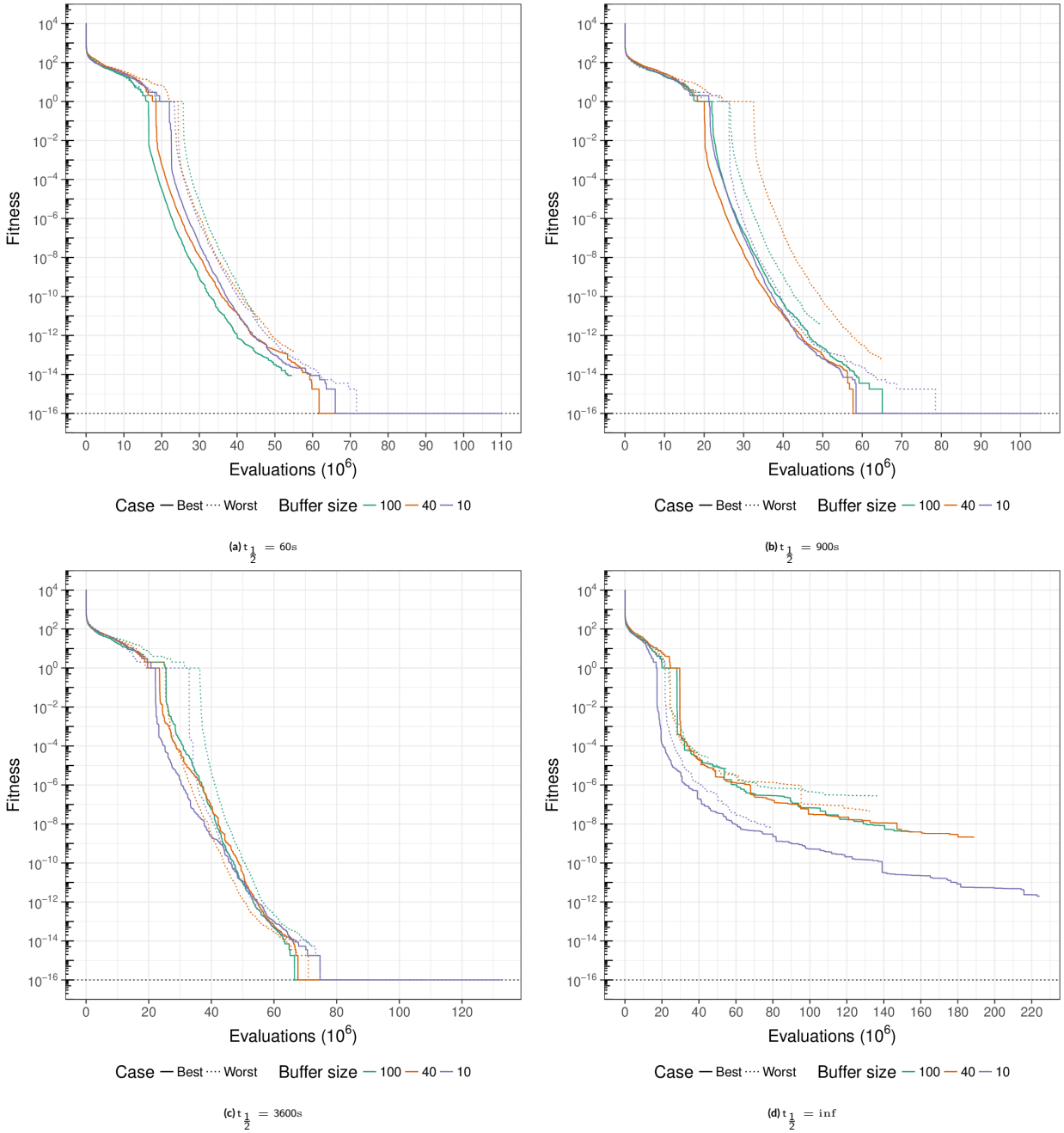


FIGURE A9 Rastrigin function. Fitness as a function of the number of evaluation for the best and worst runs of the annealed buffer, for different $t_{\frac{1}{2}}$ values. A 10^{-16} constant has been added to results to visualize the global optimum.

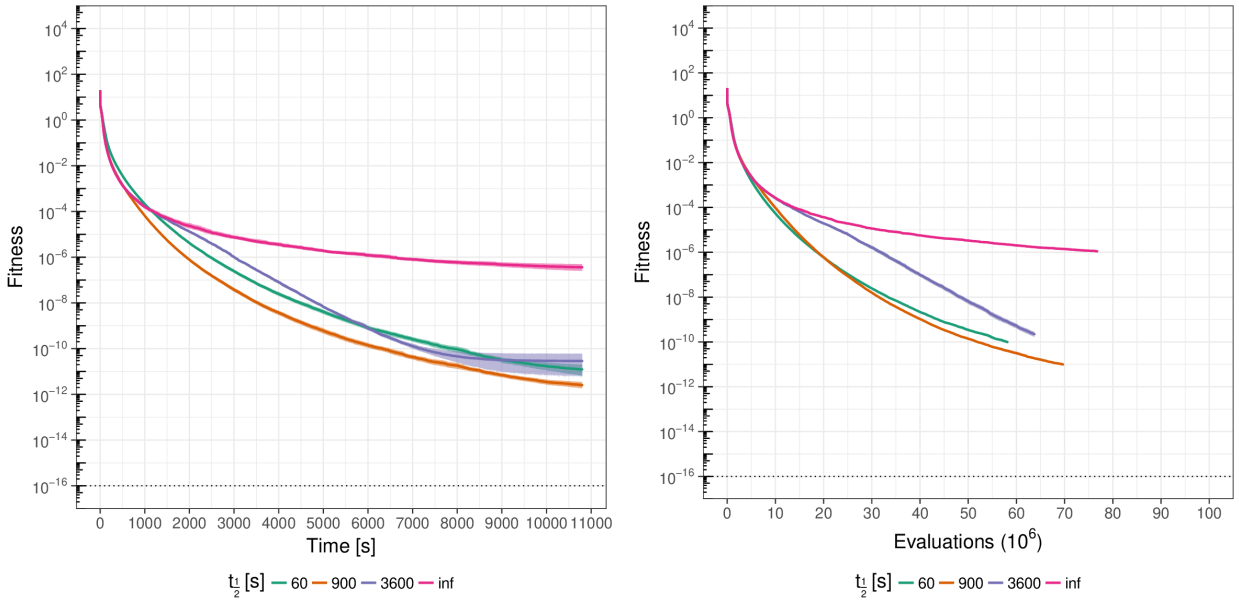


FIGURE A10 Ackley function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where the buffer size was equal to **10**. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

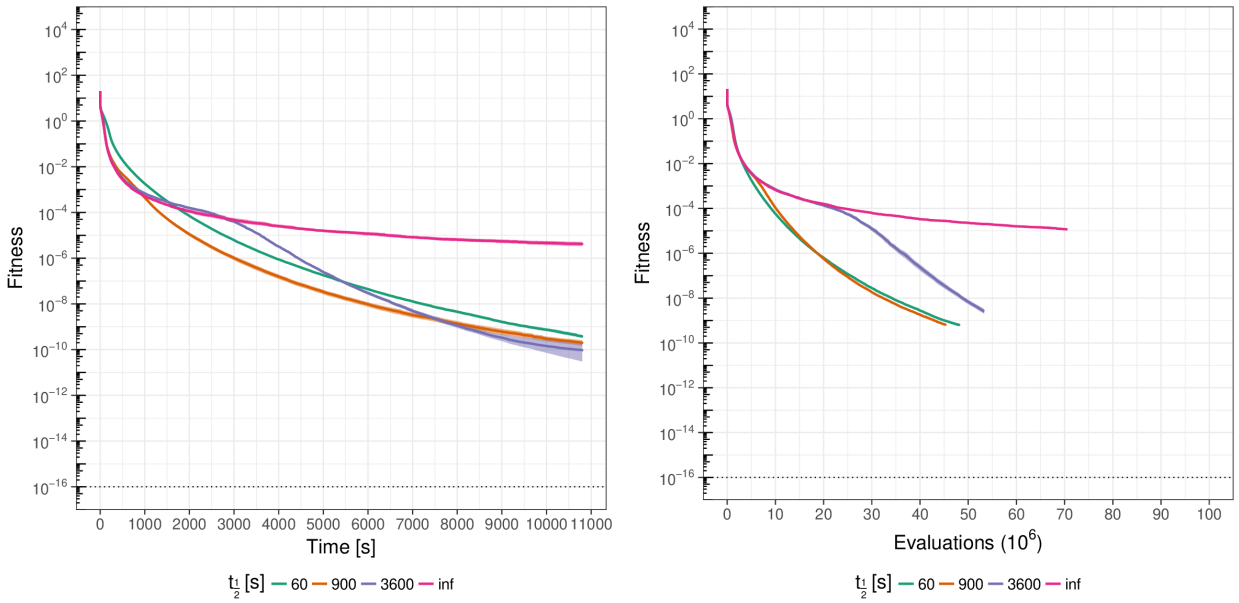


FIGURE A11 Ackley function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where the buffer size was equal to **40**. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

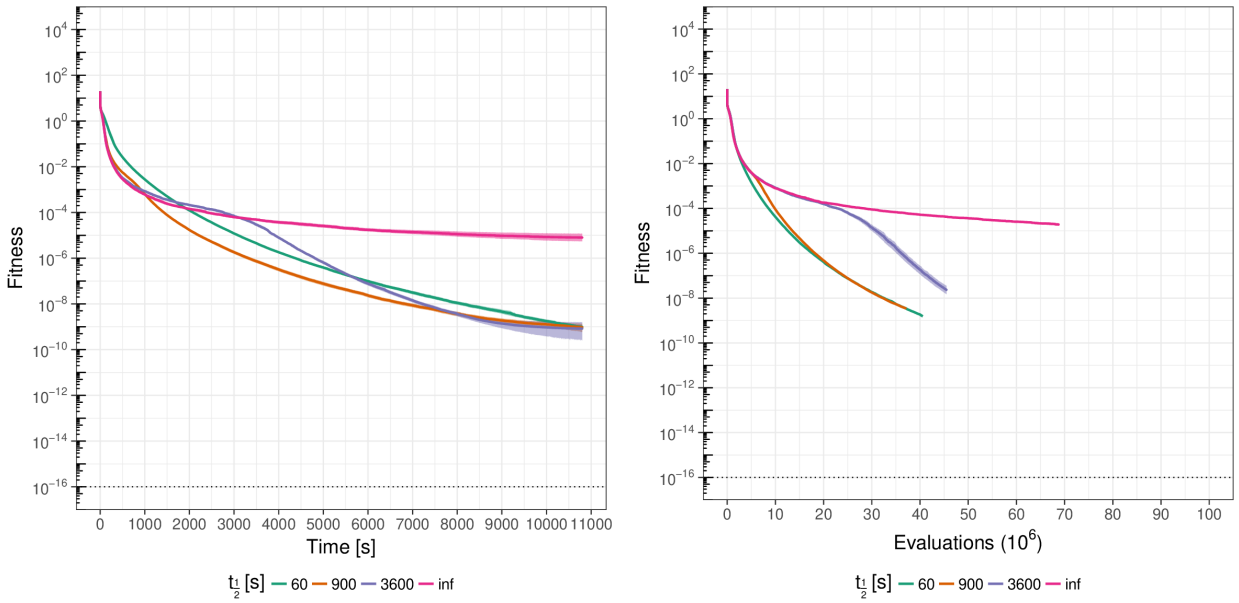


FIGURE A12 Ackley function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where the buffer size was equal to **100**. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

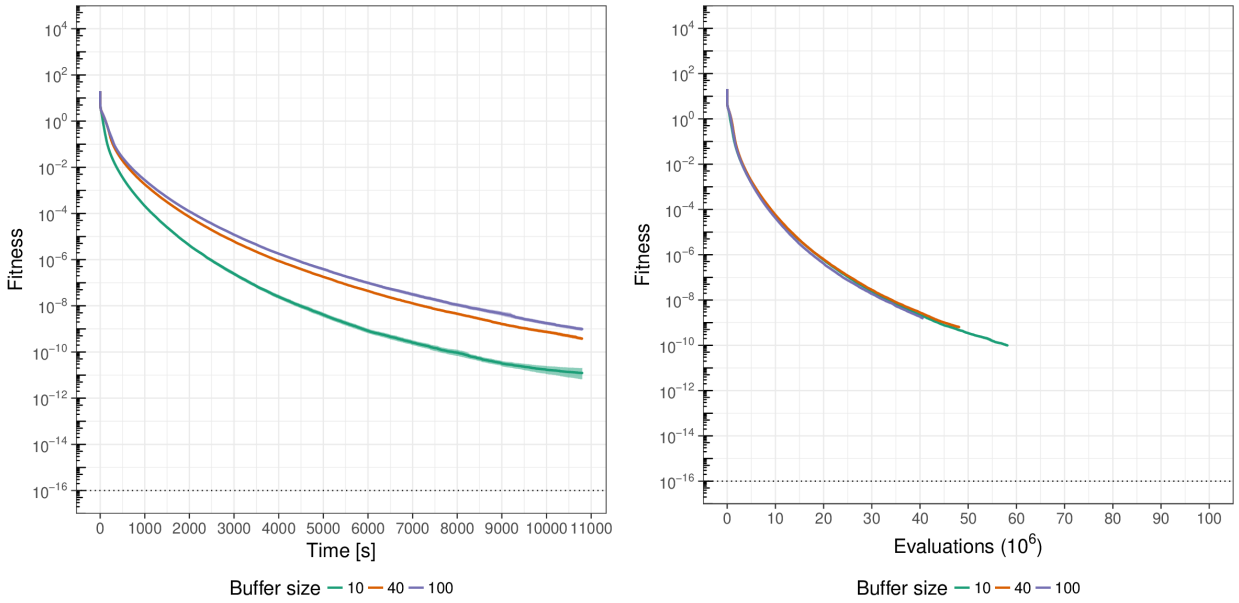


FIGURE A13 Ackley function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where $t_{\frac{1}{2}} = 60$s. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

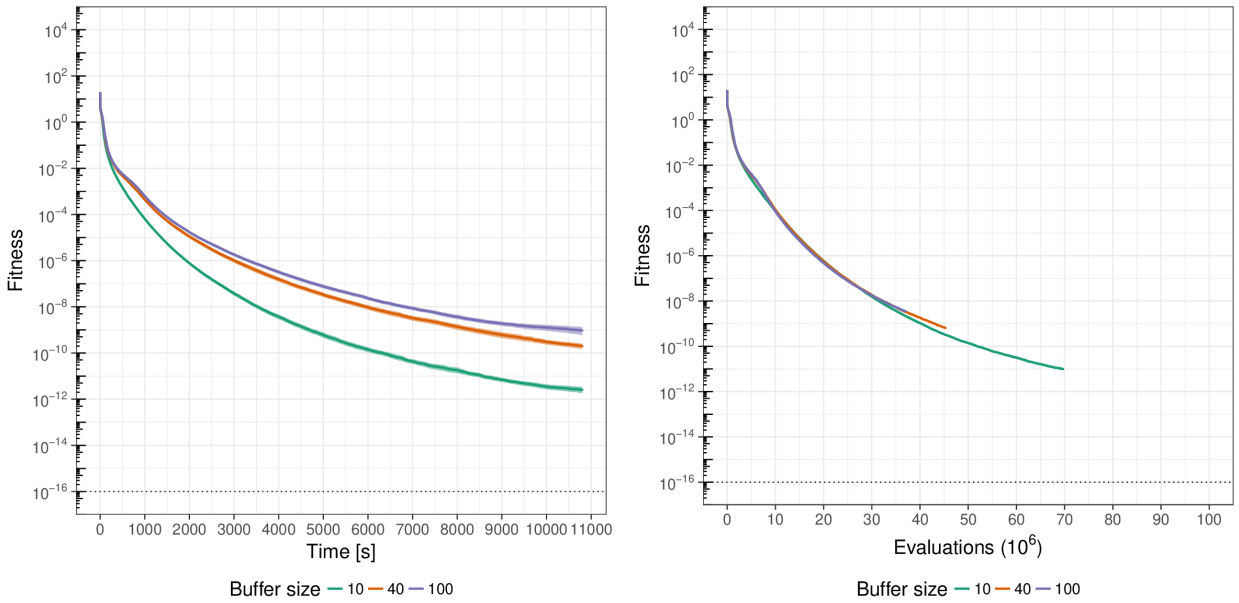


FIGURE A14 Ackley function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where $t_{\frac{1}{2}} = 900s$. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

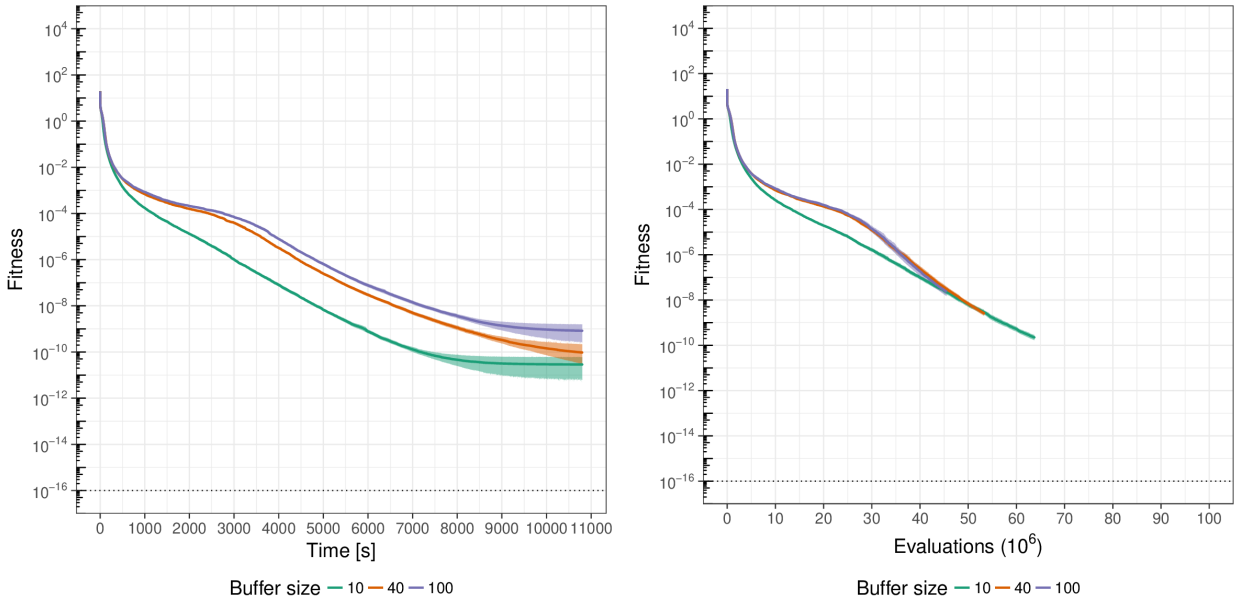


FIGURE A15 Ackley function. Best fitness value as a function of time (on the left) and evaluation number (on the right) for configurations of the annealed buffer where $t_{\frac{1}{2}} = 3600s$. A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

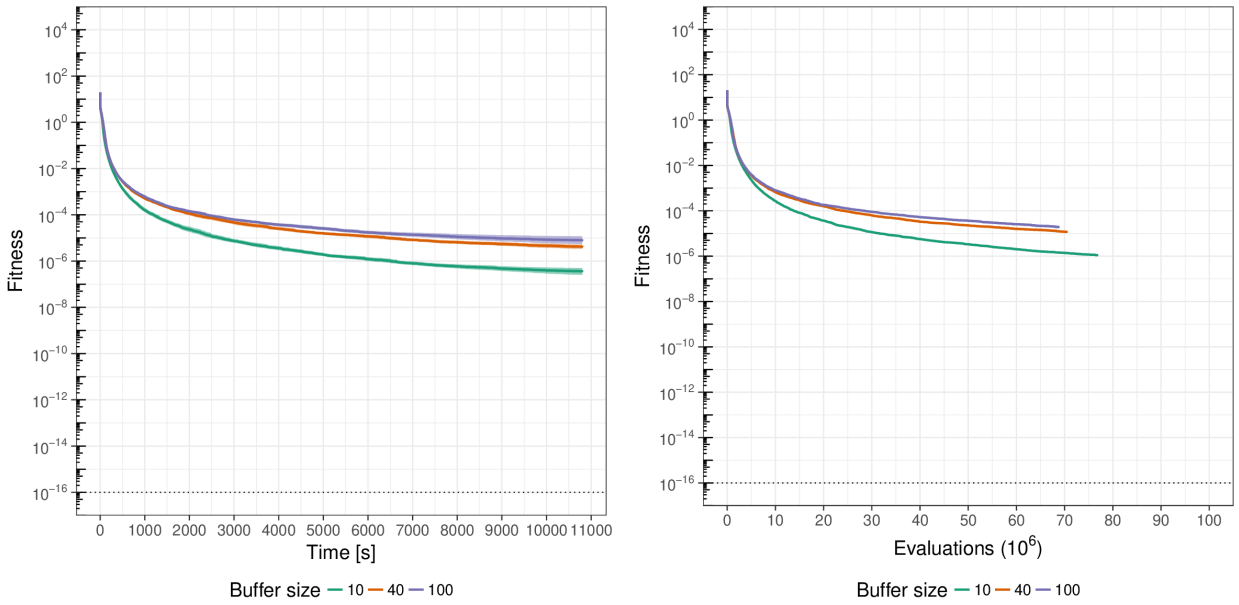
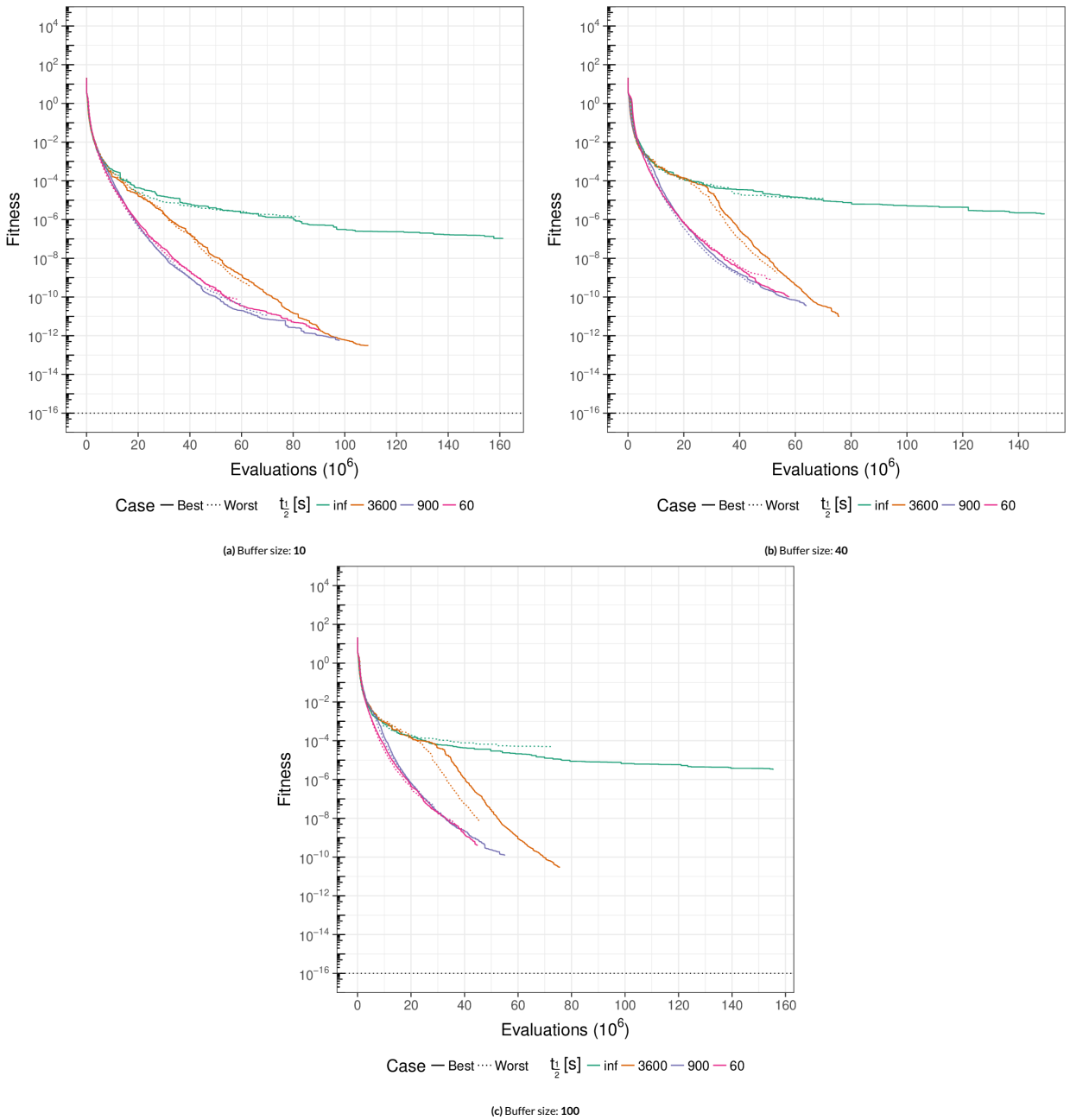


FIGURE A16 Ackley function. Best fitness value as a function of time (on the left) and evaluation number (on the right), for configurations of the annealed buffer where the $t_{\frac{1}{2}}$ parameter was disabled (therefore equivalent to a random buffer). A 10^{-16} constant has been added to results to visualize the global optimum. The darker line represents the mean value and the lighter ribbon shows a 95% confidence interval.

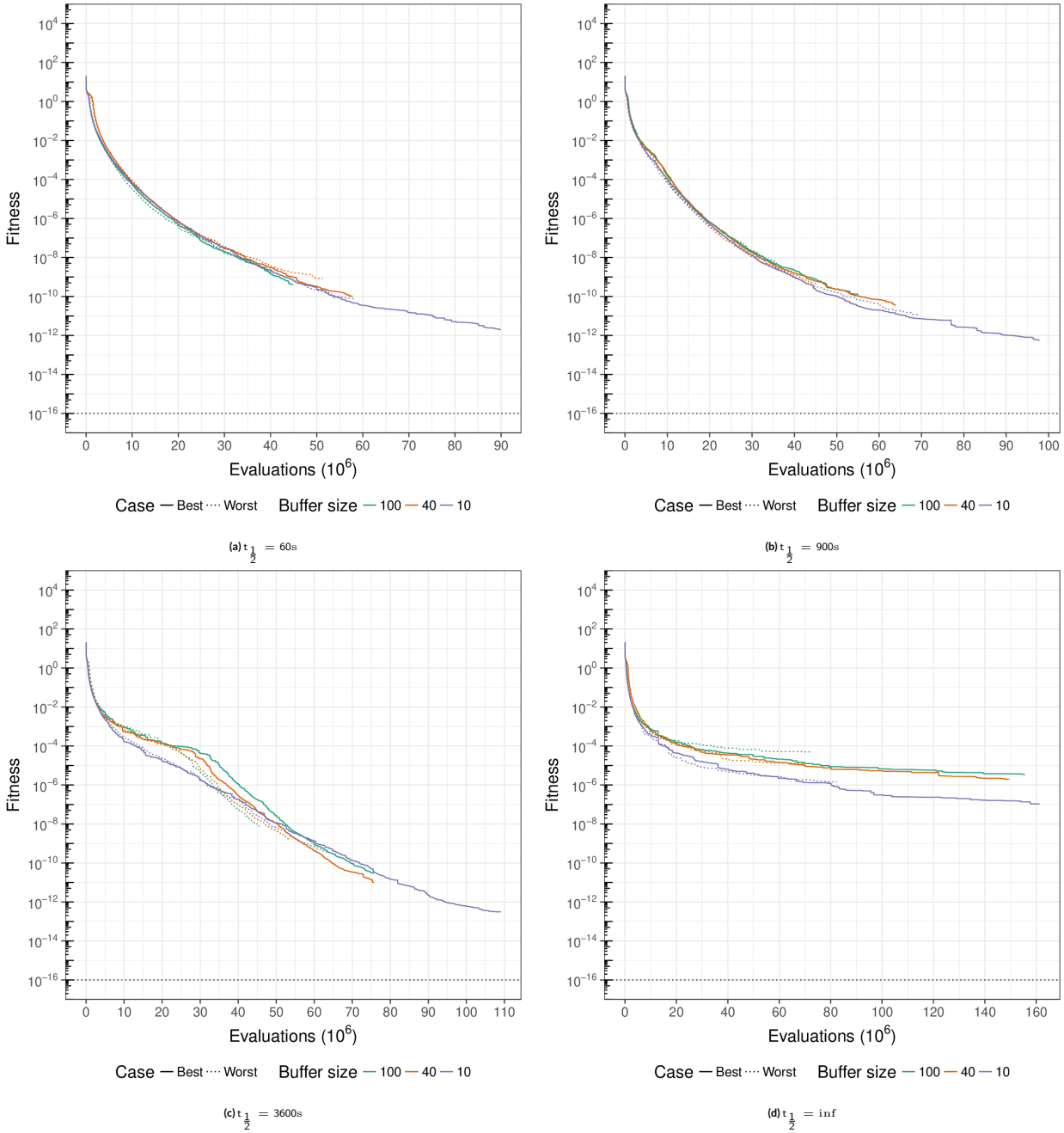


(a) Buffer size: **10**

(b) Buffer size: **40**

(c) Buffer size: **100**

FIGURE A17 Ackley function. Fitness as a function of the number of evaluations for the best and worst runs of the annealed buffer, for different buffer sizes. A 10^{-16} constant has been added to results to visualize the global optimum.



(a) $t_{\frac{1}{2}} = 60s$ (b) $t_{\frac{1}{2}} = 900s$

(c) $t_{\frac{1}{2}} = 3600s$ (d) $t_{\frac{1}{2}} = inf$

FIGURE A18 Ackley function. Fitness as a function of the number of evaluations for the best and worst runs of the annealed buffer, for different $t_{\frac{1}{2}}$ values. A 10^{-16} constant has been added to results to visualize the global optimum.

4. Overview of Experimental Results

The different execution models described in Section 3 were subjected to experiments aimed at analyzing their properties, both with regard to their raw efficiency as well as their possible impact on the semantics of the algorithm. I summarize below the methodology and experimental results described in more detail in the corresponding papers.

The experiments consisted in applying the multi-agent algorithm to classical optimization problems: searching for the minimal values of the Rastrigin function or the Ackley function – common benchmarking functions used to compare evolutionary algorithms [34]. These functions are highly multimodal with one global minimum equal to 0 at $\bar{x} = 0$.

Methodology All the results were obtained by running simulations on the Pl-Grid[1] infrastructure at the ACC Cyfronet AGH[2]. The precise hardware used in experiments is detailed in every paper, but the experiments were mostly run on 12 core nodes.

The actor-based execution model was implemented in Scala using the Akka library and also in native Erlang. The skeleton-based model was implemented in Erlang using the Skel library [3]. The dataflow-based execution model was implemented in Scala using the Akka Streams library. The code of the implementations is open source [4].

Metrics Two metrics were recorded in all experiments. The first was the best fitness value found so far at any time. This metric measures how effective is the algorithm, for example by observing the convergence to the known solution.

The second metric was the total number of fitness function evaluation performed at any time [5]. It is an estimation of the efficiency of the algorithm. In real-word optimization problems, the cost of computing a single fitness value is usually high. Therefore, an algorithm which achieves similar results using less fitness function evaluations is more efficient. Additionally, the number of fitness function evaluation indicates the raw throughput of the system and is a useful measure of scalability.

[1] http://www.plgrid.pl/en

[2] http://www.cyfronet.krakow.pl/en/

[3] https://github.com/ParaPhrase/skel

[4] https://github.com/ParaPhraseAGH/erlang-mas, https://github.com/eleaar/scala-mas

[5] Some of the papers report on the number of agent reproductions. Fitness evaluation of the children happens after the reproduction of the parents, therefore the two measures are strongly related

A third, derived metric is the best fitness value found *after a given number of fitness function evaluations*. Running the same algorithm on a faster node (or running a parallel algorithm on a node with more cores) will allow to find the optimum faster, but in part because there will be more fitness function evaluations in every unit of time. This metric "normalizes" the efficiency of the algorithm and as such enables us to meaningfully compare experiments on different nodes, with different numbers of cores.

Results Several conclusions can be drawn from the experimental results across the publications. Some selected graphs from the publications are included below for illustrative purposes.

The first conclusion is that all models exhibit very good scalability with regard to the number of cores of the underlying node, at least as good as a trivial parallelization of the synchronous algorithm using the island model (one environment per thread and intermittent agent migrations between threads) (Fig. 4.1). This almost linear scalability has been demonstrated for up to 12 cores. Some further experiments suggest that the skeleton-based model scales linearly well beyond, even up to 64 cores, but that is still subject of ongoing research. In other words, decoupling the semantics of the algorithm from its execution model does not hinder scalability.

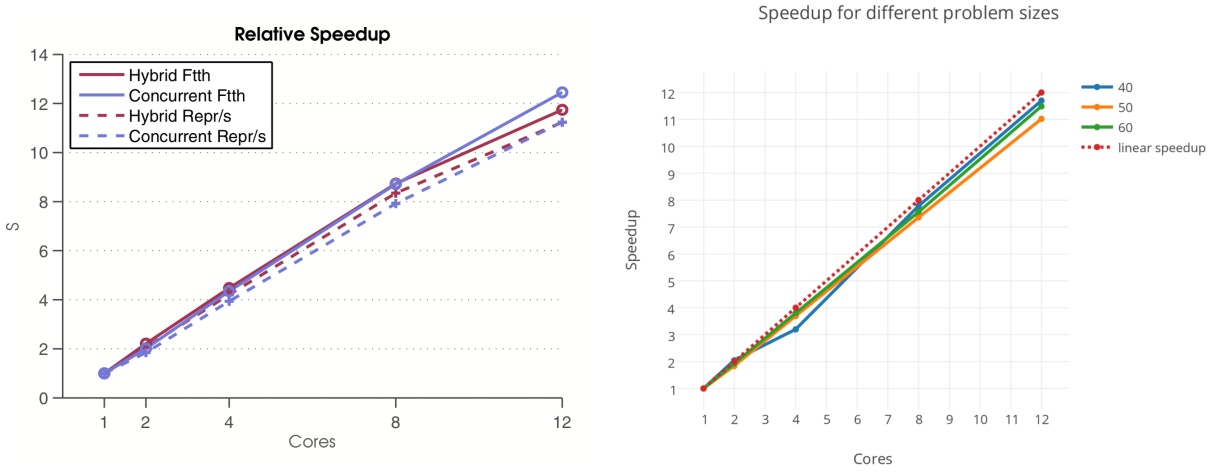


Figure 4.1. The scalability of actor-based (left) and skeleton-based (right) execution models. Both graphs illustrate the scalability of the number of agent reproductions in a unit of time, which is proportional to the number of fitness evaluations during a unit of time. On the left, the actor-based model is labeled as "concurrent", while the "hybrid" one is a trivial parallelization of the synchronous model with concurrent islands.

The second conclusion is that the amount of concurrency in the computation has an impact on the semantics of the algorithm. Both in the actor-based model and the dataFlow-based model, the normalized efficiency of the optimization algorithm was significantly better than in the synchronous one (Fig. 4.2). In other words, a solution of the same quality could be reached within much less fitness evaluations. This result was consistent no matter the number of cores or the parameters of the model itself. The skeleton-base model, which has parallelism but no concurrency, behaved similar to the synchronous

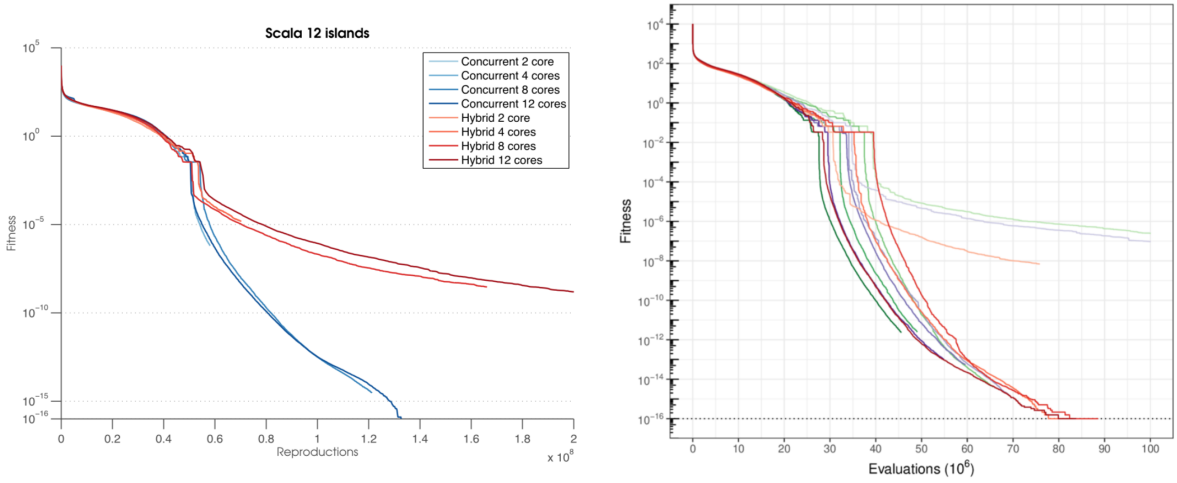


Figure 4.2. The normalized efficiency of actor-based (left) and dataFlow-based (right) execution models for the Rastrigin function, compared to equivalent synchronous models. A 10^{-16} constant has been added to results to visualize the global optimum. On the left, the actor-based model is labeled as "concurrent", while the "hybrid" one is a trivial parallelization of the synchronous model with concurrent islands. On the right, the better group of series correspond to the dataflow-based model with an annealed shuffling policy, while the "flatter" group of series correspond to a emulated synchronous model. Finally, note that the x-axis has different units in both graphs: 1 reproduction/s corresponds to 1-2 evaluations/s, so the dataflow-based model is even more efficient than the actor-based one.

model, which further strengthen these results. The dataflow-based model was also slightly better than the actor-based one.

Finally, the characteristics of the synchronous model emulated on the dataflow-based model are consistent with those of the native synchronous model. This proves that the dataflow-based model is indeed able to simulate other execution models.

5. Conclusions

The following dissertation is concerned with computationally demanding multi-agent systems, in which the number of agents is very large and the interactions between them are intensive or non-trivial. Interactions are intensive when they occur very often or when many calculations are needed to determine their outcomes. They are non-trivial when it is not possible to predict beforehand and in a general way which agents will interact with each other. The main subject of this dissertation has been the concurrency of such interactions between agents, understood as the way agents perceive their interactions and the effects of these. The manner in which interactions between agents are organized were referred to as their *concurrent execution model*.

In the context of the objectives described in Section 1.2, the main results of this dissertation are the following:

– I have formalized the concurrent execution model of agent-based computations as the composition of a *behavior* and a *meetings* functions applied on a population of agents.

– I have shown that this formalization makes is possible to decouple the semantics of the algorithm from the underlying execution model, on the example of a computationally intensive use case of agent-based computing, namely Evolutionary Multi-Agent Systems.

– I have investigated, described, and implemented several concurrent execution models based on different concurrency paradigms.

– I have performed an experimental assessment of the performance and efficiency of different concurrent execution models. I have shown how the choice of the execution model affects both the behavior of the algorithm and of the efficiency of the computation.

– In particular, I demonstrated that in the case of solving an optimization problem, increased concurrency leads to a significantly more efficient algorithm; the same results can be achieved using a smaller number of fitness function evaluations.

– I have introduced a dataflow-based execution model which is efficient on modern multi-core hardware and allows for controlling the concurrency of agent interactions to the extend that it is able to simulate other execution models.

5.1. Contributions and achievements

The main contributions from the research described in this dissertation are as follows:

– The separation of the semantics from the execution model of agent-based computation, demonstrated in this dissertation, will make it possible to meaningfully compare alternative execution models for the same algorithm, in order to make an informed choice to best match a specific hardware architecture or problem size;

– Looking at the same from the opposite perspective, this decoupling also makes it possible to explore new execution models to best use the increasing capabilities of modern many-core hardware;

– In particular, the dataflow-based proves to be the most promising, both in terms of efficiency and extensibility;

– More generally, the findings from this research could help to improve the existing software used for agent-based computing and, consequently, allow for the modeling or solving of more complex problems.

Publications This dissertation is based on the following series of publications discussing the different aspects of the research objectives:

A.1 **Daniel Krzywicki**, Aleksander Byrski, Marek Kisiel-Dorohinicki
Computing agents for decision support systems
Future Generation Computer Systems, 2014.
MNiSW list A, 40 points, IF 2.786

A.2 **Daniel Krzywicki**, Wojciech Turek, Aleksander Byrski, Marek Kisiel-Dorohinicki
Massively concurrent agent-based evolutionary computing
Journal of Computational Science, 2015
MNiSW list A, 30 points, IF 1.078

A.3 Wojciech Turek, Jan Stypka, **Daniel Krzywicki**, Piotr Anielski, Kamil Pietak, Aleksander Byrski, Marek Kisiel-Dorohinicki
Highly scalable Erlang framework for agent-based metaheuristic computing
Journal of Computational Science, 2016
MNiSW list A, 30 points, IF 1.748

A.4 Jan Stypka, Piotr Anielski, Szymon Mentel, **Daniel Krzywicki**, Wojciech Turek, Aleksander Byrski, Marek Kisiel-Dorohinicki
Parallel patterns for agent-based evolutionary computing
Computer Science, 2016
MNiSW list B, 12 points

A.5 **Daniel Krzywicki**, Łukasz Faber, Roman Dębski
Concurrent agent-based evolutionary computations as adaptive dataflows
Accepted for publication in Concurrency and Computation: Practice and Experience
MNiSW list A, 25 points, IF 1.133

The topic of this dissertation has also been explored in the following publications not included in the above series:

B.1 **Daniel Krzywicki**, Łukasz Faber, Kamil Piętak, Aleksander Byrski, Marek Kisiel-Dorohinicki
Lightweight Distributed Component-oriented Multi-agent Simulation Platform
Proceedings of ECMS, 2013
MNiSW 10 points

B.2 **Daniel Krzywicki**, Jan Stypka, Piotr Anielski, Wojciech Turek, Aleksander Byrski, Marek Kisiel-Dorohinicki
Generation-free agent-based evolutionary computing
Proceedings of ICCS, 2014
MNiSW 10 points

B.3 Grażyna Skiba, Mateusz Starzec, Aleksander Byrski, Katarzyna Rycerz, Marek Kisiel-Dorohinicki, Wojciech Turek, **Daniel Krzywicki**, Tom Lenaerts, Juan C Burguillo
Flexible asynchronous simulation of iterated prisoner's dilemma based on actor model
Simulation Modelling Practice and Theory, 2017
MNiSW list A, 25 points, IF 1.954

Citations As of this writing, my publications are indexed in the following databases:

- Web of Science
  6 publications
  40 citations
  h-index = 4
- Scopus
  6 publications
  45 citations
  h-index = 4
- Google Scholar
  8 publications
  57 citations
  h-index = 4

Conferences During my work on the subject of this dissertation, I have given presentations at the following conferences:

- "Lightweight distributed component-oriented multi-agent simulation platform", 27th European Conference on Modelling and Simulation ECMS 2013, Ålesund, Norway
- "The continuous evolution of asynchronous agents", Lambda Days 2014, Kraków, Poland
- "Generation-free Agent-based Evolutionary Computing", 14th International Conference on Computational Science 2014, Cairns, Australia
- "Scaling functional multi-agent computations with reactive streams" Lambda Days 2018, Kraków, Poland

Grants Part of the research described in this dissertation has been realized or supported through the following grants:

- EU FP7 grant: "Paraphrase - Parallel Patterns for Adaptive Heterogeneous Multicore Systems"
- AGH Dean's grant ("grant dziekański"): "Asynchroniczny model agentowych obliczeń ewolucyjnych"

Scholarships During my participation in PhD studies, I was awarded with the following scholarships:

- AGH PhD scholarship ("stypendium doktoranckie") in years 2012/2013, 2013/2014, 2014/2015, and 2015/2016
- AGH scholarship for best PhD students ("stypendium dla najlepszych doktorantów AGH") in years 2012/2013, 2013/2014, and 2014/2015

Commercial Applications Part of the open-source software created during the work on this dissertation has found commercial application in the optimization of online advertisement search campaigns.

Bibliography

[1] François Bousquet and Christophe Le Page. "Multi-agent simulations and ecosystem management: a review". In: *Ecological modelling* 176.3-4 (2004), pp. 313–332.

[2] Bo Chen and Harry H Cheng. "A review of the applications of agent technology in traffic and transportation systems". In: *IEEE Transactions on Intelligent Transportation Systems* 11.2 (2010), pp. 485–497.

[3] Yoav Shoham. "Agent-oriented programming". In: *Artificial intelligence* 60.1 (1993), pp. 51–92.

[4] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. "JADE: A FIPA2000 Compliant Agent Development Environment". In: *Proceedings of the Fifth International Conference on Autonomous Agents*. AGENTS '01. Montreal, Quebec, Canada: ACM, 2001, pp. 216–217. ISBN: 1-58113-326-X. DOI: *10.1145/375735.376120*.

[5] Wojciech Turek. "Erlang as a High Performance Software Agent Platform". In: *Advanced Methods and Technologies for Agent and Multi-Agent Systems* 252 (2013), p. 21.

[6] Michael Balmer, Kai Nagel, and Bryan Raney. "Large-scale multi-agent simulations for transportation applications". In: *Intelligent Transportation Systems*. Vol. 8. 4. Taylor & Francis. 2004, pp. 205–221.

[7] Christophe Deissenberg, Sander Van Der Hoog, and Herbert Dawid. "EURACE: A massively parallel agent-based model of the European economy". In: *Applied Mathematics and Computation* 204.2 (2008), pp. 541–552.

[8] Aleksander Byrski et al. "Evolutionary multi-agent systems". In: *The Knowledge Engineering Review* 30.2 (2015), pp. 171–186.

[9] D. Krzywicki et al. "Computing agents for decision support systems". In: *Future Generation Computer Systems* (2014). ISSN: 0167-739X. DOI: *http://dx.doi.org/10.1016/j.future.2014.02.002*.

[10] Sean Luke et al. "Mason: A multiagent simulation environment". In: *Simulation* 81.7 (2005), pp. 517–527.

[11] Łukasz Faber et al. "Agent-Based Simulation in AgE Framework". In: *Advances in Intelligent Modelling and Simulation*. Ed. by Aleksander Byrski et al. Vol. 416. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2012, pp. 55–83. ISBN: 978-3-642-28887-6. DOI: *10.1007/978-3-642-28888-3_3*.

[12] Daniel A Reed and Jack Dongarra. “Exascale computing and big data”. In: *Communications of the ACM* 58.7 (2015), pp. 56–68.

[13] K. Cetnarowicz, M. Kisiel-Dorohinicki, and E. Nawarecki. “The application of evolution process in multi-agent world (MAW) to the prediction system”. In: *Proc. of the 2nd Int. Conf. on Multi-Agent Systems (ICMAS’96)*. Ed. by M. Tokoro. AAAI Press, 1996.

[14] M. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. Cambridge, MA, USA: MIT Press, 1998. ISBN: 026222058X.

[15] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, 1989.

[16] Hans-Paul Schwefel and Gunter Rudolph. “Contemporary Evolution Strategies”. In: *European Conference on Artificial Life*. 1995, pp. 893–907.

[17] A. Byrski, W. Korczyński, and M. Kisiel-Dorohinicki. “Memetic Multi-Agent Computing in Difficult Continuous Optimisation”. In: *Proceedings of 6th International KES Conference on Agents and Multi-agent Systems Technologies and Applications, 2013, Hue City, Vietnam, IOS Press (accepted in 2013)*. Springer.

[18] S. Pisarski et al. “Evolutionary Multi-Agent System in Hard Benchmark Continuous Optimisation”. In: *Proc. of EVOSTAR Conference, Vienna*. IEEE (accepted for printing), 2013.

[19] Aleksander Byrski. “Tuning of Agent-based Computing”. In: *Computer Science (accepted)* (2013).

[20] A. Byrski et al. “Evolutionary Multi-agent Systems”. In: *The Knowledge Engineering Review* (2013 (accepted for printing)).

[21] Krzysztof Wrobel et al. “Evolutionary Multi-Agent Computing in Inverse Problems”. In: *Computer Science (AGH)* 14.3 (2013), pp. 367–384. DOI: *10.7494/csci.2013.14.3.367*.

[22] M. Polnik, M. Kumiega, and A. Byrski. “Agent-based optimization of advisory strategy parameters”. In: *Journal of Telecommunications and Information Technology* 2 (2013), pp. 54–55.

[23] N. R. Jennings, K. Sycara, and M. Wooldridge. “A Roadmap of Agent Research and Development”. In: *Journal of Autonomous Agents and Multi-Agent Systems* 1.1 (1998), pp. 7–38.

[24] S. W. Mahfoud. “A Comparison of Parallel and Sequential Niching Methods”. In: *Proceedings of the 6th International Conference on Genetic Algorithms*. Ed. by L. J. Eshelman. Morgan Kaufmann, 1995, pp. 136–143.

[25] Daniel Krzywicki. “Niching in evolutionary multi-agent systems”. In: *Computer Science* 14 (2013).

[26] John Vlissides et al. “Design patterns: Elements of reusable object-oriented software”. In: *Reading: Addison-Wesley* 49.120 (1995), p. 11.

[27] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[28] Carl Hewitt, Peter Bishop, and Richard Steiger. “Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence”. In: *Advance Papers of the Conference*. Vol. 3. Stanford Research Institute. 1973, p. 235.

[29] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[30] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[31] Christopher Brown et al. “Cost-directed refactoring for parallel Erlang programs”. In: *International Journal of Parallel Programming* 42.4 (2014), pp. 564–582.

[32] Aleksander Byrski and Robert Schaefer. “Formal model for agent-based asynchronous evolutionary computation”. In: *Evolutionary Computation, 2009. CEC’09. IEEE Congress on*. IEEE. 2009, pp. 78–85.

[33] Jeffrey S Vitter. “Random sampling with a reservoir”. In: *ACM Transactions on Mathematical Software (TOMS)* 11.1 (1985), pp. 37–57.

[34] Thomas Bäck and Hans-Paul Schwefel. “An overview of evolutionary algorithms for parameter optimization”. In: *Evolutionary computation* 1.1 (1993), pp. 1–23.