Project no. 288570

PARAPHRASE

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

MAS Application Porting Report
D6.8

Due date of deliverable: 31 Jul. 2014

Start date of project: October 1st, 2011

Type: Deliverable
WP number: WP6
Task number: T6.6

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>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
</tr>
<tr>
<td>PP</td>
</tr>
<tr>
<td>RE</td>
</tr>
<tr>
<td>CO</td>
</tr>
</tbody>
</table>

Project co-funded by the European Commission within the Seventh Framework Programme
Executive Summary

This deliverable describes the aspects of porting multi-agent applications to the ParaPhrase infrastructure. We focus on two area of applications: multi-agent computations and multi-agent simulations, with specific use-cases described in the appropriate chapters. We show how these systems can be decomposed and parallelized with the use of Paraphrase parallel patterns and concrete skeletons from the skel library.

This deliverable is a direct follow-up of the D6.7 MAS Framework and Use cases Report. In this deliverable, we show how the functional approach described in deliverable D6.7 leads to the design of multi-agent systems which are easy to decompose and therefore, to parallelize. The output from this deliverable are concrete multi-agent applications ported to the skel library. These application will then be fine-tuned, optimized and benchmarked, the results of which will be described in the last D6.9 deliverable.

Figure 1: Positioning of deliverable D6.8 w.r.t. other WP6, WP2 and WP3 deliverables.
Contents

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

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

2. General considerations .................................................. 5

3. Multi-Agent Systems in Optimization ............................... 7
   3.1 Porting ....................................................................... 8
   3.2 Discussion and conclusion ......................................... 10

4. Multi-Agent Systems in Simulation ................................. 12
   4.1 Porting ....................................................................... 13
   4.2 Discussion and conclusion ......................................... 17

5. Conclusion ....................................................................... 18
1. Introduction

According to one of the most popular definitions proposed by Wooldridge, an agent is a computer system situated in an environment, and capable of undertaking independent, autonomous actions in this environment in order to fulfill tasks on behalf of its user [30]. Autonomy is perceived as one of the most crucial features of the agent. A multi-agent system is one that consists of a group of agents which interact with one another [8, 10].

During the last decades intelligent and autonomous software agents have been widely applied in various domains, such as power systems management [21], flood forecasting [9], business process management [11], intersection management [5], or solving difficult optimisation problems [19], just to mention a few.

Agents act in their environment, which defines what they can do and where. Different groups of agents may perform their tasks in different parts of the environment. In particular, their activities may overlap. For example, agents may be able to communicate with each other within a “close” neighbourhood; what a neighbourhood is depends on the topology of the environment in a given application. Another example of activity overlap is when we introduce constraints in the environment, e.g. only one agent-robot may pass through a door at a time, or only one agent may stand on a cell on a grid [29].

Because of these characteristics, it is usually difficult to parallelize a multi-agent system. The interactions between agents are decentralized and complex and it is not easy to determine which actions are truly independent. Most software platforms solve this problem by using thread-contained agent populations which interact only through message-passing or migrations of individuals. However, more fine grained parallelism may be more efficient in some applications.

In this deliverable, we show how the functional approach described in deliverable D6.7 leads to the design of multi-agent systems which are easy to decompose and therefore, to parallelize. We also demonstrate how the parallel patterns from the skel library can be used to that extent.

Chapter 2 starts with general considerations concerning our approach to decompose a multi-agent system, presented with appropriate background and the discussion about the specific requirements. Chapter 3 and 4 follow with a detailed description of our porting efforts in two application areas: agent-based computing and agent-based simulations. For each of these areas, we introduce specific use cases and discuss how to parallelize them by introducing parallel patterns and the
skel library implementation. Finally, chapter 5 summarizes our current efforts and outlines future work.
2. General considerations

A multi-agent system typically consists in a collection of agents which repeatedly:
- acquire information from their environment
- make some decision to act based on their knowledge and state
- act to modify their state and their environment

The environment of an agent consists of the state of the other agents and some global properties. A multi-agent system can also be spatially embedded - in that case additional spatial fields can be included in the environment. In spatial simulations, agents are able to interact with the environment only within some spatial neighbourhood, such as a sphere with a given cut-off radius.

Fine grained parallelism can be achieved in such multi-agent systems with the use of the actor model [17, 27]. However, this approach will not be efficient in many applications which require frequent global synchronization, such as most spatial simulations.

In this project, we describe a coarse-grained approach to parallelize discrete time multi-agent computations and simulations. We assume discrete time simulations and computations, as these are most easily modelled as a data-centric workflow typical of Paraphrase.

**Decomposing the system** Although agents are independent by nature, their actions may affect each other during a single time step. Therefore, the key to introduce parallelism in such a system is identify those agents or those actions which do not depend on each other or will not interfere with each other.

In other words, if we can partition the agents space or their actions space into mutually independent partitions, each such subset can be processed in parallel. The results can then be reduced into the population at the next time step.

There are several ways of decomposing a multi-agent system. In this document we describe two types of use-cases corresponding to two approaches:

- Functional decomposition, i.e. to separate agent actions which by definition do not interfere with each other (see Sec. 3.1).

- Spatial decomposition, i.e. to separate agent actions which could interfere with each other but will not because of spatial constraints, such as a cut-off radius (see Sec. 4.1).
Resolving conflicts It is common in multi-agent systems to have multiple agents compete for a single resource, spatial location, etc. Two types of strategies are usually used to resolve such conflicts.

The first one is to choose a winner deterministically based on the properties of the agents and some predefined rules. The other strategy is to choose an agent at random. Usually, this will be achieved by combining a policy of first-come first-served and a random order of iteration over the collection of agents.

In both cases all the relevant agents need to be gathered in the same partition of the multi-agent system, either to be able to compute the deterministic decision or to keep pseudorandom properties.

Therefore, the multi-agent system must be decomposed so that all agents competing for a single resource belong to the same partition. That is not an obstacle to parallelism as resources in a multi-agent system are usually distributed and local to only some agents - there should be no global resources and therefore no single, global partition.
3. Multi-Agent Systems in Optimization

Metaheuristics are a group of complex search techniques, usually inspired by nature, that should only be applied to solve difficult computing problems [28]. Generally speaking, evolutionary metaheuristics process a population of individuals representing exemplary solutions to a certain problem. The goal is to find an optimal solution (or solutions) to the problem by maximising a predefined goal function (also called “fitness function”) used to evaluate these individuals.

These individuals contain a genotype which is an encoded solution to the optimization problem. The genotype consists of genes, describing different features of the solution. Different representations are used in different problems: continuous problems require binary or real-value based representation, while combinatorial problems usually require permutation representations.

Metaheuristics may easily be connected with agent-based computing techniques. Because of its decentralized nature [30], the agent approach is well suited to design scalable distributed models and has been applied in various applications requiring optimization.

In 1996, Krzysztof Cetnarowicz proposed an evolutionary multi-agent system (EMAS) [2] dedicated to solving computing problems, with interesting features like distributed selection and lack of global control. Since then, the idea of EMAS has been applied to different problems (e.g. single, multimodal [26] and multicriteria optimisation).

This approach is extensible and can be used with other metaheuristic techniques (e.g. with cultural or memetic local search mechanism [3, 14, 16]. Applications to decision support have also been considered [15].

In this use case, we consider a simple EMAS where agents contain a genotype and some energy. Agents fight by comparing their phenotypes (the evaluation of their genotypes), losers give some energy to the losers. When agents gather enough energy, they start to reproduce. The genotypes of the children are derived from their parents’ using classical evolutionary operators. The children also receive some energy from their parents. When an agent’s energy drops to zero, it dies. The selection pressure emerges from these energetic dynamics.

Agent-based computing systems are not easy to program concurrently because
of intense and unpredictable interactions between agents. A functional approach, as opposed to a now traditional object-oriented one, is more appropriated in that context. Designing the multi-agent computation in terms of the high-level functions described in deliverable D6.7 (e.g. listing 3.1) makes it easier to reason about the algorithm and refactor it in order to introduce parallelism.

### Listing 3.1: The behaviour function in a simple Evolutionary Multi-Agent System

```plaintext
behaviour(Agent) ->
  {_, _, Energy, Island} = Agent,
  Behaviour = case Energy of
    N when N == 0 -> death;
    _ when rand() > probability -> migration;
    N when N > threshold() -> reproduction;
    _ -> fight
  end,
  {Behaviour, Island}.
```

In the following sections, we describe the step we took in order to parallelise an EMAS using the skel library.

### 3.1 Porting

In an EMAS, there is no spatial aspect in the computation. However, multiple environments can be used, with agent migrations between them, as in the classical island model. Then, separate environments are a trivial way to decompose and parallelize the meta-population.

More fine grained parallelism can be achieved within a single population, though. Agent interactions within an EMAS are defined for pairs of agents. So as soon as the population is divided into pairs (usually randomly), the meeting of each such pair can be computed independently and thus in parallel.

In practise, meetings in an EMAS are numerous but short. Therefore, we need to choose an optimal granularity of parallelism by analysing the cost of the meetings and comparing it to the cost of introducing parallelism.

The problem is, agents may behave differently during meetings, depending on their own state and the state of the other agent, which are both stochastic. Therefore, it may be difficult to estimate the average time cost of a meeting, which may also vary at different phases of the computation.

In order to overcome this, we can restrict meetings so that agents only meet with other agents with the same behaviour - fighting agents can only meet with each other, so do reproducing agents. Then, the cost of any type of meeting, such as fights, reproduction, etc., can be more accurately estimated. The granularity of parallelism can be then tuned appropriately for every type of meeting.

To sum up, the meta-population of agents can be decomposed functionally into
meetings of agents from the same environment and with the same behaviour. Then, these meetings can be executed in parallel with some optimal granularity.

**Sequential implementation**  A sequential implementation of a simple EMAS is shown in Listing 3.2. The population is split between groups of agents having the same behaviour on the same island. These groups are then further partitioned into pairs of agents. Every such partition is updated by calling the logic corresponding to the appropriate behaviour.

Listing 3.2: A sequential functional implementation of a simple evolutionary multi-agent system

```prolog
step(0, Agents, 0) -> Agents;
step(StepsRemaining, Env) ->
    Partitions = split(Env),
    UpdatedParts = [ process(P) || P <- Partitions],
    UpdatedEnv = lists:flatten(UpdatedParts),
    step(StepsRemaining - 1, UpdatedEnv).

split(Env) ->
    MapBehaviour = fun(A) ->
        {behaviour(A), A} end,
    ReduceBehaviour = fun({K,V}, D) ->
        dict:append(K, V, D) end,
    dict:to_list(lists:foldl(ReduceBehaviour, dict:new(), lists:map(MapBehaviour, Env))).

process({{Behaviour, Island}, Agents}) ->
    case Behaviour of
        death -> [];
        migration -> doMigrate(Agents, Island);
        reproduction -> doReproduce(Agents);
        fight -> doFight(Agents)
    end.
```

**Parallel implementation**  There are two parts in the above algorithm which are best candidates for introducing parallelism - the partitioning of the population and the updates of the independent partitions. Listing 3.3 shows how we refactored the initial algorithm by introducing skel.
Listing 3.3: A refactored implementation of a simple evolutionary multi-agent system using skel

\[
\begin{align*}
\text{step}(0, \text{Agents}, 0) & \rightarrow \text{Agents}; \\
\text{step}(\text{StepsRemaining}, \text{Env}) & \rightarrow \\
\quad \text{MapBehaviour} = \text{fun}(X) \rightarrow \\
\quad \quad \text{dict:from_list}([\{\text{behaviour}(X), [X]\}]) & \text{ end}, \\
\quad \text{Concat} = \text{fun}(_K, A, B) \rightarrow A \mathbin{\ast\ast} B & \text{ end}, \\
\quad \text{ReduceBehaviour} = \text{fun}(X, Y) \rightarrow \\
\quad \quad \text{dict:merge}((\text{Concat}, X), Y) & \text{ end}, \\
\quad \text{Id} = \text{fun}(X) \rightarrow X & \text{ end}, \\
\quad \text{Split} = \text{fun}([[B, I], \text{Agents}) \rightarrow \\
\quad \quad \text{lists:map}((\text{fun}(X) \rightarrow [[B, I], X]) & \text{ end}, \\
\quad \quad \text{part}(B, \text{Agents})) & \text{ end}, \\
\quad \text{Process} = \text{fun} \text{process}/1, \\
\quad \text{Workflow} = \\
\quad \quad \text{[[map, [MapBehaviour]],} \\
\quad \quad \quad \{\text{reduce}, \text{ReduceBehaviour}, \text{Id}\}, \\
\quad \quad \quad \text{fun \text{dict:to_list}/1,} \\
\quad \quad \quad \{\text{map, [Split]}\}, \\
\quad \quad \quad \text{fun \text{lists:flatten}/1,} \\
\quad \quad \quad \{\text{map, [Process]}, \\
\quad \quad \quad \text{fun \text{lists:flatten}/1],} \\
\quad \quad \text{UpdatedEnv = skel:do(Workflow, Env),} \\
\quad \quad \text{step(StepsRemaining - 1, UpdatedEnv).}
\end{align*}
\]

The refactoring of the partitioning phase required to change a fold left operation into a more general fold one, which could then be replaced by a reduce skeleton.

The updates of the independent partitions were the easiest to change, as we simply refactored a list comprehension into a map skeleton. However, as the workflow needs to be constructed at compile time, we cannot

### 3.2 Discussion and conclusion

The challenge with a computation where agents may have different behaviour is to ensure that the work in the workflow is well-balanced and that the parallelism has an appropriate granularity. It is not possible to use nested map patterns with different numbers of workers, as the workflow is constructed statically and does not differentiate the data passing through.

Instead, it is the data which needs to be partitioned into chunks requiring a similar amount of work with big enough granularity. In this case, we group together agents with similar behaviour, and we split these groups into smaller chunks ac-
cording to the computational cost of every behaviour.

In other words, we are performing an explicit divide and conquer high level pattern, by first decomposing the population and then mapping every partition.

The two mapping phases (Split and Process) could be replaced by a cluster skeleton with an appropriate decomposer function. However, the decomposer function in the cluster skeleton is sequential. Therefore, it is more efficient to use a workflow item instead of a callback.

To sum up, it wasn’t very hard to refactor the evolutionary multi-agent system in order to introduce parallelism. However, it is not trivial to choose the optimal parameters both in terms of the number of workers in the workflow and in the size of the agent environment’s partitions. Future work includes extensive profiling of the different behaviour in order to determine the optimal values of these parameters.
4. Multi-Agent Systems in Simulation

The agent model is well suited in various kinds of simulations, especially when we consider multiple individuals performing independent tasks and in systems with emergent behaviour. Therefore, the agents’ autonomy [30] may be used efficiently in expressing the behaviour of many beings, making easier the process of designing, implementing and executing such simulations.

A first example of application is the Standing Ovation Problem (also known as Audience Problem [22]). It is a popular simulation problem with a wide range of variants many application areas. This kind of simulation can be seen as a basic model for many social or biological models, such as crowd control or disease propagation [6]. This model is especially useful in complex adaptive social systems, where it helps understand the behaviours and actions taken under the influence of the situation and under the pressure of peers.

A basic SOP can be defined as follow: At the end of a spectacle a certain part of the audience stands up and starts clapping. How will the remaining audience react - will enthusiasm spread or die out?

This kind of simulation can be modelled as a cellular automata, but it can also be interpreted as a simple multi-agent simulation. We consider such an application, where agents represent the audience and are each assigned a seat. The agents observe their neighbours and decide how to act accordingly. We are interested in the collective behaviour which emerges in the system.

A more complicated example of application is the simulation of the lifecycle of Foraminifera. Foraminifera are single-celled eukaryotes that occupy marine benthic and pelagic zones. They have an extraordinary fossil record since the Cambrian (about 540 million years ago). Their cells produce organic and/or mineral shells which are easily fossilizable, leaving perfect signals of palaeoenvironmental conditions “frozen” in time and space [1]. During reproduction foraminifera leave their shells, which are accumulated on the sea floor, as a part sedimentary record. This makes them an ideal model organism and microfossil often used for paleo reconstructions and testing general evolutionary hypotheses [7, 18, 25].

Since computer simulations can be helpful in understanding foraminiferal evolution and evolutionary processes per se, our goal is to construct a model reflecting
growth processes of foraminiferal individuals (ontogenesis) within a virtual environment where they could move, eat, grow and reproduce following rules that resemble their physiology and behaviour. The ultimate aim is to test micro- and macroevolutionary rules and interactions controlling the overall complex system which acts at various spatiotemporal scales necessary for the emergence of complexity of life [12].

There is a number of software platforms which address the requirements of such simulations (e.g. Swarm/RePast [13, 23, 24], MASON [20] and others). However, only a few of them provide support for large-scale simulations (e.g. RepastHPC [4] and Framsticks [13]). Yet, they do so mainly in terms of distribution and are not able to efficiently use multicore heterogeneous hardware.

Agent-based simulations put even bigger constraints on parallelism than generic computations, because of spatial requirements. Again, a functional approach opens new possibility in the design and execution of such agent simulations. We assume the multi-agent simulation is defined in terms of the high-level functions described in deliverable D6.7. What follows is a description of the low-level implementation using Skel skeletons.

4.1 Porting

In a spatial simulation, the range of interactions, observation or movement of an agent will always be restricted to some neighbourhood of size $R$. Let us divide the simulation space into areas of size not smaller then $2R$ (Fig. 4.1). The agents from area $A$ cannot directly interact with agents from area $C$ within one time step.

![Figure 4.1: If the range of interaction is $R$, agents from area $A$ cannot directly interact with agents from area $C$ in the same time step.](image)

They may however interact indirectly through the agents in area $B$: an agent in $A$ may update an agent in $B$, which may cause the latter to update an agent in $C$. Whether that happens depends on the order in which we update the agents.

In general, that iteration order affects the result of such a multi-agent simulation. In practice, agents will be randomly shuffled so that the iteration order
becomes stochastically insignificant.

However, some iteration orders are more beneficial for us than others. Let us suppose that all agents from $B$ are processed first. We can then reason that agents from $A$ and $C$ can no longer interact, either directly nor indirectly. Those two areas could then be processed independently and thus in parallel.

**Graph colouring** We can generalize this reasoning to more complex partitions of the simulation space. Let us define the graph $G = \{V, E\}$ so that $V$ is the set of areas, corresponding to some partitioning of the simulation space. We choose those areas to have a diameter no smaller than $2R$. The relation $E$ contains those pairs of areas for which the distance between them is smaller than $2R$.

Following such a definition, agents located in a particular node of the graph can only interact directly with agents located on neighbour nodes. Moreover, for any proper vertex colouring we choose, agents on nodes with the same color cannot directly interact with each other.

![Figure 4.2: Several types of partitioning with different chromatic numbers: stripes, quad-trees, kd-trees.](image)

Therefore, the simulation can be computed in parallel for every area with the same color, one color at a time. Many types of partitioning may considered, such as quad-trees or kd-trees (Fig. 4.2). In general, the partitioning algorithm should:

- have a low computational complexity (as it may need to be computed dynamically at every step)
- provide partitions with similar numbers of agents (to load balance the workers)
- provide partitions with a minimal diameter of $2R$
- have a low chromatic number (to shorten the sequential critical path)

We are investigating several partitioning algorithms to optimize the above criteria. In the current version of the use case, we have used the simplest to implement and analyse, i.e. stripes.

Finally, although we restrict the space of possible iteration orders, we can preserve the stochastic properties of the algorithm by randomly iterating within the areas and by randomly permuting the colors of the areas.
**Sequential implementation**  We started by writing a sequential implementation. A generic functional version of such a multi-agent simulation is shown in listing 4.4. The types of the Agent and Env arguments are problem dependent, so does the implementation of the update function.

Listing 4.4: A generic functional implementation of a multi-agent simulation

```haskell
step(0, Env) -> Env;
step(StepsRemaining, [], Env) ->
    UpdatedEnv = updateAgents(getAgents(Env), Env),
    step(StepsRemaining - 1, UpdatedEnv).

updateAgents([], Env) -> Env;
updateAgents([Agent | Agents], Env) ->
    UpdatedEnv = update(Agent, Env),
    updateAgents(Agents, UpdatedEnv).
```

The Env arguments represent an immutable data structure with all the information about the agents, their state and environment. The update function performs an action for the given agent and returns the resulting new environment.

**Partitioned implementation**  We had to refactor this implementation to include some partitioning of the environment, as shown in listing 4.5. We in order to do so, we introduce three new functions.

The split function partitions the environment. The resulting partitions are processed by the colour function so that independent partitions are grouped together. The merge function applies the changes from a partition onto the environment. The updated partitions are thus folded over the initial environment with the merge function into a single new environment.

Listing 4.5: The environment is partitioned and independent partitions are processed together then merged.

```haskell
step(0, Env) -> Env;
step(StepsRemaining, Env) ->
    Partitions = split(Env),
    ColoredPartitions = colour(Partitions),
    UpdatedEnv = process(ColoredPartitions, Env),
    step(StepsRemaining - 1, UpdatedEnv).

process([], Env) -> Env;
process([Color | OtherColors], Env) ->
    UpdatedParts = [updateAgents(getAgents(Part), Env)
    || Part <- Color],
    UpdatedEnv = lists:foldl(fun merge/2, Env, UpdatedParts),
```
process(OtherColors, UpdatedEnv).

If the \textit{split} and \textit{colour} function follow the consideration described at the beginning of this section, we are guaranteed that there will be no conflicts when merging the changes to the environment from the different partitions.

This triplet of functions, \textit{split}, \textit{color} and \textit{merge}, are domain-dependent. The same implementation can be reused across different spatial simulations, as long as similar data structures are used or a common API is followed.

Currently, we only provide a trivial implementation for spatial simulations: Given $N$ workers, the environment is divided into at most $2N$ vertical stripes of size no smaller than $2R$. Stripes are then coloured modulo 2.

\textbf{Parallel implementation} In order to achieve a parallel implementation, we refactored the \textit{process} method to use a map and a reduce skeleton, as shown in listing 4.6.

The actions of retrieving the agents from a given partition and executing their actions is made parallel with the \textit{map} skeleton. The results of the updates are returned as differences with regard to the initial environment.

These updates can then be combined. As the updates to individual partitions cannot conflict with each other, the \textit{merge} function can be considered as an associative operator. Therefore, it can be safely made parallel with the \textit{reduce} skeleton.

Finally, the merged updates are applied on the initial environment. The simulation can then continue with the partitions of the next color, until all partitions are computed and the global time step is done.

\begin{verbatim}
Listing 4.6: A skel-based implementation of the process method.
process([], Env) -> Env;
process([Color | OtherColors], Env) ->
  GetAgents = {seq, fun getAgents/1},
  UpdateAgents = {seq, fun(Agents) -> updateAgents(Agents, Env) end},
  Merge = fun(Part1, Part2) -> merge(Part1, Part2) end,
  FinalMerge = fun(MergedParts) -> merge(Env, MergedParts),

  UpdatedEnv = skel:do([map, [GetAgents, UpdateAgents], N],
                        {reduce, Merge, fun util:id/1],
                        {seq, FinalMerge},
                        Color).
  process(OtherColors, UpdatedEnv).
\end{verbatim}
4.2 Discussion and conclusion

The approach described above is similar to the Divide and Conquer high level pattern. However, one major difference is that not all partitions are independent and that they need to be computed in a specific order. Therefore, the custom Conquer phase need to iterate other partitions colours.

The effort to make the simulation was two-fold. The first step, which took two weeks, was to refactor the algorithm in order to include the partitioning of the simulation’s environment. The second step, to introduce skel and actual parallelism, was pretty straightforward and took only two days.

However, additional work still needs to be done. Preliminary performance tests indicate that the simulation scales well with the number of agents, but with a big constant time cost which makes it slower than a sequential version (which scales less well).

In the current implementation, a new workflow is constructed at every step for every colour. This implicates that the overhead of creating a skel workflow is multiplied by the number of steps and colours. It would be more efficient to create a single workflow per step or even a single workflow for the whole simulation.

We now assume that all nodes of a given colour need to be computed before we move on to the next colour. In fact, for a given node of colour \( i \) to be computed, only its neighbours of colour \( i-1 \) need to have been computed. We could take advantage of that property and further refactor the workflow to include some pipelining between colours.

Another improvement to be made is to use a more sophisticated partitioning algorithm, such as a quadtree or kd-tree, in order to achieve partitions with better properties.

Finally, we need to meta-optimize the number of workers allocated at each stage of the workplace.
5. Conclusion

The deliverable gives an insight into the details of porting two general application areas, namely agent-based optimization and simulation to the ParaPhrase infrastructure using the skel library. Besides technical description, general considerations are presented and specific computing and simulation systems are discussed.

In this deliverable, we have shown that the key to parallelize a multi-agent system is to find an efficient way to decompose it. As we have seen, it is easier to do so when the system has a functional and share-nothing design.

We described two different approaches corresponding to two areas of applications: a multi-agent computation is best decomposed functionally so that agents with different behaviour are computed independently. A multi-agent simulation is best decomposed spatially, so that agents which cannot directly interact are computed independently. When agents can also interact indirectly, an iterated approach based on graph colouring can also be efficiently parallelized.

On a practical side, it was relatively easy to refactor the sequential implementations. As they were already based on functional operators like fold and map, these could easily be replaced by the skeletons corresponding to the appropriate patterns.

On the other hand, there is one caveat with using skel. Skel skeletons hide from the user that Erlang processes are used under the hood. This implementation detail is transparent in most aspects but one - most data types and data structures need to be copied between process memories, even if they are immutable. The only exception is binary data on the same node.

This property makes it more difficult to scale generic computations and simulations, which do not inherently operate on binary data, like image processing use cases. Of course, many of the information concerning the agents can be encoded in binary form and decoded when needed, but that also comes at a cost. In consequence, for small scales of simulations it is more efficient to just use traditional data structures in the workflow, but for bigger scales a binary representation may be required. Preliminary performance tests indicate that the communication cost of big, non-binary data can be significant in long workflows.

With regard to future work, we now need to extensively benchmark our applications to fine-tune the parameters of the skel workflows and the sizes of agent environment partitions. We will also work on improving the partitioning algorithm used in spatial simulations. Finally, we will investigate how agent behaviour and interactions can be offloaded to GPGPUs.
Bibliography


