

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

Pool Evolution: A Parallel Pattern for Evolutionary and Symbolic Computing

This is the author's manuscript

Original Citation:

Availability:

This version is available <http://hdl.handle.net/2318/1522392> since 2016-11-19T09:24:28Z

Published version:

DOI:10.1007/s10766-015-0358-5

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

Pool evolution: a parallel pattern for evolutionary and symbolic computing

Marco Aldinucci · Sonia Campa · Marco Danelutto · Peter Kilpatrick · Massimo Torquati

Published by Springer on 18 Mar 2015

Abstract We introduce a new parallel pattern derived from a specific application domain and show how it turns out to have application beyond its domain of origin.

The *pool evolution* pattern models the parallel evolution of a population subject to mutations and evolving in such a way that a given fitness function is optimized. The pattern has been demonstrated to be suitable for capturing and modeling the parallel patterns underpinning various evolutionary algorithms, as well as other parallel patterns typical of symbolic computation.

In this paper we introduce the pattern, we discuss its implementation on modern multi/many core architectures and finally present experimental results obtained with FastFlow and Erlang implementations to assess its feasibility and scalability.

Keywords Parallel design patterns · algorithmic skeletons · multi/many core architectures · evolutionary computing · FastFlow

1 Introduction

Design patterns were originally proposed as a tool to support the development of sequential object oriented software [12] and have proven to be a very ef-

M. Aldinucci
Dept. of Computer Science, Univ. of Torino
E-mail: aldinuc@di.unito.it

S. Campa, M. Danelutto, M. Torquati
Dept. of Computer Science, Univ. of Pisa
E-mail: {campa,marcod,torquati}@di.unipi.it

P. Kilpatrick
Dept. of Computer Science, Queen's Univ. Belfast
E-mail: p.kilpatrick@qub.ac.uk

fective programming methodology, greatly improving the time-to-deploy and maintainability of complex applications.

Later, the pattern concept migrated to the parallel programming world where it has been seen to be a realistic instrument with which to attack the programmability issue of modern parallel architectures [4]. According to Mattson, Sanders and Massingill [17], *... a design pattern describes a good solution to a recurring problem in a particular context. The pattern follows a prescribed format that includes the pattern name, a description of the context, the forces (goals and constraints), and the solution.* In their book, the authors identify a number of patterns, organized in four *design spaces*: the “finding concurrency”, “algorithm structure”, “support structure” and “implementation mechanisms” spaces. Each of the spaces includes several patterns aimed at capturing and modelling parallelism exploitation features typical of the abstraction level represented in the design space. Proceeding from the top down, the finding concurrency space hosts patterns identifying the available concurrency; the algorithm structure space hosts patterns modelling different kinds of algorithm; and the support structure and implementation mechanisms spaces provide patterns modeling the typical implementation mechanisms (high and low level) used to construct parallel computations.

Design patterns, as well as parallel design patterns, are described by suitable sections of text, and programmers wishing to use the patterns to implement their applications can follow the pattern *recipe* but they must write all the code needed to implement the patterns on their own and, very often, from scratch.

In the ParaPhrase project [19], we devised a slightly different parallel application development methodology. Parallel design patterns are used to define the correct parallel structure of an application. Then, the implementation of the “patterned” application uses a library of composable algorithmic skeletons, as suggested by M. Cole. Cole, in his “skeleton manifesto” [7], observes how patterns may be used to abstract computation and interaction structures in parallel applications: *Skeletal programming proposes that such patterns be abstracted and provided as a programmer’s toolkit, with specifications which transcend architectural variations but implementations which recognize these to enhance performance.* Thus we use the algorithmic skeletons provided by the FastFlow programming framework [3, 8, 11], and by the `skel` Erlang skeleton library [5, 10], to implement, alone or in suitable composition, the parallel patterns deemed appropriate by the application programmer.

The parallel design patterns identified by Mattson et al. are quite generic. They include general patterns such as divide&conquer (algorithm space) and master/slave (implementation structure space), to mention just two well-known pattern examples. Parallel patterns identified by different authors [18] are also generic/general purpose. Because of the widespread applicability of such general patterns, it is worthwhile investing effort in developing efficient implementations for a range of languages/architectures. However, the application programmer needs to know how to assemble them to realise an application. In contrast, application programmers tend to identify as “patterns” computation

structures very close to their application domain. For example, programmers of evolutionary algorithms readily recognize as a pattern the (parallel) *computation of the possible evolution of a gene population*, that is the application of some “evolution function” to all the genes of a given population. Numerical application programmers, however, will recognize the same parallel computation schema as a *map* pattern, that is, a pattern processing an input collection—a vector or a matrix—by applying to each element of the collection x_i the same function f and returning the collection of the $f(x_i)$ s.

In this paper we introduce a new parallel pattern, the *Pool Evolution Pattern*, which originated in a domain-specific context (evolutionary computing) but has been demonstrated to be more general and useful in a number of different applications, both from the evolutionary computing and the symbolic computing domains. Thus, in generality, it sits between patterns such as master/slave, etc. which are general-purpose, and domain-specific patterns such as the Orbit pattern (see Table 2) which have much narrower applicability. The intent is to provide application programmers from the named domains with a pattern which is somewhat general, yet is tuned to their way of thinking.

The main contribution of the paper can be summarized as follows:

- Definition of a new parallel pattern—*pool evolution*—capturing the iterative evolution of a population. The pattern logically belongs to the “algorithm structure” design space, according to the layering of Mattson et al.
- Identification of a list of applications, from different domains, whose parallel behaviour may be perfectly modelled via the new pattern.
- Implementation of the pattern as a new algorithmic skeleton, such that the application programmer may implement a pool evolution patterned application by just providing the functional parameters (business logic code of the evolutionary algorithm) to a pool evolution skeleton object. Both **FastFlow** and Erlang implementations have been developed.
- Experimental results assessing the scalability and efficiency of both the **FastFlow** and Erlang skeletons implementing the new parallel pattern.

The remainder of this paper is organized as follows: Section 2 introduces the new pattern and lists different applications whose parallel structure may be suitably modeled using the pattern. Section 3 outlines possible implementation strategies for an algorithmic skeleton implementing the pattern and then outlines the **FastFlow** and Erlang skeleton implementations. Section 4 presents experimental validation of the **FastFlow** and Erlang implementations of the new pattern. Finally, Section 5 outlines related work and Section 6 draws conclusions.

2 Pool evolution pattern

In this section we first describe the new parallel pattern and then provide different patterns, from the same or from other application domains, that may be implemented as specialisations of the new pattern.

Name	Pool evolution pattern
Problem	<p>The pattern models the evolution of a population. In the pattern, a “candidate selection” function (s) selects a subset of objects belonging to an unstructured object pool (P). The selected objects are processed by means of an “evolution” function (e). The evolution function may produce any number of new/modified objects out of the input one. The set of objects computed by the evolution function on the selected object are filtered through a “filter” function (f) and eventually inserted into the object pool. At any insertion/extraction into/from the object pool a “termination” function (t) is evaluated on the object pool to determine whether the evolution process should be stopped or continued for further iterations. A pool evolution pattern therefore computes P as result of the following algorithm:</p> <pre> 1: while $\text{not}(t(P))$ do 2: $N = e(s(P))$ 3: $P = P \cup f(N, P)$ 4: end while </pre>
Forces	<p>The selection function may be implemented in a data parallel way, by partitioning the object pool and applying selection in parallel on the different partitions. The evolution function step is clearly an embarrassingly parallel one, as is the filter process. The termination process may be implemented as a reduce pattern, but in some cases (e.g. when the “counting” termination function is used) it may be implemented as a plain sequential process. Candidate partitioning in small blocks may improve load balancing. Job stealing may be used to improve load balancing in the last stage of the computation. Mapreduce implementation of the selection, evolution and filtering stages may be considered to improve parallelism exploitation when large populations are evolved.</p>

Table 1 Pool evolution pattern

2.1 Pattern definition

By means of Table 1 we provide a concise pool evolution pattern definition, in the style of Mattson et al. We deliberately omit the solution and example sections, as pattern implementation will be discussed thoroughly in Section 3 and sample usage of the pattern will be discussed later in this section and in Section 4. From an algorithmic skeleton perspective, the pool evolution pattern may be described in terms of its functional (what it computes) semantics and parallel (how the results are computed in parallel) semantics. The functional semantics is that defined by the **while** loop in the *Problem* part of Table 1. We now consider the parallel semantics in more detail.

In principle, the computation of a pool evolution pattern is an iterative process. Iteration i builds on the results given by iteration $i - 1$ and so iteration computations must be serialized. Also, each iteration is built of different stages (selection, evolution, filtering) which should also be executed sequentially, as each stage takes as input the output of the previous stage. However, the different stages of the single iteration may be computed in parallel, as suggested by *Forces* in Table 1. In particular:

- The selection process is usually based on some function selecting the “best” individuals as candidates to be submitted to the evolution process. This

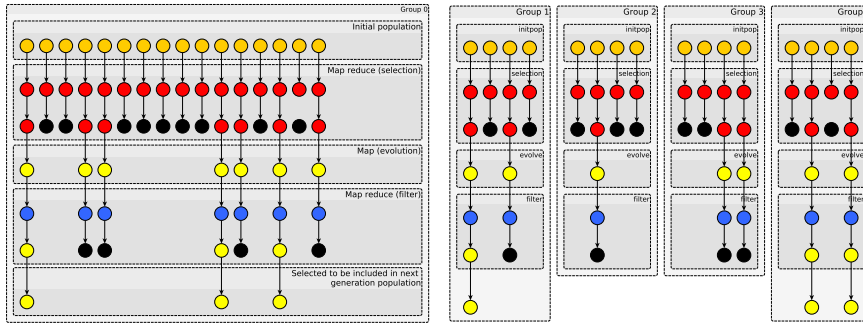


Fig. 1 Alternative implementations (with multipopulation) of the pool evolution pattern: single population (left) vs. multipopulation (right)

obviously makes the selection process a good candidate for application of a mapreduce pattern¹: first map the function estimating how good is an individual, then filter (reduce) the better candidates.

- The evolution process is embarrassingly parallel: each individual (or group of individuals, e.g. in a crossover) may be “evolved” independently of the other selected individuals.
- The filtering process usually evaluates some *fitness* function on the evolved individuals. Finally, the new individuals with the “best” fitness values are selected to be kept in the next generation. In some cases they are added to the current population, retaining in the population the original individuals that led to the evolution of the new ones. More commonly, the new individuals replace the ones originating the evolution. In either case, the filtering process may be another candidate for application of a mapreduce pattern, with the fitness function being applied in the map and the filtering being applied in the reduce.

Assuming availability of the usual map and mapreduce patterns in the implementation structure design space², the complete iteration process may be structured as follows:

```
pipe(mapreduce( $f_{sel}$ ,  $\oplus_{max_k}$ ), map( $f_{evol}$ ), mapreduce( $f_{fitness}$ ,  $\oplus_{max_k}$ ))
```

$f_{fitness}$ being the function computing the fitness of an individual and \oplus_{max_k} the operator used to compute the “best” individual according to its fitness.

Fig. 1 (left) outlines the kind of computation performed, with circles representing individuals in the population transformed by the different map and mapreduce phases. This raises the opportunity to group differently the computations eventually leading to the new population by:

¹ for simplicity we use the term “mapreduce” to denote the optimized composition of a map and a reduce pattern; note this is different from the “Google mapreduce”, the latter involving multiple, parallel reduce steps after the map phase, each relative to a different *key*

² we assume here availability of the corresponding algorithmic skeletons

1. splitting the population into disjoint groups G_1, \dots, G_g ;
2. within each group selecting individuals, evolving them, computing fitness and filtering new individuals; and
3. putting back the selected individuals from the different groups into the population and evaluating the termination condition;

as sketched in Fig. 1 (right). This makes sense from the pure parallelism exploitation viewpoint; however it also slightly changes the evolution process semantics (single population vs. multiple population algorithms [2]), as the reduce steps will probably lead to different results, given that the population is split into groups.

In the definition of the pattern, this is not relevant, as the *Solution* text only suggests possible solutions but it does not impose any parallel implementation schema. When moving to the algorithmic skeleton implementation of the pattern, this possibility should be taken into account and the user (application programmer) may be conveniently provided with a boolean skeleton parameter enabling/disabling population decomposition. The boolean parameter will provide the application programmer with the possibility (and duty) to evaluate the trade-off between the parallelism exploited and the kind of evolutionary algorithm computed.

2.2 Pattern usage examples

Having defined the pool evolution pattern, we now describe a) more patterns that may be implemented in terms of the pool evolution pattern, and b) applications from different domains that may be implemented using the pattern. The goal is to assess whether the new pool evolution pattern is, in fact, a more general pattern, not specifically bound to the particular domain where it originated. If this is the case then it is of course desirable that efficient implementations of the pattern can be obtained via specific (compositions of) algorithmic skeletons as this broadens the reach of the pattern.

2.2.1 More patterns

Table 2 includes a number of patterns drawn from two distinct application domains: evolutionary computing and symbolic computing. These patterns may be “reduced” to the pool evolution pattern, that is, a pool evolution pattern instance exists computing the same (parallel) computation captured by each of these more specific patterns.

The *orbit* pattern comes from the symbolic computing community [14, 22, 23]. It computes the transitive closure of a set through a set of generators. From the pool evolution perspective, the selection function is the identity (all the items in the current set are selected), the evolution function is a *map* pattern over the generator set items (for each individual p_i compute the set $gen_1(p_i), gen_2(p_i), \dots, gen_g(p_i)$) and finally the filter function checks that the new individuals do not already belong to the population.

Symbolic computing domain	
Name	Orbit
Problem	The orbit pattern comes from the <i>symbolic computing</i> domain and models the iterative construction of a set of items starting from an initial set (S) of items using <i>generator functions</i> from a generator function set ($G = \{g_1, \dots, g_k\}$). At each stage of the iterative process, the set of generators are applied to all the elements in the current set. The resulting set of new items is then added to the original set taking care to avoid duplicates. Therefore the orbit pattern computes the transitive closure of a set according to a set of generator functions.
Evolutionary computing domain	
Name	Genetic algorithm pattern
Problem	The genetic algorithm pattern describes an iterative behaviour in which, at each iteration step, a set of items (the individuals belonging to a population) evolves. The size of the population could change or could be statically defined. How the items change depends on the <i>genetic operator</i> that the pattern will apply (mutation and crossover, for instance), and is thus a matter of application specificity.
Name	Global single population pattern
Problem	The Global single population genetic pattern is a domain-specific instance of the Genetic Algorithm pattern (see above) where the evolution is a process involving the whole population in each generation. In fact, the population is seen as a single entity over which individuals evolve on the basis of a set of genetic operators. The population size tends to be statically defined, and so does not change as the computation proceeds. The result of the global single population genetic pattern may be defined in terms of the algorithm computed by the pool evolution pattern algorithm.
Name	Multiagent system pattern
Problem	The multiagent system pattern models the evolution of a multiagent system. A multiagent system can be described as a set of autonomous, independent and decentralized agents which represent, through their activities, the evolution of a - virtual, discrete or continuous - environment. Agents can be provided with different degrees of "intelligence", depending on the application domain and the problem to solve but, generally speaking, they act all independently, they could go through significant inactivity periods, they do not necessarily share information, and their behaviour is generally highly influenced by the environment. A multiagent system can be seen as a set of agents A_1, \dots, A_n , each executing one or more jobs j_1, \dots, j_m . Jobs could be provided with a weight and agents can have a limit of workload assigned a_1, \dots, a_n . A pattern for a multiagent system is that which assigns jobs to each agent (the so called Job Assignment Problem) so as to obtain the maximum utility in the minimum overall completion time, i.e. to maximise $\sum_{a \in \{A_1, \dots, A_n\}} M_i \frac{1}{t_i} U_i$ where M_i is the load of each job, t_i is the execution time of the i^{th} job imposed by agent a , and u_i is the utility gained from the job being executed by agent a .
Name	Concurrent memetization pattern
Problem	This pattern is also used in evolutionary computation in which iterative progress processing, such as growth or development in a population, is performed. With respect to other patterns in the family of genetic algorithm patterns, here the population is selected, during the iterative process, using suitable search operators in order to achieve the desired goal. The pattern involves continuous optimisation and combinatorial optimisation phases. It may be useful for implementing Lamarckian or Baldwinian memetic variation operators [6]. The procedure starts with a certain individual i_{init} and a set of mutation operators (M) available. Then, according to the parameters, a series of mutations $m \in M$ and evaluations f of new individuals are performed in parallel. The best obtained solution i becomes a new starting point for another phase of the memetization. The best individual after the assumed number of phases is returned.

Table 2 Specific domain patterns suitable for reduction to pool evolution pattern

The other patterns all come from the evolutionary computing community. The *genetic algorithm pattern* maps one-to-one onto the pool evolution pattern, as does the *Global single population pattern*. In fact, they can be understood as the pattern(s) that generated—through generalisation and abstraction—the pool evolution pattern.

The *Multiagent system pattern* is somehow more interesting. In terms of the pool evolution pattern, the selection function selects all agents that have an event to process in their input event queue (a message, a synchronisation request, etc.); the evolution function updates the agent’s internal state based on the input event accepted; and the filter function is the identity function (all transformed individuals/agents are put back into the original population replacing their previous (state) instances). The “evolution” of an agent, however, may generate events directed to other agents. These events should be directed to the correct agent queues during the filtering (update) function, which makes the filtering / termination test slightly more complex. The *concurrent memetization pattern* may be reduced to the pool evolution pattern with a process similar to that used to reduce the orbit pattern to the pool evolution one.

2.2.2 More applications

As far as applications are concerned, we consider a few representative applications whose parallel behaviour may be readily modeled by the pool pattern.

Strings of a given length generated by a grammar This is clearly an instance of the orbit pattern (and thus, transitively, of the pool evolution pattern). The productions of the grammar are used as generators and the filtering function (with no fitness function included) simply filters those items a) not already belonging to the current population and b) not longer than the given length. Termination is determined by an empty set to be added to the population after an iteration.

Brute force sudoku solver This is a plain pool evolution pattern instance. The population is initially a partially filled board. The evolution function generates a board with possible assignments of an empty cell from the original board and the filter function sends back to the population those boards adhering to the sudoku rules. A more efficient variant is that where the evolution function picks up an empty cell and generates only those configurations filling the cell with legal values. In this case the filter function is the identity function, and the filtering activity itself is moved from filter (sequential execution in the implementation schema P1 in Section 3 below) to evolution (computed in parallel).

Function minimum in an interval This is a plain genetic pattern with population made of random points in the interval, a fitness function computing the function on the given point of the interval, and the evolution pattern(s) generating new random points, or new points close to the “best” (i.e. minimum) elements in the population.

Finding a function approximating a given set of $\langle \text{point}, \text{value} \rangle$ pairs A population of random functions is evolved by selecting those giving the best approximation of the function value on the known points and applying different random mutations, including a kind of crossover, to obtain new functions. The fitness function measures the distance of the function from the target points. Here the selection function requires evaluation of the distance of the computed points from the target values for all the functions in the population, and a reduction to determine the functions with minimal distance, similar to the function used to select the best mutated individuals as candidates to be inserted in the new population.

3 Skeleton implementation

The parallel implementation of the pool evolution pattern may employ parallelism at different levels and with different granularities. We consider three possibilities, namely:

- P1 parallel computation (map pattern) of the *evolution* function over the selected individuals, with sequential computation of the other phases (selection, filtering and termination).
- P2 parallel computation of all phases (as outlined at the end of Section 2): mapreduce for *selection* and *filter* phases and map for the *evolution* phase.
- P3 split the population into sub-populations and map the whole computation relative to one iteration on the sub-populations, merging the updates after the termination of the sub-computations (map of `filter(evolve())` over sub-partitions, then “reduce” filtered individuals for inclusion in the pool population).

The three alternatives use different grains of parallelism (P1 and P2 process individuals in parallel, while P3 processes partitions of the population) and two of them (P1 and P2), while working at the same granularity, use different extents of parallelism (P1 has a greater serial fraction than P2).

In accordance with the ParaPhrase methodology—which provides both C++/FastFlow and Erlang pattern implementations—we implemented two versions of the pool pattern: a FastFlow [11] version and an Erlang version.

The FastFlow version is built on top of a *task-farm-with-feedback* core skeleton, suitably customized to take into account the features of the pool as implemented according to schema P1 or P2. The Erlang version, instead, is built on top of the `skel` Erlang skeleton library [5, 10]. Different versions of the Erlang

pool evolution pattern use *map* skeleton instances to implement the different parallel parts of the skeleton over the selected population items (P1 or P2).

The Erlang version is very compact and the code corresponds one-to-one to the pseudo code given in Table 1 to describe the pattern’s functional semantics. The code implementing the P1 pool pattern version on top of **skel** is:

```

1 pool(Termination, Selection, Evolution, Filter) ->
2   fun(Set) ->
3     case (Termination(Set)) of
4       true ->
5         Set;
6       false ->
7         {Selected,Rest} = Selection(Set),
8         Evolved = parmap(Evolution,Selected),
9         Filtered = Filter(Evolved),
10        Newset = union(Rest, Filtered),
11        (pool(Termination, Selection, Evolution, Filter))(Newset)
12      end
13    end.

```

where the `parmap` computes (in parallel) the same result as that computed by a `lists:map`.

In the **FastFlow** implementation, by default only the evolution phase is computed in parallel (P1 schema). However, it is also possible to configure the pool implementation to compute the selection and the filtering map-reduce phases in parallel. On the contrary, the termination phase is always computed sequentially in the current implementation. Both the map and map-reduce phases have been implemented using the *ParallelForReduce* high-level pattern [8] already available in the **FastFlow** framework. The *ParallelForReduce* pattern allows efficient parallelisation of parallel loops and parallel loops with reduction variables. It is implemented using the *task-farm-with-feedback* core skeleton of the framework. In the **FastFlow** task with feedback skeleton, an emitter thread schedules tasks (either appearing on an input stream or generated from an in-memory data-structure) to a pool of worker threads. The workers compute the task results and deliver them back to the emitter. The emitter scheduling policy may be customised by the user.

In Fig. 2 are sketched both the concurrency structure of the possible parallel pattern implementing the pool evolution and the concrete implementation skeleton currently implementing the pool pattern in **FastFlow**.

This quite simple but effective parallel structure is provided to the parallel application programmer through an interface similar to that used for the other **FastFlow** high level patterns (see [20]), supporting all the parameters needed to specialize the pool evolution pattern by means of user (application programmer) supplied business code and non-functional parameters.

In particular, the pool evolution pattern interface has been designed as follows:

```

1 template<typename T, typename env_t=char>
2 class poolEvolution : public ff_node {
3 public:
4   /* selection_t : it takes the population and returns a sub-population
5   * evolution_t : it works on the single element
6   * filtering_t : it takes the population produced at the previous step,
7   *               the output of the evolution phase and produces a new population

```

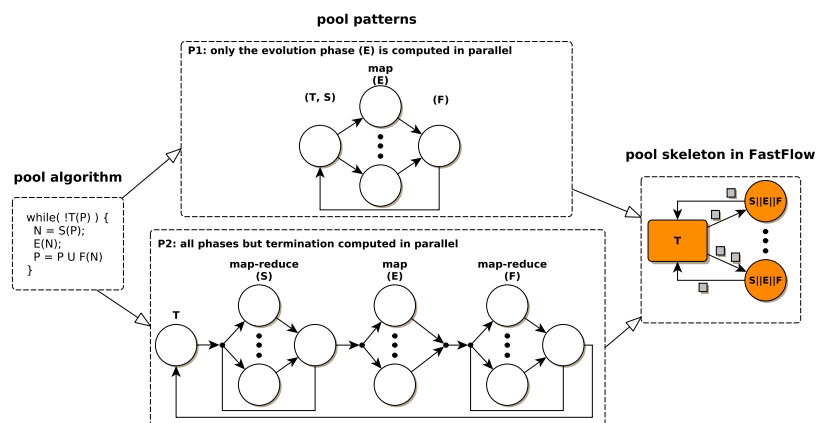


Fig. 2 Algorithm, concurrent activity graphs and the FastFlow parallel skeleton of the pool evolution pattern. Each circle represents a thread, arrows represent communication channels, which are implemented through FastFlow lock-free buffers.

```

8  */
9  typedef void (*selection_t)(ParallelForReduce<T>&,std::vector<T>&,std::vector<T>&,
    env_t&);
10 typedef const T&(*evolution_t)(T&, const env_t&);
11 typedef void (*filtering_t)(ParallelForReduce<T>&,std::vector<T>&,std::vector<T>&,
    env_t&);
12 typedef bool (*termination_t)(const std::vector<T>&,env_t&);
13 protected:
14   env_t env;
15   ParallelForReduce<T> mapreduce;
16   ...
17 public :
18   /* constructor : to be used in non-streaming applications */
19   poolEvolution (size_t maxp,          /* maximum parallelism degree in all phases */
20                 std::vector<T> &pop, /* the initial population */
21                 selection_t sel      /* the selection function */
22                 evolution_t evol,   /* the evolution function */
23                 filtering_t fil,    /* the filter function */
24                 termination_t ter,   /* the termination function */
25                 const env_t &E= env_t()); /* user's environment */
26   /* constructor : to be used in streaming applications */
27   poolEvolution (size_t maxp,          /* maximum parallelism degree in all phases */
28                 selection_t sel      /* the selection function */
29                 evolution_t evol,   /* the evolution function */
30                 filtering_t fil,    /* the filter function */
31                 termination_t term, /* the termination function */
32                 const env_t &E= env_t()); /* user's environment */
33
34   /* changing the parallelism degree of the evolution phase */
35   void setParEvolution(size_t pardegree);
36   const env_t& getEnv() const { return env;}
37   ....
38

```

The pattern has two constructors: one to support standalone execution of the pattern, where execution processes only the input population specified as parameter; and another to support execution of the pattern over population

items appearing on the pattern’s input stream³. The *env_t* type passed as template parameter is the type representing the “external environment” of the pool pattern. It is the type of a data-structure passed in the constructors, and accessed in a read-only way in the evolution phase, which is used to store “global” information between the computation of the four different phases of the pattern.

To exemplify the user’s (application programmer’s) perspective of the pool evolution pattern, we now discuss the kind of code needed to program a simple application exploiting parallelism via the pool evolution pattern in **FastFlow**. In particular, we consider an application modeling a population evolution where:

- each individual of the population is tagged as selected or not-selected by means of some criterion;
- individuals are evaluated in parallel and those exhibiting very good fitness can generate new individuals and/or mutate; and
- the new or mutated individuals are indiscriminately added to the original population.

The following outline code captures this scenario:

```

1 #include <ff/poolEvolution.hpp>
2 const long MAXGENERATION = 100;
3 typedef long Env_t; /* this is my simple environment */
4 class Individual { /* implementation of an individual */
5 ...
6 };
7 /*----- genetic operators ----- */
8 bool termination(const std::vector<Individual> &P, Env_t &E) {
9     return E.numGeneration >= MAXGENERATION;
10 }
11 void selection(ParallelForReduce<Individual> &mapreduce,
12               std::vector<Individual> &P,
13               std::vector<Individual> &newP, Env_t &E) {
14     ... // implementation of selection: P --> newP
15 }
16 const Individual& evolution(Individual &t) {
17     ... // implement the evolution changes in t
18 }
19 void filter(ParallelForReduce<Individual> &mapreduce,
20            std::vector<Individual> &P,
21            std::vector<Individual> &newP, Env_t &E) {
22     ... // filter individuals (P) to be added to current pop
23     newP += P;
24 }
25 /*----- end of genetic specific material -----*/
26
27 int main(int argc, char *argv[]) {
28     std::vector<Individual> initialP = ...;
29     Env_t num_generations=0; /* my simple environment */
30     poolEvolution<Individual, Env_t> /* instantiate the pattern pardegree=48 */
31     pool(48, initialP, selection, evolution, filter, termination,
32         num_generation);
33     if (pool.run_and_wait_end()<0) /* then run it and wait completion */
34         error("poolEvolution fails to run\n");
35 }

```

³ in **FastFlow** any pattern has an input and an output stream to support pattern composition; or it is executed just once, in which case input data is passed via pattern parameters

4 Experimental validation

To evaluate the implementation of the pool evolution pattern we performed a set of experiments using simple synthetic benchmarks and three applications: as described in Section 2.2, the first searches for the best function that approximates a given set of $\langle \text{point, value} \rangle$ pairs; the second computes the approximation of the minimum of a given function in a given interval ($\min f(x) \ x \in [a, b]$); and the third implements a sudoku solver. The first application provides an example of how the different phases of the pattern could be parallelized; the second experiment, focusing on the parallelisation of the evolution phase only, shows how the pattern maintains good performance even when just one phase is parallel and/or some phases are sequential, but computationally irrelevant. The third demonstrates application of the pool pattern in symbolic computation domains. Code used for the experiments is (in part) available from the `FastFlow` svn [11] and `ParaPhrase` web site [19].

As target architecture for the experiments we use a dual-socket NUMA Intel multi-core Xeon E5-2695 Ivy Bridge micro-architecture running at 2.40GHz featuring 24 cores (12+12) each with 2-way Hyperthreading. Each core has 32KB private L1, 256KB private L2 and 30MB shared L3. The operating system is Linux 2.6.32 x86_64 shipped with CentOS 6.5.

Micro-benchmark tests

The first set of experiments was aimed at assessing the general properties of the pool evaluation pattern implementation(s), and in particular focusing on the implementation described in Section 3 (P1 case).

Therefore, we considered a synthetic benchmark where the various functions used in the pattern (termination, selection, evaluation, filter) are simple functions performing floating point computations while accessing some fixed amount of memory. In all tests, the number of memory accesses to shared and private data structures is kept fixed, while the amount of time spent in the computation of each function may be varied by adjusting application parameters.

We studied the performance, and thereby the overhead introduced by the parallel implementation, when varying the computational grain for the single element of the population in the evolution phase. We considered three distinct cases:

1. one single floating point operation per element (1 flop);
2. up to ten floating point operations per element (10 flops);
3. up to one hundred floating point operations per single element (100 flops).

The results of these tests are sketched in Fig. 3. As can be seen, the pool pattern implementation is able to obtain quite good performance even in the case of very fine-grain computation for the single element of the population. In particular, for the `FastFlow` implementation the maximum scalability for the case 1flop is 4.1 considering the overall execution time. But, if we consider

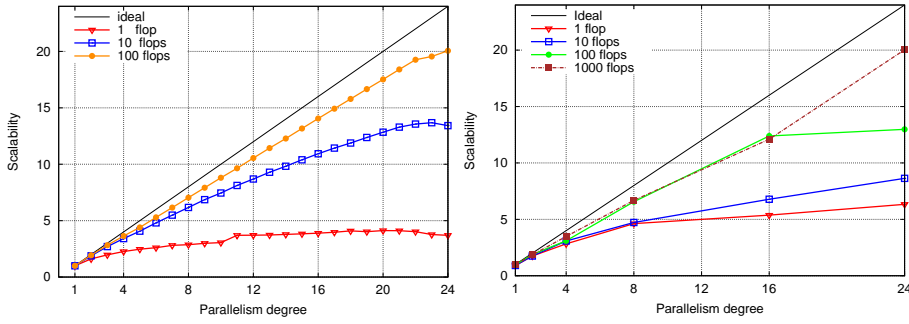


Fig. 3 Benchmark scalability (P1 version) varying the parallelism degree of the evolution phase for three different levels of computation grain (1x, 10x and 100x floating point operations per single population element). The selection and filtering phases are kept sequential, the number of iterations is fixed at 50. Left): FastFlow C++ implementation. Right): skel Erlang implementation. Floating point operations are used only as an indicative measure of the amount of computation per operator and refer to single precision.

only the execution time improvement for the evolution phase, the gain is much greater: the execution time of this phase goes from $300ms$ using 1 thread down to $15ms$ using 24 threads. The Erlang graph in Fig. 3 also plots the curve relative to 1k flop, showing that the interpreted execution of Erlang needs somewhat larger grain to achieve reasonable speedups.

The next benchmark is aimed at assessing the pool evolution pattern when all phases (selection, evolution and filtering) are executed in parallel according to the P2 schema described in Section 3. In this test, the selection function computes the average value of a population of N elements ($N = 10^6$ in the tests) and then selects all elements whose value is greater than the average value. In this way, at each iteration of the pool evolution loop, the population decreases. This benchmark has been run using the FastFlow implementation of the pool pattern, as this is able to exploit much finer grain parallelism than the Erlang implementation, as pointed out before. The computation ends when there are fewer elements than a given threshold value. The evolution and filtering functions apply a synthetic function on each element of the selected population. The results of this test, varying the parallelism degree for each phase, are sketched in Fig. 4. As can be seen, by increasing the number of worker threads in each phase, the execution time decreases up to (12,24,48) and then starts to increase slowly. It is interesting to note is that using the maximum level of parallelism in each phase (i.e. (48,48,48)) does not lead necessarily to the best execution time for the case considered, even though the best execution time for the single phase (i.e. considering the other two sequential) is obtained when using the maximum number of threads/cores in the target platform. Therefore, having the flexibility to vary the parallelism degree for each single phase of the pool evolution pattern gives greater possibility of reaching the best execution time on the target platform.

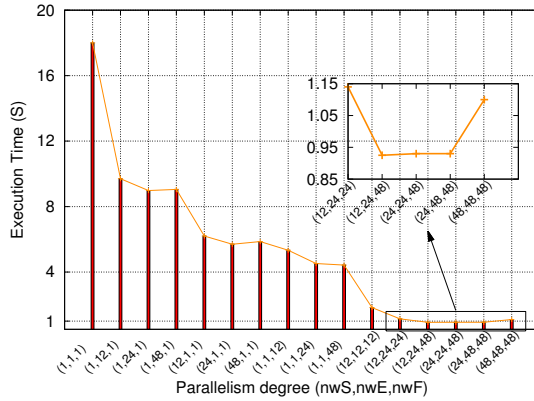


Fig. 4 FastFlow poolEvolution benchmark (P2 version) execution time while varying the parallelism degree of each of the three phases: (nwS, nwE, nwF) is the number of worker threads used in the selection, evolution and filtering phases, respectively. The total number of iterations executed in each step is 21.

Function approximating a set of points

The second experiment was aimed at assessing the implementation of the pattern with an application. We considered the problem of finding the best function approximating another (unknown) function defined by a given set of $\langle x_i, f_{measured}(x_i) \rangle$ pairs [15,16]. In this case we developed only the FastFlow implementation. The selection, termination and filtering functions are computed sequentially, with only the evolution component being parallelised, and only a single run of the pool pattern is executed (stream length $m = 1$). The initial population is given by a set of randomly generated functions represented as *trees*. Each node of the tree can be a variable, a constant, or an application of a binary ($g \in \{-, +, *, /, power\}$) or unary ($f \in \{exp, sin, cos, log\}$) operator. Variables and constants are leaves of the tree, a unary operator has a single child subtree and a binary operator has two child subtrees.

The application evaluates the approximating function f' by minimizing the distance assumed by each $x \in P = \{x_0, \dots, x_n\}$ in the target function $f_{measured}(x)$ w.r.t. the approximated value $f'(x)$. For each function (tree) f' , we evaluate in parallel the root mean square error over P , given by

$$E(f', P) = \sqrt{\frac{\sum_{i=0}^n (f'(x_i) - f_{measured}(x_i))^2}{n}}$$

and the functions having the better fitness value are selected for modification by the *crossover* or the *mutation* operator. The crossover randomly selects two subtrees belonging to two different trees respectively and exchanges them. The mutation substitutes a randomly selected subtree with a new one of the same depth. In both cases the newly generated trees substitute the modified ones in the population. Thus, generation after generation, the functions exhibiting the

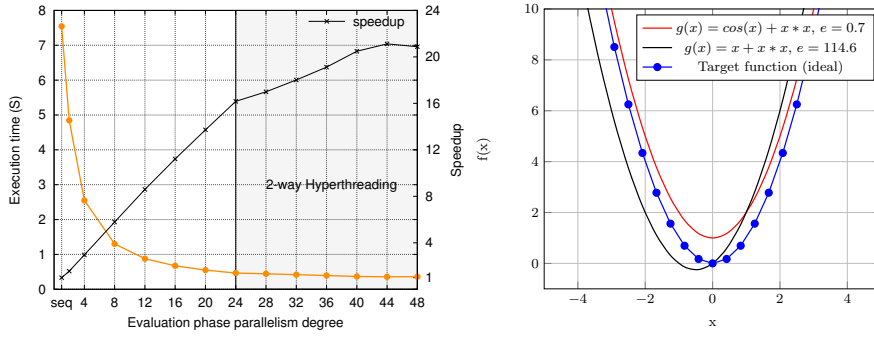


Fig. 5 Application execution time (P1 version) varying the parallelism degree of the evaluation phase (left) and Approximation of $f(x) = x^2$ by two different runs (5 vs 50 iterations over the same initial population (right)).

better fitness (i.e. showing minimal $E(f, P)$) are kept within the population and contribute to production of improved “children” until a reasonable or the exact solution is reached. In other words, the problem of approximating a function given a set of points P is a search of the function that minimises $E(f, P)$ as far as possible.

The scalability results achieved are presented in Fig. 5 (left). The Figure clearly shows that the application scales well on the target architecture (20 fold speedup with the available resources). In this case the serial fraction of the computation for each iteration (i.e. the time spent in the selection, filtering and termination functions) is negligible with respect to the time spent in the computation of the evolution function, so in principle we should have obtained an even higher scalability. The main bottleneck in this case is the memory bandwidth available per core for accessing shared and private data structures. Hyperthreading helps in masking memory access time and so provides extra speedup. In fact, by doubling the number of worker threads from 24 to 48 we obtained an improvement of about 23%. A more complex and time consuming re-design of shared data structures, together with careful placement of objects in the different memory banks of the platform, would probably produce even better results at the cost of reduced portability. This limits the possibility of reaching the ideal performance on the target platform.

In Fig. 5 (right), the convergence of a run of our application approximating $f(x) = x^2$ is shown over a population of 100 function “trees”. Each tree is built with a depth of at most three levels, and the initial population is randomly created using the tree grammar. In the figure a run over 50 generations is compared with a run over 5 generations: the first run computes $g(x) = x + x^2$ as best solution, while producing an error $e = 114.604$. In the second run, besides the shorter “evolution” time, the error is reduced to 0.7070 and an approximated function $g(x) = \cos(x) + x^2$ is computed. The figure shows that in both cases the algorithm reached a form of convergence. For the sake of simplicity we decided to consider a population comprising only 3-level trees.

However, due to the limited genetic information provided by these trees, single modifications introduced by crossover and mutation operators could imply huge differences in the tree structures and as a consequence, to their fitness values. Thus, in the initial part of the evolution process, it could happen that at a lower number of generations, a better solution is found in terms of error produced. This phenomenon is recognised in evolutionary algorithm theory and, in general, the convergence will improve as the number of generations increases. In fact, in this sample, over 100 iterations we got the approximated function being $f(x) = \sin(x) + x^2$, thus confirming a certain convergence stability.

Convergence also depends on the size of the starting population: in general bigger is better. For instance, by duplicating the population to obtain 200 individuals, the exact solution is again obtained, thus confirming that increasing diversity allows a greater exploration of the solution space: in 5 iterations the optimal detected function result was $g(x) = (0 + x) * (x - 0) = x * x$, while in 50 iterations the optimal is, equivalently, $g(x) = x * x + x^0 = x * x + 1$ with a greater error but a smaller number of tree nodes. In 100 iterations, the solution tree is $g(x) = x * x + \sin(0) = x * x$ which is optimal and is also the shortest optimal tree found. In this example, we can see that, even if the fitness is apparently unstable at the beginning of the evolutionary process, the convergence improves over a greater number of iterations both in terms of error and in terms of the structure of the solution tree.

Finding the global minimum of a function in a range

In the third experiment, we implemented an application searching for the global minimum of an input function in a range $[a, b]$ [13]. The aim of this experiment, leading to both C++/FastFlow and Erlang versions of the application, is to check whether the two implementations (FastFlow and Erlang) of the pool pattern exhibit similar behaviour on the same application and to experiment with different versions of the pool pattern, namely the simple P1 version and the more parallel P2 version described in Section 3.

The application searching for the minimum of a function in an interval is implemented as follows. A set of random points $x_1, \dots, x_n \in [a, b]$ are selected as the initial minimum candidates. This initial population is submitted to the evolution pool pattern where:

- The selection phase is the identity function.
- In the evolution phase, for each of the points in the candidate set (population) p we consider the values $f(p + \epsilon)$ and $f(p - \epsilon)$ and determine the minimum f value. The corresponding point ($p - \epsilon$ or $p + \epsilon$) is kept as candidate in place of the original p . This process is iterated, on each point, for a given number k (*internal iterations*). At the end of this step the new set of minimum candidates is considered and:
 - the *best* candidate (i.e. the one leading to the minimum f value) is kept;
 - a portion of the rest of the candidates is subject to crossover; and

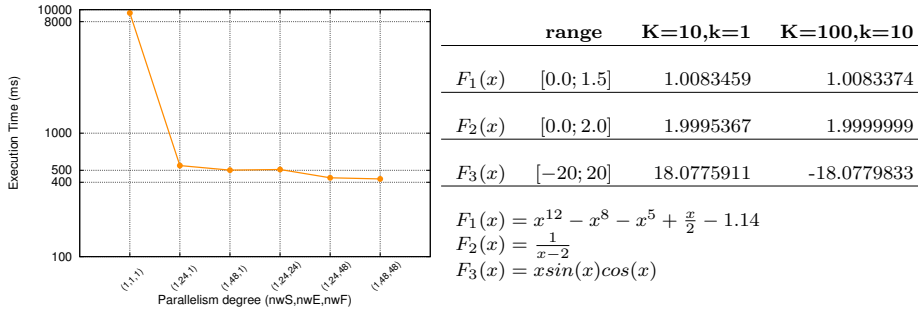


Fig. 6 Execution time (in milliseconds, P2 version of the pattern) when varying the parallelism degree of the evolution and filtering phases using $P = 1000$, $t = 500$, $K = 200$, $k = 100$ in the range (1, 1.1) (left) and, the minimum value found when evaluating three different functions F considering two external (K) and internal (k) iteration cases ($P = 10000$, $t = 5000$, $\epsilon = 10^{-6}$) (right).

- the rest of the candidates are discarded and substituted by new random points in $[a, b]$.
- The filter phase is again the identity function.

The pattern is iterated a maximum number of times (*global or external iterations*) or up to convergence to a given minimum value.

The whole application is defined by a set of parameters which are: the dimension of the search space, the number of candidates, the number of iterations for the global as well as for the local search, the number of survivors (i.e. the candidates subject to crossover) and, for the parallel version, the number of workers involved in the pool.

In Fig. 6 (left) we can see the execution time exhibited by the application implemented with the P2 FastFlow pool implementation while varying the parallelism degree using 1000 points and 500 “survivors” over 200 global iterations and 100 internal iterations. The chart shows that the minimum execution time (426ms) is obtained when using the maximum parallelism degree for both the evolution and filtering phases. By increasing the population size ten times, we measured a speedup increase of about 25% (from 22.1 to 28.33) confirming that performance remains strong while scaling the problem size. Fig. 6 (right) shows the behavior of the application with respect to three different object functions whose search space varies from a very small to a large one. Each row of the table shows the minimum value returned by the application when varying the internal *vs* external iterations rate. As can be seen, results are sensitive in precision to the parameters given. Moreover, in the case of a function like $x \sin(x) \cos(x)$ which is symmetric with respect to the x -axis and has two minimums at approximately ± 18 , different runs can converge to different values depending on the iterations ratio specified as input. These results show that the pattern, when used to implement a genetic algorithm, reflects the high sensitivity with respect to the input parameters provided by the user, which is typical of these kinds of computation.

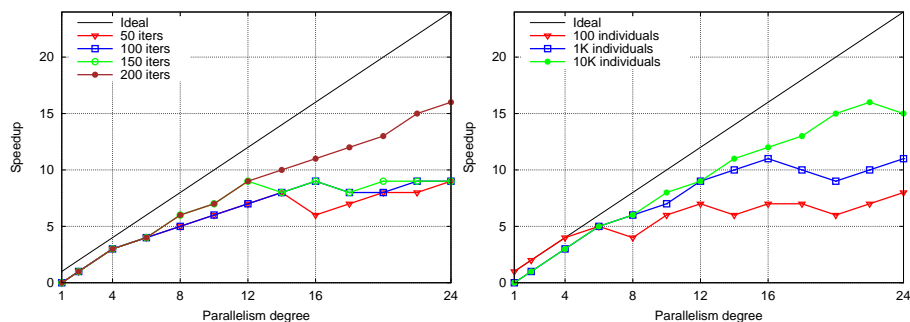


Fig. 7 Speedup varying the computational grain of the evaluation phase (left) and the number of individuals in the population (right) (Minimum of a function in an interval, Erlang P1 pool implementation).

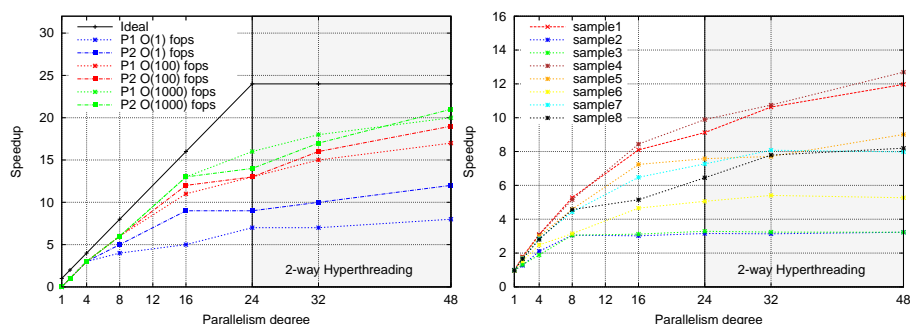


Fig. 8 Speedup of the Erlang application computing the minimum of a function in an interval (P1 vs. P2 version of the pool, left), Speedup of brute force sudoku solver (Erlang, P1 version of the Pool, right)

Fig. 7 presents the variation in the speedup of the application finding the minimum of a function in an interval implemented in Erlang (P1 version of the pool). Augmenting the number of internal iterations performed during the evolution phase, the speedup increases as a consequence of the augmented grain of the evolution phase. The speedup increases also when the number of individuals in the population increases, again as a consequence of the larger number of single individual evolution functions to be computed in the single “evolution” map partition.

Finally, Fig. 8 (left) shows the effect of using the P2 version of the pool pattern vs. the P1 version while computing the minimum of a function in an interval (Erlang version). As expected, the version with the parallel filter phase performs better, although the increase in speedup turns out to be larger with finer grain computations (smaller number of flops used to compute the function $f(x)$).

Brute force sudoku solver

The fourth experiment is aimed at demonstrating the feasibility of using the pool pattern to implement symbolic computations. In particular, we implemented the orbit pattern discussed in Section 2.2.1 using the pool pattern and also used it to implement a brute force sudoku solver as follows:

- a set of generators g_1, \dots, g_9 is used, such that each g_i applied to a sudoku board configuration B generates the configuration B' where all the empty positions are filled with the item i , if and only if the fill does not conflict with already completed board positions;
- a null filter function is used, as the board configurations generated are different from those already present in the population (initially just the start board configuration) *by construction*;
- the termination function simply checks the presence of a board configuration with no empty cells.

Obviously, the orbit brute force sudoku solver suffers the state explosion problem. To avoid the explosion of the board configurations in the population, we adopted a severe filtering function adding only a few configurations to the population list, rather than adding all the (huge number of) configurations generated at each step.

We implemented the sudoku brute force solver in Erlang, using the P1 version of the pool. The typical results achieved are shown in Fig. 8 (right). With different initial board configurations (with more or less alternative values suitable for filling a single cell to be evaluated at each step), the speedup varies between very small numbers (about 3) to higher values (close to 12) on the 24 core test machine. However, if we relax more and more the filtering phase such that more and more alternatives are considered, we easily come up with speedups close to 20 in computations requiring quite a lot of memory to host the population state. In all the experiments, the time spent to get the solution was in the range of the hundreds of milliseconds when 16 to 24 cores were used.

5 Related work

The design pattern community is increasingly interested and active in the field of *parallel* design patterns. Besides the groups that published the two classic parallel pattern books [17, 18], several different authors and research groups contribute to the research area. The pages at <http://www.cs.uiuc.edu/homes/snir/PPP/> and <http://www.cs.wustl.edu/~schmidt/patterns-ace.html> present a number parallel patterns and parallel pattern frameworks. To the best of our knowledge, no parallel pattern has previously been proposed to model evolutionary applications. Different research groups active in the algorithmic skeleton area have put forward algorithmic skeletons modeling com-

plex, high level parallel patterns. The MALLBA team⁴ proposed several algorithmic skeletons including skeletons modeling genetic algorithms and memetic operators. However, they keep distinct the two skeletons rather than providing a general skeleton that may be specialized to realise the implementation of the two patterns [1]. In [9] general purpose iterative skeletons are proposed that may be used to model genetic computations. The focus here is on iterative processes, however, with genetic computations being employed as a use case. Within the ParaPhrase project, our pool evolution pattern is used in two different contexts: researchers using Erlang to implement Agent Systems are considering the pool evolution pattern to model agent interactions; and researchers working in the use-case workpackage are experimenting with the pool evolution pattern to model parallelism exploitation in applications previously using low level patterns such as pipelines and task farms [21].

6 Conclusions and future work

We have introduced a new parallel pattern, the pool evolution pattern, that captures the idea of the evolution of a population. The pattern has been derived as a generalisation of an existing domain-specific pattern, and has been shown to have wider applicability in the evolutionary and symbolic computing domains. We have also discussed how different specialized versions of the pattern—employing different amounts (and kinds) of parallelism—may be used to implement different applications, ranging from numeric applications (minimum of a function, approximation of an unknown function) to symbolic (sudoku). We designed and implemented implementations of the different variants of the pool pattern in C++/FastFlow as well as in Erlang/skel. Both implementations run on top of state-of-the-art shared memory multicore servers. A full set of experiments has been discussed assessing the features of the pool pattern as well as the efficiency and scalability of the pattern when used to implement various parallel applications. In particular, we have demonstrated that reasonable performances may be achieved with modest programming effort while noting that, in certain cases, manual, *ad-hoc* optimisation of the parallel code taking into account the specific target architecture features may lead to further minor performance improvement.

Acknowledgements The work has been partially supported by FP7 EU projects No. 288570 ParaPhrase and No. 609666 Repara. We wish to thank Holger Schöner who contributed greatly to the abstraction of the evolution pool pattern from the genetic sub-patterns.

References

1. Alba, E., Luque, G., Garcia-Nieto, J., Ordonez, G., Leguizamon, G.: Mallba: a software library to design efficient optimisation algorithms. *Int. J. Innov. Comput. Appl.* **1**(1),

⁴ <http://www.lsi.upc.edu/~mallba/public/>

- 74–85 (2007). DOI 10.1504/IJICA.2007.013403. URL <http://dx.doi.org/10.1504/IJICA.2007.013403>
2. Alba, E., Troya, J.M.: A survey of parallel distributed genetic algorithms. *Complex.* **4**(4), 31–52 (1999). DOI 10.1002/(SICI)1099-0526(199903/04)4:4<31::AID-CPLX5>3.3.CO;2-W. URL [http://dx.doi.org/10.1002/\(SICI\)1099-0526\(199903/04\)4:4<31::AID-CPLX5>3.3.CO;2-W](http://dx.doi.org/10.1002/(SICI)1099-0526(199903/04)4:4<31::AID-CPLX5>3.3.CO;2-W)
 3. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating code on multi-cores with fastflow. In: E. Jeannot, R. Namyst, J. Roman (eds.) *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing, LNCS*, vol. 6853, pp. 170–181. Springer, Bordeaux, France (2011). DOI 10.1007/978-3-642-23397-5_17
 4. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiawicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. *Commun. ACM* **52**(10), 56–67 (2009). DOI 10.1145/1562764.1562783. URL <http://doi.acm.org/10.1145/1562764.1562783>
 5. Brown, C.: SKEL home page and repository (2012). <https://github.com/dhg/skeleton>
 6. Chen, X., Ong, Y.S., Lim, M.H., Tan, K.C.: A multi-facet survey on memetic computation. *Trans. Evol. Comp* **15**(5), 591–607 (2011)
 7. Cole, M.: Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.* **30**(3), 389–406 (2004). DOI 10.1016/j.parco.2003.12.002. URL <http://dx.doi.org/10.1016/j.parco.2003.12.002>
 8. Danelutto, M., Torquati, M.: Loop parallelism: a new skeleton perspective on data parallel patterns. In: M. Aldinucci, D. D’Agostino, P. Kilpatrick (eds.) *Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing*. IEEE, Torino, Italy (2014)
 9. Dieterle, M., Horstmeyer, T., Berthold, J., Loogen, R.: Iterating skeletons - structured parallelism by composition. In: R. Hinze (ed.) *IFL, Lecture Notes in Computer Science*, vol. 8241, pp. 18–36. Springer (2012)
 10. Elliott, A., Brown, C., Danelutto, M., Hammond, K.: Skel: A Streaming Process-based Skeleton Library for Erlang (2012). 24th Symposium on Implementation and Application of Functional Languages, IFL 2012, Oxford, UK
 11. FastFlow home page (2014). <http://sourceforge.net/projects/mc-fastflow/>
 12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
 13. Goldberg, D.E.: *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1989)
 14. Hammond, K., Zain, A.A., Cooperman, G., Petcu, D., Trinder, P.: Symgrid: A framework for symbolic computation on the grid. In: S. Verlag (ed.) *Euro-Par 2007 Parallel Processing, Lecture Notes in Computer Science*, vol. 4641, pp. 457–466 (2007)
 15. Keane, M.A., Koza, J.R., Rice, J.P.: Finding an impulse response function using genetic programming (1993)
 16. Koza, J.R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA (1992)
 17. Mattson, T., Sanders, B., Massingill, B.: *Patterns for Parallel Programming*, first edn. Addison-Wesley Professional (2004)
 18. McCool, M., Reinders, J., Robison, A.: *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science (2012). URL <http://books.google.it/books?id=2hYqeo08t8IC>
 19. ParaPhrase web page (2014). <http://www.paraphrase-ict.eu>
 20. ParaPhrase project report. Final Pattern Definition Report (2013). Available at <http://www-paraphrase-ict.eu>
 21. Rossbory, M., Reisner, W.: Parallelization of algorithms for linear discrete optimization using paraphrase. *Proc. of the 23rd International Workshop on Database and Expert Systems Applications* pp. 241–245 (2013). DOI <http://doi.ieeeecomputersociety.org/10.1109/DEXA.2013.32>
 22. SCIENCE deliverable series: Deliverable D5.10 (JRA 1.6) – Report on Symbolic Computation Grid Exemplars (2012)
 23. SCIENCE deliverable series: Deliverable D5.13 (JRA 1.6) – Report on Multilevel Parallelism (2012)