7th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2014)
Amsterdam, Netherlands
July 3-4, 2014

Pool evolution: a domain specific parallel pattern

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

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, developed in the framework of the ParaPhrase EU-funded FP7 project, 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 effective programming methodology, greatly improving the time-to-deploy and maintainability of complex applications.

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

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 [14],

> ... 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 [16], 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 actual 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 within ParaPhrase 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 in Mattson's book 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 [15] are also generic/general purpose. However, 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 possibile 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.

We are therefore faced with two distinct but opposite forces:

- domain specific patterns may significantly improve the productivity of programmers in the specific domain; but
- general purpose patterns may be more easily implemented, optimized and eventually adapted–via suitable functional and non-functional parameters–to implement domain specific parallel computations.

If a domain specific pattern is identified that may be generalized to a non-domain specific context and still proves to be useful in a number of different applications, the domain specific pattern becomes a worthwhile candidate to extend our parallel pattern set.

In this paper we introduce a new parallel pattern that 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.

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

- Definition of a new, domain specific, 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's book.
- 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 rest of this paper is organized as follows: Sec. 2 introduces the new pattern and lists different applications whose parallel structure may be suitably modeled using the pattern. Sec. 3 outlines possible implementation strategies relative to an algorithmic skeleton implementing the pattern and then outlines the actual FastFlow and Erlang skeleton implementations. Sec. 4 presents some preliminary experimental validation of the FastFlow and Erlang implementations of the new pattern. Finally, Sec. 5 outlines related work and Sec. 6 draws conclusions.

Name	Pool evolution pattern
Problem	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: 1: **while** $not(t(P))$ **do** 2: $N = e(s(P))$ 3: $P = P \cup f(N, P)$ 4: **end while**
Forces	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.

Table 1 Pool evolution pattern

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 specializations of the new pattern.

2.1 Pattern definition

By means of Tab. 1 we provide a concise pool evolution pattern definition, in the style of Mattson's book. We deliberately omit the solution and example sections, as pattern implementation will be discussed thoroughly in Sec. 3 and sample usage of the pattern will be discussed later in this Section and in Sec. 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 build of different

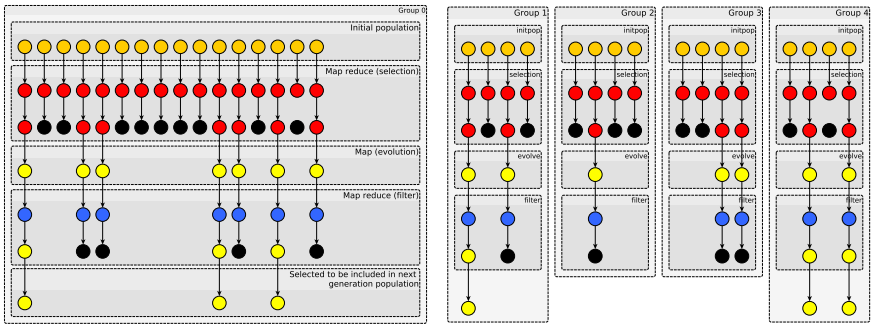

Fig. 1 Alternative implementations (with multipopulation) of the pool evolution pattern: single population (left) vs. multi population (right)

stages (selection, evolution, filtering) that 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 Tab. 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 obviously makes the selection process a good candidate for application of a mapreduce pattern: first map the function estimating how good is an individual, then filter (reduce) the better candidates.
- The evolution process is embarrassingly parallel: each individual 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 space[1], the complete iteration process may be structured as follows:

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

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] actually we assume here to have the corresponding algorithmic skeletons available

1. splitting the population into disjoint groups $G_1, \ldots, G_g$;
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 in 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 actually relevant, as the *Solution* text only suggests possible solutions but it does not actually 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 [13, 19–21]. 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 $gen_1(p_i), gen_2(p_i), \ldots, gen_g(p_i)$) 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 *symbolic computing* domain and models the iterative construction of a set of items starting from an initial set (S) of items using *generator functions* from a generator function set ($G = \{g_1, \ldots, g_k\}$). 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 of avoiding 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 *genetic operator* 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 (see Tab. 1)
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 $A_1, \ldots, A_n$, each executing one or more jobs $j_1, \ldots, j_m$. Jobs could be provided with a weight and agents can have a limit of workload assigned $a_1, \ldots, a_n$. 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 $\sum_{a \in \{A_1, \ldots, A_n\}} M_i \frac{1}{t_i} U_i$ 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 optimization and combinatorial optimization 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*. Actually, they can be understood as the pattern(s) that generated–through generalization 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 synchronization 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 (rather than patterns) are concerned, we consider a few representative applications whose parallel behaviour may be readily modeled by the pool evolution 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 that 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 an empty 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 Sec. 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 ⟨ point, value ⟩ 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 Sec. 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 P2. The Erlang version, instead, is built on top of the **skel** Erlang skeleton library [5, 10] and, in particular, uses a *map*

skeleton instance to implement in parallel the *evolve* function over the selected population items (schema P1).

The Erlang version is very compact and the actual code corresponds one-to-one to the pseudo code given in Tab. 1 to describe the pattern's functional semantics. The code implementing the pool pattern 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 = skel:do([{map,[{seq,fun ?MODULE:Evolution/1}]}], [Selected]),
9          Filtered = Filter(Evolved),
10         Newset = union(Rest, Filtered),
11         (pool(Termination, Selection, Evolution,Filter))(Newset)
12       end
13   end.
```

where the

```
skel:do([{map,[{seq,fun ?MODULE:Evolution/1}]}], [Selected])
```

is functionally equivalent to a

```
lists:map(fun ?MODULE:Evolution/1, Selected)
```

but is computed in parallel using the *map* skeleton in the **skel** library.

In the FastFlow implementation, by default only the evolution phase is computed in parallel. However, it is also possible to configure the pool implementation to compute the selection and the filtering map-reduce phases in parallel also. 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 parallelization 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 [17]), 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:

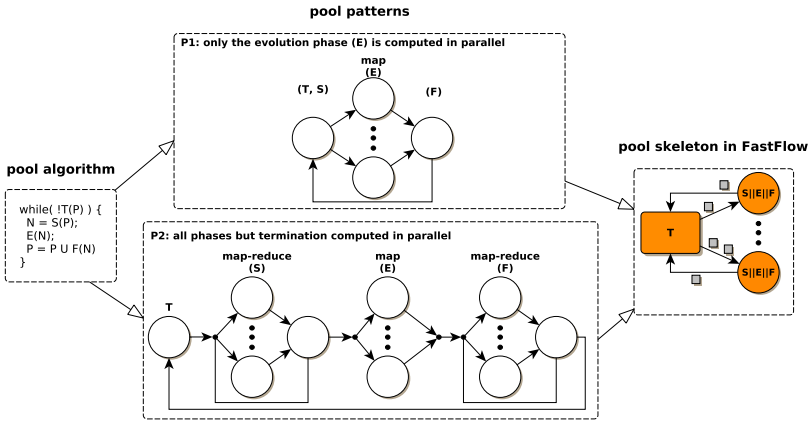

Fig. 2 Algorithm, concurrent activity graphs and the FastFlow parallel skeleton of the pool evolution pattern. Each circle represents a thread, arrows represent communication channels, which are implemented through FastFlow lock-free buffers.

```
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
8    */
9    typedef void(*selection_t)(ParallelForReduce<T>&,std::vector<T>&,std::vector<T>&,
         env_t&);
10   typedef const T&(*evolution_t)(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 stream[2].

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 a 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:

```
#include <ff/poolEvolution.hpp"
const long MAXGENERATION = 100;
typedef long Env_t; /* this is my simple environment */
/* implementation of an individual */
class Individual {
  ...
};
//--------------- genetic operators --------
bool termination(const std::vector<Individual> &P,Env_t &E) {
  return E.numGeneration >= MAXGENERATION;
}
void selection(ParallelForReduce<Individual> &mapreduce,
               std::vector<Individual> &P,
               std::vector<Individual> &newP, Env_t &E) {
 // implementation of selection: P --> newP
}
const Individual& evolution(Individual &t) {
// implement the evolution changes in t
}
void filter (ParallelForReduce<Individual> &mapreduce,
             std::vector<Individual> &P,
             std::vector<Individual> &newP, Env_t &E) {
  // filter individuals to be added to current pop
  newP += P;
}
/*----------end of genetic specific stuff ---------*/

int main(int argc, char *argv[]) {
  std::vector<Individual> initialP = ....;
  Env_t num_generations=0; /* my simple environment */
  /* instantiate the pattern with a max parallelism of 48 */
  poolEvolution<Individual,Env_t>
  pool(48, initialP , selection , evolution , filter , termination,
       num_generation);
  if (pool.run_and_wait_end()<0)
    error (" poolEvolution fails to run\n");
  ...
}
```

[2] 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 two real applications: as described in Sec. 2.2, the first searches for the best function that approximates a given set of ⟨ point, value ⟩ pairs; the second computes the approximation of the minimum of a given function in a given interval ($min\ f(x)\ x \in [a, b]$). The first application provides an example of how the different phases of the pattern could be parallelized; the second experiment, focusing on the parallelization 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.

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 one 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 our pool evaluation pattern implementation(s), and in particular focusing on the implementation described in Sec. 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 the case $1flop$ is 4.1 considering the overall execution time. But, if we consider only the execution time improvement for the evolution phase, the gain is much more: the execution time of this phase goes from $300ms$ using 1 thread down to $15ms$ using 24 threads. The Erlang graph in the Fig. 3 actually refers to

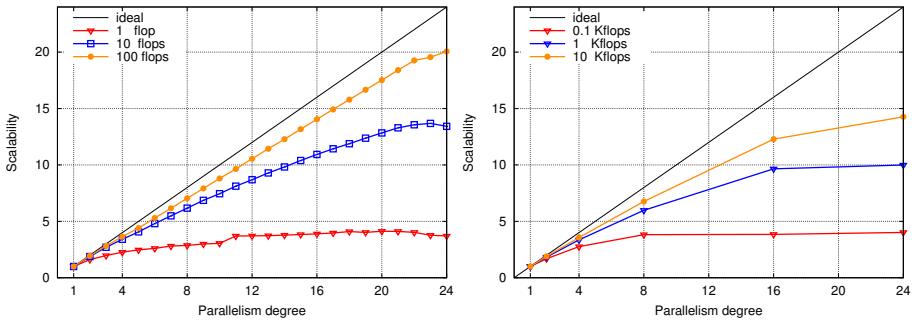

Fig. 3 Benchmark scalability (P1 version) varying the parallelism degree of the evolution phase for three different levels of computation grain (1x, 10x and 100x floating point operations per single population element). The selection and filtering phases are kept sequential, the number of iterations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

computations with 100 times the amount of floating point operations with respect to the left FastFlow graph. This evidences the coarser grain needed to scale in the Erlang implementation, clearly deriving from the fact that Erlang is interpreted.

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 Sec. 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 than starts to increase slowly. What 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 exploiting 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.

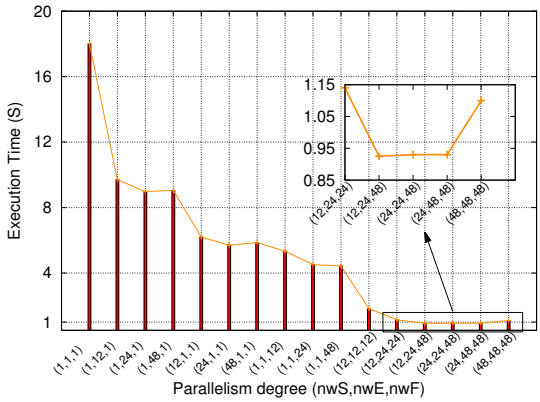

Fig. 4 FastFlow poolEvolution benchmark (P2 version) execution time varying the parallelism degree of each of the three phases: (nwS, nwE, nwF) is the number of worker threads used in the selection, evolution and filtering phases, respectively. The total number of iterations executed on each step is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

Function approximating a set of points application

The second experiment was aimed at verifying the implementation of the pattern with a real 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. In this case we developed only the FastFlow implementation. The selection, termination and filtering functions are computed sequentially and only a single run of the pool pattern is executed (the stream length in this case is $m = 1$).

The application computes the approximation function f' by minimizing the distance between each point belonging to the target function $f_{measured}(x)$ and the approximated value $f'(x)$. As a consequence, the fitness for a function f measured on a set of points $P = \{(x_o, y_o), \ldots, (x_n, y_n)\}$ is represented by the error

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

At the beginning of the computation, a set of random functions is generated and a percentage of them are selected as potential candidates for evolution. Each function is represented by a binary tree built using the following grammar:

$$\begin{array}{ll}
\langle const\rangle & ::= 0 \mid 1 \mid \ldots \\
\langle var\rangle & ::= x \mid y \mid \ldots \\
\langle unaryOp\rangle & ::= exp|sin|cos|log \\
\langle unaryNode\rangle & ::= exp(\langle node\rangle) \mid sin(\langle node\rangle) \mid cos(\langle node\rangle) \mid log(\langle node\rangle) \\
\langle binOp\rangle & ::= -| + | * |/|pow \\
\langle binaryNode\rangle & ::= \langle node\rangle\langle binOp\rangle\langle node\rangle
\end{array}$$

$$\langle node \rangle \quad ::= \langle var \rangle \mid \langle const \rangle \mid \langle unaryNode \rangle \mid \langle binaryNode \rangle$$

Each node of the tree can be a variable, a constant, or an application of a binary or unary 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.

For each function (tree), we evaluate in parallel the error measure over all points, and the functions having the better fitness value are selected for modification by the *crossover* operator or by the *mutation* operator. The crossover operator works on two functions at a time. It is implemented by randomly selecting a single cut point on each of the two trees and exchanging the related subtrees. The two children produced by "mating" the two functions are added to the population in place of their parents so that the size of the population does not change during the overall computation. The mutation operator works on a single function. It is implemented by randomly selecting a single subtree in the function tree and substituting the subtree by another random subtree with the same depth. The resulting function is added to the population in place of the original function, and so this operator does not change the size of the population. Ideally, generation after generation, the functions scoring the better fitness (i.e. better minimizing the approximation error) are kept within the population and contribute to producing better and better "children" until a reasonable solution, or the exact solution, is reached.

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 24 cores). 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 a 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 an actual 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 grammar above. 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

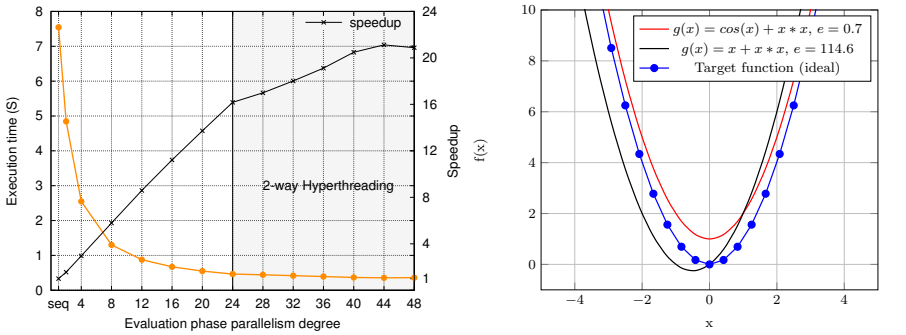

Fig. 5 Application execution time varying the parallelism degree of the evaluation phase (left) and Approximation of $f(x) = x^2$ by two different runs (5 vs 50 iterations over the same initial population (right).

figure shows that in both cases the algorithm reached a form of convergence. Due to the shortness of our sample trees (i.e. to the limited genetic information provided by the 3-level trees), modifications introduced by crossover and mutation operators induce substantial differences in the tree structures and, as a consequence, to their fitness values. Therefore, the solution space is unstable when approximating the optimal solution in a fixed number of generations. This explains why we are much closer to the optimum after 5 generations than after 50 generations.
We also tried the algorithm running 100 iterations with the approximated function being $f(x) = sin(x) + x^2$, thus confirming a certain stability. By duplicating the population to obtain 200 individuals, the exact solution is reached, thus confirming that much more diversity allows better 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. In 100 iterations, the solution tree is $g(x) = x * x + sin(0) = x * x$.

Finding the global minimum of a function in a range

The second experiment concerns the search for a global minimum for a function which is continuous over a given range of values.
The application proceeds in two nested steps:

- given a global interval $[a, b]$, $x_1, \ldots, x_n \in [a, b]$ points are selected and evaluated by a local minimum search in smaller ranges $x_i \pm \Delta$; each local search defines a candidate as a local minimum, all the local minimums are compared and a global minimum is elected. At this point, the genetic logic takes its role since the next generation is built by including the best minimum; the best t individuals are selected as "survivors" and are crossed-over for producing new t individuals and substituted in the population by

their children; the remaining $n - t - 1$ individuals are randomly generated and introduced in the new population for the next generation;
- the search repeats until a given number of iterations is reached
- the search returns the global minimum found.

The local minimum search applied to the set of elements $x_1, \ldots, x_n \in [a, b]$ selected in the global interval is, again, an iterative process:

- given a point p of the space and its function value $f(p)$, values $f(p + \epsilon)$ and $f(p - \epsilon)$ are evaluated in order to determine if the function at those points increases or decreases. At each step, p is substituted by the point that decreases the value of f;
- the process is repeated over a number of k iterations;
- the local minimum search returns the point p' that has better minimized f in the range of $(p \pm \epsilon) \times k$ adjustments.

The 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 and, for the parallel version, the number of workers involved in the pool.

In terms of the pool pattern, the algorithm is described by a starting population represented by the n points in the space $[a, b]$ which are randomly generated in order to start the investigation. Once they have been defined:

- the *selection* phase is actually void, since the *evaluation* applies to all the individuals of the population in parallel;
- the *evaluation* applies the local minimum algorithm to single individuals by iterating it k times;
- the *filter* phase collects all the local minimums defined by the previous, parallel phase and, sequentially, selects the best;
- the *termination* phase determines if the limit of iterations has been reached and, if so, returns the global minimum so defined.

Even though the *filter* could be implemented as a parallel reduce, our implementation defines this application as a use case in which only the evaluation phase is parallelized. Moreover, the pattern is sufficiently flexible that it allows the design of empty phases such as, in this case, the *selection* which does not imply any computational or structural change over the population. In our experiment, the population is represented by pairs $(x, f(x))$ of 250 floating values evolving over an epoch of 10 generations (i.e. global iterations). In Fig.6 we can see the scalability exhibited by the application as the evaluation time varies (i.e. as the number of iterations related to the local minimum search increases).

The chart shows how an increment of the computational grain corresponds to a performance improvement, obtaining very good scalability values in the range of $[50, 200]$ local iterations. The experiment demonstrates that this implementation of the pool pattern performs well even when not all the potential parallelism is expressed. Performance remains strong provided that the computational cost improvements compensate those incurred by the pattern

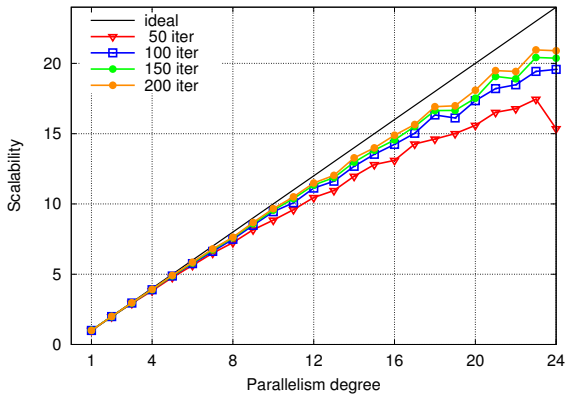

Fig. 6 Scalability varying the computational grain of the evaluation phase.

instantiation.

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 [14, 15], 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 complex, high level parallel patterns. The MALLBA team[3] 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 [18].

[3] `http://www.lsi.upc.edu/~mallba/public/`

6 Conclusions and future work

We have introduced a new domain specific pattern, the pool evolution pattern, which captures the idea of the evolution of a population. We have demonstrated that it has, indeed, wider applicability, as other patterns in the evolutionary and symbolic computing domains may be implemented as specializations of pool evolution. We have also outlined how these specialised versions may be used to implement different applications. In the framework of the ParaPhrase project, the pattern has been implemented via the FastFlow framework and in Erlang. It is currently being used in the development of various use cases. Here we discussed preliminary experimental results demonstrating the scalability of the pool evolution implementation on state-of-the-art multi-core architectures. We are currently performing more experiments, with other applications from different domains, which will be presented and discussed in future research work.

Acknowledgements This work has been partially supported by FP7 EU STREP ParaPhrase and FP7 EU STREP 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), 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`
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., Kubiatowicz, 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. 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)
14. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, first edn. Addison-Wesley Professional (2004)
15. McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming: Patterns for Efficient Computation. Elsevier Science (2012). URL http://books.google.it/books?id=2hYqeoO8t8IC
16. ParaPhrase web page (2014). http://www.paraphrase-ict.eu
17. Paraphrase project report. Final Pattern Definition Report (2013). Available at http://www-paraphrase-ict.eu
18. 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.ieeecomputersociety.org/10.1109/DEXA.2013.32
19. SCIEnce deliverable series: Deliverable D5.8 (JRA 1.5) – Report on Patterns of Symbolic Computation for the Grid (2010)
20. SCIEnce deliverable series: Deliverable D5.10 (JRA 1.6) – Report on Symbolic Computation Grid Examplars (2012)
21. SCIEnce deliverable series: Deliverable D5.13 (JRA 1.6) – Report on Multilevel Parallelism (2012)