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 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 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).

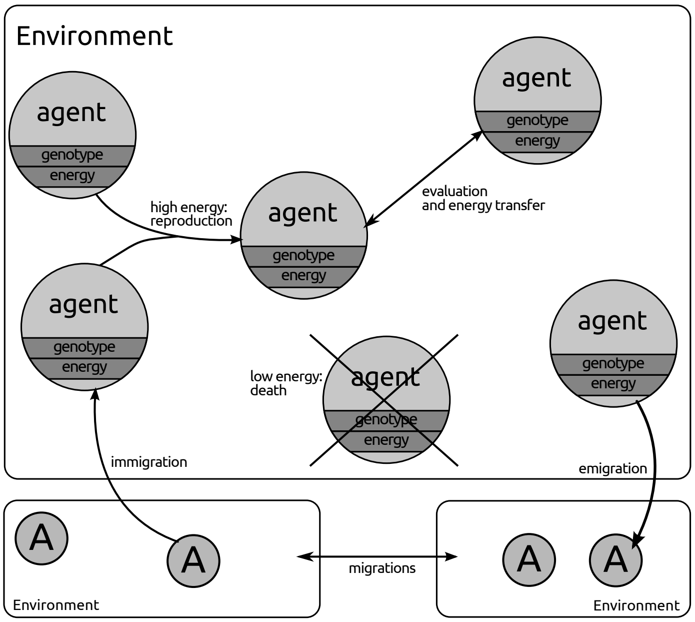


Figure 1. EMAS structure and principle of work.

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.

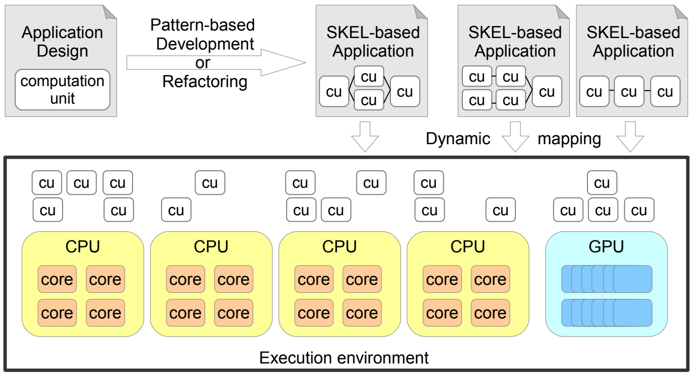


Figure 2. Parallel program execution schema. Application written using Skel as a graph of patterns is dynamically mapped to available hardware.

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.

```
StopCondition = fun(_Agents) -> os:timestamp() < EndTime end.
Skeleton = {feedback, [MainWorkflow], StopCondition}.

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.

```
MainWorkflow = {pipe, [{seq, GroupAgents},
                       {map, [{seq, UpdateAgents}], Workers},
                       {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.

```
GroupAgents = fun (Agents) ->
           AgentsWithAction = lists:map(ChooseAction, Agents),
           Migrated = lists:map(Migrate, AgentsWithAction),
           GroupByAction(Migrated)
              end,


UpdateAgents = fun({{Island, Behavior}, Agents}) ->
                  NewAgents = Meetings({Behavior, Agents}),
                  [{Island, A} || A~<- NewAgents]
               end,

Shuffle = fun(Agents) ->
              shuffle(lists:flatten(Agents))
          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.

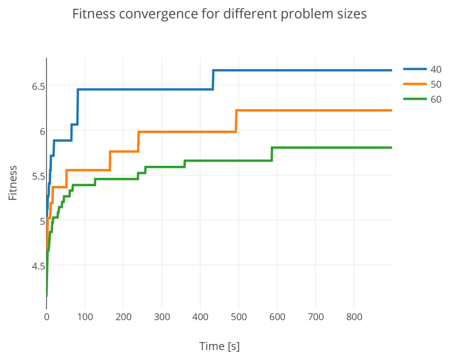
6.3. Experiment results

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.

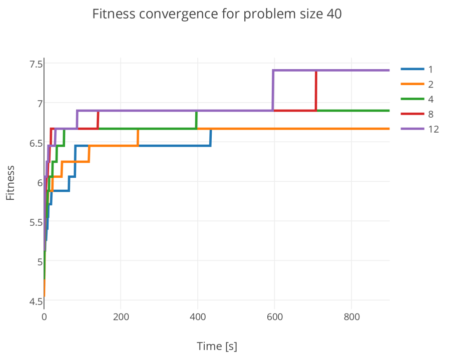
To assess the scalability of the system, we recorded the intensity of interactions in the system, represented by the number of agent reproductions happening every

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

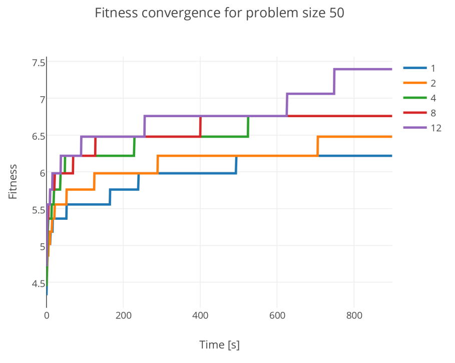
[8]`http://www.cyfronet.krakow.pl/en/`



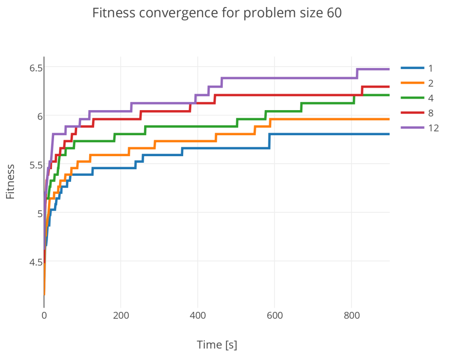
(a) Fitness convergence for different problem sizes on 1 CPU core.



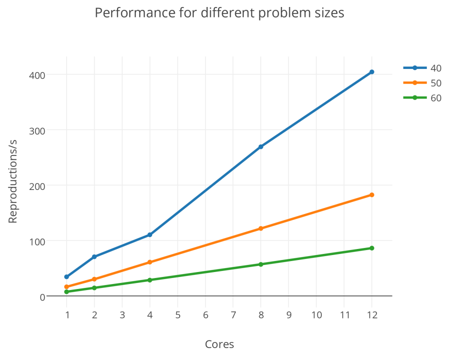
(b) Fitness convergence for problem size 40 on different cores.



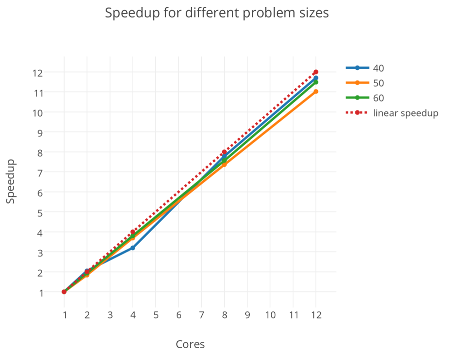
(c) Fitness convergence for problem size 50 on different cores.



(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