Project no. 288570

PARAPHRASE

Strategic Research Partnership (STREP)
PARALLEL PATTERNS FOR ADAPTIVE HETEROGENEOUS MULTICORE SYSTEMS

MAS Framework and Use cases Report
D6.7

Due date of deliverable: 31 Jan. 2014

Start date of project: October 1\(^{st}\), 2011
Type: Deliverable
WP number: WP6
Task number: T6.5

Responsible institution: AGH
Editor and editor’s address: Aleksander Byrski, et al. olekb@agh.edu.pl

Version 0.1 (Revision)

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PU</strong> Public</td>
<td></td>
</tr>
<tr>
<td><strong>PP</strong> Restricted to other programme participants (including the Commission Services)</td>
<td>√</td>
</tr>
<tr>
<td><strong>RE</strong> Restricted to a group specified by the consortium (including the Commission Services)</td>
<td></td>
</tr>
<tr>
<td><strong>CO</strong> Confidential, only for members of the consortium (including the Commission Services)</td>
<td></td>
</tr>
</tbody>
</table>
Executive Summary

This deliverable describes the multi-agent applications and use case scenarios selected by AGH within WP6 (Use Cases, Requirements and Evaluation). We describe the motivation and evaluation criteria for these use cases, along with details of particular example algorithms and simulations.

We also formulate high-level multi-agent patterns which can be used to model the chosen use cases and show how these high-level patterns may be expressed in terms of the generic patterns (deliverable D2.1) and application specific patterns (deliverable D2.3) provided by Paraphrase.
# Contents

Executive Summary .................................................. 1

1. Introduction ......................................................... 3

2. Multi-Agent Systems Use Cases .................................. 5
   2.1 Applications .................................................... 5
   2.2 Use Cases ..................................................... 6
     2.2.1 Agent-based simulations ................................. 6
     2.2.2 Agent-based computing ................................. 6
   2.3 Algorithms, Relevant Patterns, Hardware and Expectations .... 6
     2.3.1 Predator-Prey Simulation ............................... 6
     2.3.2 Crowd Simulation ....................................... 7
     2.3.3 Evolutionary Multi-Agent Systems .................... 8

3. High Level Patterns for Multi-Agent Systems .................... 9
   3.1 Patterns for Agents .......................................... 9
   3.2 System Examples ............................................. 10
     3.2.1 Predator-prey ........................................... 10
     3.2.2 Crowd Simulation ..................................... 10
     3.2.3 EMAS ..................................................... 11

4. Conclusion .......................................................... 13
1. Introduction

The agent paradigm has been widely studied over the last few decades. Many different applications for agent systems have been tested and the paradigm has definitely proven its usability and led to significant advantages in many areas, including distributed computations [3], optimization [4] or control of multi-robot systems [9,11]. Although researchers do differ in the precise definition of an agent, the basic assumptions and concepts of the paradigm are commonly accepted [5,6]. These include the most basic idea of autonomy of coexisting entities, which is based on the concept of the Actor model proposed in the seventies [8]. The need of interactions between autonomous agents leads to another basic concept, which is communication based on asynchronous messages.

Autonomous agents, which can communicate using asynchronous messages, require a proper execution environment. Focusing on software agents, the environment can be defined as a computer system, which allows the execution of agents threads or processes and supports mechanisms for transferring messages from one agent to another. Such a system is typically called an agent platform.

It is relatively easy to create an agent platform, therefore many different implementations have been developed over last decades. Although the basic features are common, the platforms can differ significantly in the underlying technology and the services implemented. The noble goal of the standardization of the services provided by agent platforms was the foundation of the creation of the FIPA organization [1]. The commonly used standards, like FIPA Agent Communication Language (ACL), specify frames for development of agent platforms.

The variety of services provided by the most advanced agent platforms is impressive. However, there is one issue, which significantly reduces their usability in production applications: performance and limitations. The problem of agent platform performance has received significant attention over the last years. A detailed review of the research performed in this area can be found in [2]. The most general conclusion from these publications is that the performance of wide-spread used agent platform is unsatisfactory and that scalability is insufficient.

The performance of an agent-based system is determined by the performance of software agents and the performance of the agent platform. While the performance of the agents is dependent on the problem solved and can be optimized by the agents’ creator, the performance of the agent platform creates impassable limits.

Agent platforms are typically implemented in popular imperative languages,
using well-established technologies. This approach has many advantages. Existing libraries and components can be reused, platforms can be integrated with other systems and the development of agents is not a hard task. From typical multi-agent applications perspective, the most important seems the support sophisticated communication protocols and/or organization of the system. This is not necessarily true for multi-agent simulations or computational systems. In this case, an agent system could exhibit far better performance if the implementation was based on a technology which natively supports actor model and message passing communication.

It is easy to notice that the basic assumptions of the Erlang technology are very similar to those attributed to agent platforms [10]. Although Erlang creators did not refer to the agent paradigm, Erlang language and runtime environment can definitely be used as an agent platform. Several services of the EVM and the Erlang software development kit (called Erlang Open Telecom Platform, OTP) are very similar to the more advanced services of typical agent platforms. A naming service is a good example of that. Each process has its unique identifier. However, each process can also register in platform naming service to become discoverable by any potential user.

The idea of using Erlang as an agent platform is not completely novel. Several years ago Di Stefano and Santoro published results of their work on an Erlang based agent platform, called eXAT [7]. They focused on providing advanced mechanisms for creating intelligent agents, like a complete rule based system. Unfortunately, the project seems to be discontinued now.

The ParaPhrase project’s aim is to create software that is easy to write but which can still use the hardware effectively. It builds on a multi-level model of parallelism with applications expressed in terms of interacting components and using high-level parallel patterns that will allow components to be refactored to leverage all of the available heterogeneous hardware.

In this document we present the design of such a MAS Framework, built on high-level patterns, along with the description of relevant use cases.
2. Multi-Agent Systems Use Cases

2.1 Applications

The field of multi-agent systems has been an active area of research at the Intelligent Information Systems Group at AGH. We have been involved in several projects using agent-based soft computing techniques in applications ranging from portfolio optimization to crisis management.

There is a number of software tools supporting agent-based programming, one of the most popular being JADE\(^1\). Software agents are used as building blocks in systems which thereby become autonomous, proactive and able to learn. Compliance with the FIPA\(^2\) standards allows to compose such systems and have heterogeneous agents interoperate.

However, these possibilities come at the expense of an increased agent complexity which makes them heavyweight. Evidence suggest that such an approach is ill-suited to applications with massive numbers of short lived, heavily communicating agents [10].

In this project, we focus on such a class of applications with large numbers of more or less homogeneous agents, like agent based simulations and computations. In these applications, agents are a part of the domain rather than of the implementation. These kinds of multi-agent systems are usually closed, therefore interoperability is not an issue. Instead, efficiency and scalability are essential in order to apply them to practical problems.

Such multi-agent systems are also a good candidate for high parallelism, as they usually fall under the Gustafson law - additional resources not only increase the speed of these systems, but also their scope or accuracy, by using more agents.

There are also some platforms targeted at agent-based simulations, e.g. Netlogo, Mason. However, they each assume a particular model of execution in the design of the multi-agent system, i.e. how agents are mapped to hardware resources.

In this project, we want to apply to Paraphrase approach to such multi-agent systems. We want to be able to model agents behaviour, interactions, etc. but abstract them from the way these agents are actually executed (in steps, asynchronously, etc.). In other words, we want to express multi-agent algorithms in

\(^1\)http://jade.tilab.com
\(^2\)http://www.fipa.org
2.2 Use Cases

Considering the discussion in section 2.1, we selected two use cases which present the following properties: a) large amounts of simple, short-lived agents b) intensive computational work per agent c) high potential for parallelism

These use cases presented below are the result of current work in T6.6 (MAS application use case scenarios). They will be the subject of further work in tasks T6.3 (porting and annotation of applications), T6.4 (experimental evaluation).

2.2.1 Agent-based simulations

Many social and natural phenomena can be simulated by modelling them as a multi-agent system. In such a setting, agents are given some goals and rules governing their behaviour. Depending on the problems, agents may move in space, interact with other agents, exchange resources etc. 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. The interesting result is that complex system-wide properties are said to *emerge* from the much simpler interactions of such autonomous agents.

2.2.2 Agent-based computing

Apart from simulating natural phenomena, agents can also be used to solve engineering problems such as function optimization. Multi-agent systems are ideal in representing problems which can be solved using multiple methods by numerous entities with various perspectives. Therefore, the agent-oriented approach can be easily combined with population-based meta-heuristics. 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 meta-heuristics.

2.3 Algorithms, Relevant Patterns, Hardware and Expectations

This section provides concrete examples of algorithms within the use cases described in section 2.2. The expected set of patterns for each use case is highlighted in section 2.3.3.

2.3.1 Predator-Prey Simulation

This simulation models the stability of a predator-prey ecosystem. The system is composed of agents located on a two dimensional grid. Three types of agents...
coexist: predators, prey and vegetation. Predators and prey may change location and there may be multiple such agents within a single location. In contrast, a single vegetation agent is bound to every location.

All agents have some energy. Predators and prey gradually lose energy and die if it reaches zero. In order to survive, predators need to take energy from prey and prey needs to take energy from vegetation. Vegetation grows and gradually recovers energy up to some limit.

The system may also be extended by introducing additional species and a more complex food chain. The system stability depends on the energetic parameters (the amount of energy lost/recovered by agents), the topology of the environment and the structure of the food chain. A stable system tend to maintain the presence of all species over time, even though population size may fluctuate. An unstable system eventually end with one or more species becoming extinct.

Although interesting from a theoretical point of view, this simulation has little practical application in itself. However, it be useful in testing the possibilities of the high-level patterns introduced in the project. As events happening in different locations are independent, we expect that the simulation may be carried out both on CPU and GPGPU cores, with linear speedup up to the number of locations.

2.3.2 Crowd Simulation

Crowding simulations have been used to study emergent behaviours in many systems involving the motion of large numbers of independent individuals. They have helped understanding phenomena such as the flocking of fish or birds, the emergence of traffic lanes in human crowds or traffic jams on highways. They are also a useful tool in crisis management, crowd control and the design of evacuation routes.

Several approaches have been proposed to model crowding behaviour. We focus on a discrete model of crowd simulation where every individual is modelled as an agent moving in some spatial environment. Agents start at some position and move with some velocity to some destination. Many variations can be explored by including strategies such as: trying to keep some minimal space between individuals, following others, avoiding obstacles etc.

A basic simulation will initially target multiple CPU cores. Depending on the additional strategies being used, part of the simulation may also be offloaded to GPGPUs. This use case will test the performance of the multi-agent system when every agent needs to scan a large part of its surroundings in order to make a decision. Although theoretically a linear speedup could be achieved in a sparse system, we expect slightly inferior scalability due to the amount of information likely to be exchanged between agents.
2.3.3 Evolutionary Multi-Agent Systems

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

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 optimisation 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, 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.

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.

The multi-agent level of the computation will target multiple CPU cores, as the stochastic nature of the algorithm may cause agents to perform different actions in a variable amount of time. However, the underlying fitness function evaluation may be efficiently delegated to GPGPUs, depending on the problem being solved. We expect an almost linear scaling with regard to the number of CPU cores and size of the multi-agent system as well as with regard to the number of GPU cores and size of the optimization problem.
3. High Level Patterns for Multi-Agent Systems

Depending on the granularity of agent operations and interactions, different types of parallelism may be more efficient. Therefore, it is advantageous to be able to express a multi-agent algorithm in terms of high-level patterns and leave out implementation details for later. These high-level patterns can be later refactored/mapped to match a specific problem size and the available hardware resources.

3.1 Patterns for Agents

This section proposes high-level patterns allowing to express multi-agent simulations and computations. The full behaviour of the system may be composed from these three kinds of functions:

Agent neighbourhood. Represents the topology of the multi-agent system. A function which for each agent returns its neighbourhood, that is as set of other agents. This function may need created just once or may need to be reseeded each time with some view of the general population.

A few examples:

- identity - returns the same agent
- grid - based on some grid on which agents are. May use a data structure or be implemented as a map-reduce pattern
- similarity - based on some notion of similarity between agents, usually need the full population to be computed

Agent behaviour. Agents receive a read-only state of their neighbourhood and decide what action to take. The semantics of such an action depends on the application.
Agent meeting. Agents with the same action are grouped together and each such group is processed with the use of a meeting function. The meeting function returns a list of agents. As a result of the meeting, existing agents may have state changed or new agents may have been created. Omitting an incoming agent from the output simply removes it from the system at the next iteration.

3.2 System Examples

3.2.1 Predator-prey

In every step, predators and prey should choose the location where to move next, given some topology of the environment. Agents may choose their next move at random or based on the situation around them - depending on the neighbourhood function used. Obviously, vegetation will simply regrow on the same location.

Agents are then grouped for every location chosen. Random predators are allowed to eat prey, random prey is allowed to eat vegetation. Predators and prey pay the energy tax and are removed if their energy drops to zero, vegetation regrows (Listing 3.1).

```
1 meeting(Location, Agents) ->
2 {Preds, Preys, Veg} = split(Agents),
3 {FedPreds, SurvPrey} = feed(Preds, Preys),
4 {FedPrey, SurvVeg} = feed(SurvPrey, Veg),
5
6  [ Y || X <- FedPreds ++ FedPrey ++ SurvVeg,  
7      Y <- updateEnergy(X),  
8      Y#agent.energy > 0].
```

Listing 3.1: The behaviour function chooses the prey’s next move

3.2.2 Crowd Simulation

In every step, agents should receive a view of their surroundings by means of the neighbourhood function. This will usually reflect the field of vision of the agent. Based on the situation around them, agents may choose to change direction or velocity. Their new position will be mapped to a lattice to resolve collisions - agents ending up too close may "bounce off" each other.

The most relevant Paraphrase patterns for this purpose will probably be the "map" pattern to execute agents strategies and partition them on the lattice, followed by the "farm" pattern to process each node of the lattice and correct agents positions. Finally, the "reduce" pattern would combine the results into a new population.
3.2.3 EMAS

In a simple EMAS, every agent choose the action to perform based on its own state. So the neighbourhood function is simply identity and an example of a behaviour function is shown in Listing 3.2.

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

Listing 3.2: The behaviour function chooses an action based on the state of the agent

The population is panmictic, in the sense that every agent can meet any other agent. Agents with similar behaviour are grouped together then randomly paired. For every such pair, a meeting is performed. The agents resulting from the meetings are then combined into the new population (Listing 3.3).

```
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 3.3: Arenas process partitions of the population and trigger agent meetings.

The functions described above can be combined in several ways which correspond to different models of execution.

The simplest way is to sequentially compose the behaviour and meeting functions, together with an intermediate function gathering agents in groups according to their actions. The population can then be iterated by repeatedly applying the step function:

\[
\text{step} = \text{meeting} \circ \text{groupBy} \circ \text{behaviour} \circ \text{id}
\]

A coarse-grained level of parallelism can be introduced by running several Erlang processes with a sequential island in each one. Agents can be migrated between islands through message passing, which results in the following function:

\[
\text{islandStep} = \text{receiveImmigrants} \circ \text{sendEmigrants} \circ \text{step}
\]
A fine-grained level of parallelism can be achieved by computing all agent meetings independently. Instead of keeping different islands in distinct processes, agents can be labelled accordingly so that agents meet only with other agents having the same label, while agent migration simply consist in changing (mutating) the label.

The most relevant Paraphrase patterns for this purpose will probably be the "map" pattern to divide agents into islands and arenas, followed by the "farm" pattern to perform all meetings in parallel. Finally, the "reduce" pattern would combine the results of all meetings on all islands into a new meta-population.
4. Conclusion

The introduction of agent-oriented cases into ParaPhrase opens new possibilities of applications. The creation of a dedicated MAS framework not only gives a possibility of broadening the impact of the whole project, but also paves the way for the development of truly parallel agent-oriented solutions using massive multicore and GPU-based hardware. This is also a new opening in agent-related research, where since the introduction of the eXAT Erlang platform, nothing valuable happened regarding the development of lightweight frameworks targeted at cluster and multicore infrastructure.

In this deliverable we described the use cases chosen by AGH in the application area of multi-agent systems. Additionally, we presented a framework for modelling such multi-agent applications, in the form of high-level patterns. We also discuss the applicability of these patterns to the selected use cases, as well as their relationship to generic Paraphrase patterns.

This deliverable reflects the work of tasks T6.5 and the ongoing work of T6.6. The selected use cases are also beginning to be ported to the available Erlang implementations of Paraphrase patterns. As that work progresses, the use cases may be adapted to the technical possibilities of the existing patterns. The results of application porting will be described in the future deliverable D6.8.
Bibliography


