A Parallel Swarm Library Based on Functional Programming

Fernando Rubio, Alberto de la Encina(✉), Pablo Rabanal, and Ismael Rodríguez

Facultad Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain
{fernando,albertoe}@sip.ucm.es, prabanal@fdi.ucm.es

Abstract. In this paper we present a library of parallel skeletons to deal with swarm intelligence metaheuristics. The library is implemented using the parallel functional language Eden, an extension of the sequential functional language Haskell. Due to the higher-order nature of functional languages, we simplify the task of writing generic code, and also the task of comparing different strategies. The paper illustrates how to develop new skeletons and presents empirical results.

Keywords: Metaheuristics · Parallel programming · Skeletons · Functional programming

1 Introduction

When dealing with swarm optimization methods (see e.g. [4–6,11]), one of the first problems is deciding which swarm algorithm should be chosen to solve the problem under consideration. The same issue applies for deciding how a swarm method should be parallelized, out of a given set of available parallel strategies. Under these circumstances, it is very useful to provide programmers with several implementations of several swarm intelligence methods – as long as all of them can be easily adapted and used to solve any problem under consideration. In this regard, the reusability and clear separation of concerns of functional programs fits particularly well. In this paper we present a library of parallel functional swarm intelligence algorithms. The chosen parallel functional language is Eden [7], which is a parallel extension of Haskell, a higher-order functional language that guarantees the absence of side effects. The aim of our library is providing programmers with a tool to quickly test the performance of several swarm intelligence algorithms, as well as several parallelizing strategies.

Smaller pieces of the library have been presented in previous works. In [12] we presented our Eden implementation of Particle Swarm Optimization

This work has been partially supported by projects TIN2012-39391-C04-04, TIN2015-67522-C3-3-R, and S2013/ICE-2731.

I. Rojas et al. (Eds.): IWANN 2017, Part I, LNCS 10305, pp. 3–15, 2017.
DOI: 10.1007/978-3-319-59153-7_1

(PSO) [6], whereas an Eden implementation of the Artificial Bee Colony algorithm (ABC) [5] was given in [15]. In this paper we develop an Eden implementation of Differential Evolution [3]. In addition, we glue together these three Eden implementations (and their parallel variants) by constructing a higher abstraction layer. The goal of this tool layer is providing a common unified interface to all supported methods and help the programmer to automatically test the performance of all of these three methods and their variants (as well as others that could be provided in the future) for the target problem.

The rest of the paper is organized as follows. First, we briefly describe the language used. Then, Sect. 3 summarizes the main metaheuristic used in this work. Next, in Sect. 4 we illustrate how to develop generic higher-order functions to deal with a concrete metaheuristic, while in Sect. 5 we show how to provide new parallel skeletons to deal with the same metaheuristic. Afterwards, Sect. 6 presents results obtained with our library. Finally, Sect. 7 presents our conclusions.

2 Introduction to Eden

Eden [7] is a parallel extension of Haskell. It introduces parallelism by adding syntactic constructs to define and instantiate processes explicitly. It is possible to define a new *process abstraction* `p` by applying the predefined function `process` to any function `\x -> e`, where variable `x` will be the input of the process, while the behavior of the process will be given by expression `e`. Process abstractions are similar to functions – the main difference is that the former, when instantiated, are executed in parallel. From the semantics point of view, there is no difference between process abstractions and function definitions. The differences between processes and functions appear when they are invoked. Processes are invoked by using the predefined operator `#`. For instance, in case we want to create a *process instantiation* of a given process `p` with a given input data `x`, we write (`p # x`). Note that, from a syntactical point of view, this is similar to the *application* of a function `f` to an input parameter `x`, which is written as (`f x`).

Therefore, when we refer to a *process* we are not referring to a syntactical element but to a new *computational environment*, where the computations are carried out in an autonomous way. Thus, when a *process instantiation* (`e1 # e2`) is invoked, a new *computational environment* is created. The new process (the child or instantiated process) is fed by its creator by sending the value for e_2 via an input channel, and returns the value for $e_1 e_2$ (to its parent) through an output channel.

In order to increase parallelism, Eden employs pushing instead of pulling of information. That is, values are sent to the receiver before it actually demands them. In addition to that, once a process is running, only fully evaluated data objects are communicated. The only exceptions are *streams*, which are transmitted element by element. Each stream element is first evaluated to full normal form and then transmitted. Concurrent threads trying to access not yet available input are temporarily suspended. This is the only way in which Eden processes

synchronize. Notice that process creation is explicit, but process communication (and synchronization) is completely implicit.

Process abstractions in Eden are not just annotations, but first class values which can be manipulated by the programmer (passed as parameters, stored in data structures, and so on). This facilitates the definition of skeletons [2,14] as higher order functions. Next we illustrate, by using a simple example, how skeletons can be written in Eden.

The most simple skeleton is `map`. Given a list of inputs `xs` and a function `f` to be applied to each of them, the sequential specification in Haskell is as follows:

```
map f xs  =  [f x | x <- xs]
```

that can be read as *for each element* `x` *belonging to the list* `xs`*, apply function* `f` *to that element*. This can be trivially parallelized in Eden. In order to use a different process for each task, we will use the following approach:

```
map_par f xs =  [pf # x | x <- xs]     where pf = process f
```

The process abstraction `pf` wraps the function application `(f x)`. It determines that the input parameter `x` as well as the result value will be transmitted through channels.

Let us note that Eden's compiler has been developed by extending the GHC Haskell compiler. Hence, it reuses GHC's capabilities to interact with other programming languages. Thus, Eden can be used as a coordination language, while the sequential computation language can be, for instance, C.

3 Differential Evolution

Differential evolution (DE) [3] is an evolutionary algorithm for optimizing real-valued multi-modal objective functions. Although it is related to Genetic Algorithms, it is a different option in the universe of evolutionary methods. DE maintains a population of candidate solutions and attempts to improve it by combining existing ones. The method uses NP agents as candidate solutions, where each of these agents is represented by an n-dimensional vector. The initial population is randomly chosen and uniformly distributed in the search space. DE generates new solutions by adding the weighted difference between two agents to a third one. If the new vector improves the objective function of a predetermined population member, this new vector will replace the one it was compared with, otherwise, the old vector remains unchanged.

Let $f : \mathbb{R}^n \to \mathbb{R}$ be the function to be minimized (or maximized), and let $x_i \in \mathbb{R}^n$ be an agent ($1 \leq i \leq NP$) in the population with $NP \geq 4$. The basic DE variant implemented in this paper is explained afterwards. First of all, NP agents are randomly created in the search space. Next, a loop is executed as long as an ending condition is not satisfied (typically, the number of iterations performed does not exceed the limit, or the fitness adequation is not reached). Inside the loop and for each agent x_i in the population, three agents `a, b, c` are

chosen. These agents must be distinct from each other and distinct from agent x_i. Next, we pick a random integer `R` in the range $[1, n]$, and an empty vector `y` with `n` positions is created. Then, the values of vector `y` are created as follows: For each `y(j)` a random real in the range $[0, 1]$ is assigned to variable `r`, and if the value of `r` is lower than the *crossover probability* parameter, `CR` $\in [0, 1]$, or if `j=R` then the value `a(j)+F`$\times$`(b(j)-c(j))` is set to dimension j of variable `y` (`y(j)`); else `y(j)` $=$ `x`$_i$`(j)`. When the initialization of `y` finishes, if `f(y)<f(x`$_i$`)` the i-th agent x_i is replaced with the new vector `y` (`x`$_i =$ `y`). At the end, the agent with the minimum value of f (or the maximum if maximizing) is returned.

Parameter `F` $\in [0, 2]$ is called the *differential weight*. Parameters `F` and `CR` are experimentally chosen.

4 Generic Differential Evolution in Haskell

In this section we show how to develop a new (sequential) metaheuristic by using Haskell (Eden parallelizations will be tackled in the next section). In particular, we consider the implementation of *Differential Evolution*, although we could deal with any other metaheuristic in a similar way.

Functional languages allow creating higher-order functions. Thus, we can take advantage of them to define a generic function `deSEQ` implementing the Differential Evolution metaheuristic. This function will have as input parameter a fitness function, which can be different in each case. It also needs other input parameters, like the number of candidate positions to be used, the number of iterations to be performed, the boundings of the search space, and the concrete parameters F and CR to be used. Moreover, in order to implement it in a pure functional language like Haskell, we need an additional parameter to introduce randomness. Note that Haskell functions cannot produce side-effects, so they need an additional input parameter to be able to obtain different results in different executions. The type of the Haskell function implementing Differential Evolution can be represented as follows:

```
deSEQ::RandomGen a => a          --Random generator
        ->Params                 --Adjustment parameters(F,CR)
        ->Int                    --Number of candidates
        ->Int                    --Maximum number iterations
        ->(Position->Double)     --Fitness function
        ->Boundings              --Search space boundaries
        ->(Double,Position)      --Value and position of best candidate
```

Regarding the representation of `Position`, it must be able to deal with an arbitrarily large number of dimensions. Thus, we can easily represent it by using a list of real numbers. In this case, the length of such list represents the number of dimensions, whereas the concrete elements represent the coordinate values of each of these dimensions. Note that `Boundings` can be defined in a similar way, although a pair with the lower and upper bound for each dimension is considered in this case. Finally, the type `Params` only needs to handle the parameters used

in Differential Evolution to tune up the algorithm, that is, F and CR, which are real numbers. Thus, the needed auxiliary types are the following:

```
type Position = [Double]
type Boundings = [(Double,Double)]
type Params = (Double,Double)
```

After defining the types and the interface of the main function `deSEQ`, we have to define its actual body. First, we have to randomly initialize the candidate solutions. This is done by a simple function `initializeCandidates` (not shown) that distributes the candidates randomly among the search space. After initializing the candidates, function `de'` performs the real work of the algorithm by iterating the application of the basic step as many times as needed. As in the case of the main function `deSEQ`, the auxiliary function `de'` will also need a way to introduce randomness. This is solved by using function `split` to create new random generators. Let us finally note that function `de'` needs the same inputs as the main function (number of iterations, fitness function, etc.), as it has to perform the actual work, but now it uses a list of candidate positions instead of only the number of candidates, as we have already created the appropriate list:

```
deSEQ sg p nc it f bo = obtainBestCandidate (de' sg2 p it f bo initCandis)
  where initCandis = initializeCandidates sg1 nc bo f
        (sg1,sg2) = split sg
type Candidate = (Double,Position)   -- Current value, current position
```

In order to define function `de'` we only need to use a simple recursion on the number of iterations. The base case will be when zero iterations remains. In that case, we return the same list of candidates without modifying it. Otherwise, we use function `oneStepDE` to perform one iteration of the algorithm, and then we go on performing the rest of iterations by using a recursive call to function `de'`:

```
de' _ _ 0 _ _ cs = cs
de' sg p it f bo cs = de' sg2 p (it-1) f bo (oneStepDE sg1 p f bo cs)
  where (sg1,sg2) = split sg
```

For the sake of simplicity, we assume that the only finishing condition is the number of iterations, but we can easily modify it to include alternative finishing conditions.

Finally, we only need to define how to perform each step. First, we have to generate the list of needed random numbers. For each candidate solution we need three random indexes (corresponding to the candidates `a`, `b`, and `c` described in Sect. 3, which will be used to generate a new candidate), one random dimension to be modified for sure, and one random real number for each dimension. This list of real numbers will be used to decide whether the corresponding dimension is to be modified or not, comparing the real number with the CR parameter. Function `genRanIndexDimR` generates the list of random numbers for each candidate, while the predefined higher-order function `zipWith` allows to combine each candidate with the corresponding random numbers generated by function `genRanIndexDimR`. The source code is as follows:

```
oneStepDE sg (dw,cr) f bo cs = zipWith combineCandidate cs rs
  where rs = genRanIndexDimR sg (length cs -1) (length bo)
```

The definition of `combineCandidate` is trivial. It only has to combine one candidate with the random candidates selected using the random numbers `rs`, using the formula shown in Sect. 3. The complete program is available at http://antares.sip.ucm.es/prabanal/english/heuristics_library.

After implementing the higher-order function dealing with DE metaheuristic, the user only needs to provide the appropriate fitness function corresponding to the concrete problem to be solved. Note that the user does not need to understand the internals of the definition of `deSEQ`, but only its basic interface. That is, the programmer only has to call `deSEQ` providing the fitness function and the concrete parameters to be used (number of iterations and so on).

5 Parallel Skeletons

Parallelizing a problem requires detecting time-consuming tasks that can be performed independently. In our case, in each step of the algorithm we can deal independently with each of the candidates. That is, in function `oneStepDE` we could parallelize the evaluation corresponding to each candidate solution. By doing so, we can create a simple skeleton to parallelize DE algorithms. However, in order to increase the granularity of each of the parallel tasks we should avoid creating independent processes for each candidate. It is better to create as many processes as processors available, and to fairly distribute the candidates among the processes. This can be done by substituting `zipWith` by a call to `zipWith_farm`, a parallel version of `zipWith` that implements the idea of distributing a large list of tasks among a reduced number of processes.

By using `zipWith_farm` the speedup improves. However, for each iteration of the algorithm `zipWith_farm` would create a new list of processes, and it would have to receive and return the corresponding lists of candidates. We can improve the parallel performance of the algorithm by parallelizing function `deSEQ` instead of function `oneStepDE`. We start splitting the list of candidates into as many groups as processors available. Then, each group evolves in parallel independently during a given number of iterations. After that, processes communicate among them to redistribute the candidates among processes, and then they go on running again in parallel. This mechanism is repeated as many times as desired until a given number of global iterations is reached.

The implementation of this approach requires using a function `dePAR` instead of `deSEQ`. The new function `dePAR` uses basically the same parameters as `deSEQ`, but instead of using a parameter `it` to define the number of iterations, it uses two parameters `it` and `pit`. Now, the number of iterations will be defined by `it * pit`, where `pit` indicates the number of iterations to be performed independently in each process without communicating with other processes, whereas `it` indicates the number of parallel synchronous steps to be performed among processes. In addition to that, we also include a new parameter `nPE` to define

the number of independent processes to be created. In the most common case, this parameter will be equal to the number of processors available. Taking into account these considerations, the type interface of the new function is as follows:

```
dePAR::RandomGen a => a          --Random generator
        ->Params                 --Adjustment parameters (F,CR)
        ->Int                    --Number of candidates
        ->Int                    --Iterations per parallel step
        ->Int                    --Number of global steps
        ->Int                    --Number of parallel processes
        ->(Position->Double)     --Fitness function
        ->Boundings              --Search space boundaries
        ->(Double,Position)      --Value and position of best candidate
```

The definition of the body of the main function `dePAR` requires creating as many processes as requested in the corresponding parameter. Thus, before defining this function, we will show how to define a function to deal with the behaviour of each process. Such function will need the corresponding parameter to create random values, and it will also receive the tuning parameters of the metaheuristic (i.e. F and CR), the number of iterations to be performed in each parallel step `pit`, the fitness function `f`, and the boundings of the search space `bo`. Then, the process will receive a list with `it` lists of candidates through an input channel, and it will produce as output a new list with `it` lists of candidates. Note that the main function `dePAR` will perform `it` global synchronous steps, where each step will perform `pit` iterations in parallel without synchronization. Thus, `dePAR` will assign `it` tasks as input to each process, and each process will return `it` solutions as output, where those solutions will be used as input of other processes in the next global step. Let us remark that, in Eden, list elements are transmitted through channels in a stream-like fashion. This implies that, in practice, each process will receive a new list of candidates through its input channel right before starting to compute a new parallel step. The complete source code defining a process is as follows:

```
deP sg p pit f bo [] = []
deP sg p pit f bo (bs:bss)=de' sg1 p pit f bo bs : deP sg2 p pit f bo bss
  where (sg1,sg2) = split sg
```

As it can be seen, it is only necessary to define it recursively on the number of tasks. When the input list of lists is empty, the process finishes returning an empty list of results. Otherwise, it uses exactly the same sequential function `de'` described in the previous section to perform `pit` iterations, and then it goes on dealing with the rest of the input lists.

Let us now consider how to define the main function `dePAR`. First, it has to create the initial list of random candidates, exactly in the same way as in the sequential case `deSEQ`. Then, the main difference with the sequential case appears: we create `nPE` copies of process `deP`. Each of them receives the main input parameters of the algorithm (tuning parameters F and CR, fitness function, etc.), and it also receives its own list of tasks `(pins!!i)`. Each element of the list

of tasks contains an input list of candidates, that will be processed by `deP` during `pit` iterations. The output of each process is a new list of lists of candidates. Each inner list was computed after each parallel step, and they must be redistributed among the rest of processes before starting the next global step. This is done by function `redistribute`. The final result of function `bestPAR` is obtained by combining the last results returned by each process. The source code is as follows:

```
dePAR sg p nc pit it nPE f bo = obtainBestCandidate (last poutsFlat)
 where initCandidates = initializeCandidates sg nc bo f
       sgs = tail (generateSGs (nPE+1) sg)
       pouts=[process (deP (sgs!!i) p pit f bo # (take it (pins!!i))
              |i<-[0..nPE-1]]
       poutsFlat = flatXsss pouts
       pins = redistribute nPE (initCandidates:poutsFlat)
```

It is important to note that the user of the library does not need to understand the low level details of the previous definition. In fact, in order to use it, it is only necessary to substitute a call to the sequential function `deSEQ` by a call to the parallel scheme `dePAR`, using appropriate values for parameters `it`, `pit`, and `nPE`. The last parameter will be typically equal to the number of available processors. Thus, the only programming effort will be to decide the values of `it` and `pit`. In case `pit` is very small, the granularity of tasks will be reduced, whereas very large values of `pit` would reduce the possibility to exchange candidates among processes. As a degenerate case, we could use `it` $= 1$ and `pit` being equal to the total number of iterations to be performed. By doing so, we would create groups searching for a solution in a completely independent way.

The previous parallel skeleton can be easily modified to handle different approaches. For instance, when we are using several computers in parallel, it could be the case that each of them is different. Thus, it would be reasonable to assign more candidates to those computers with faster processors, and less candidates to the slower ones. This can be easily done. First, instead of receiving the number of processes, we need to receive as input parameter the speed of each processor. This can be done by using a list of real numbers. Obviously, given the list we can trivially know the number of processes to be created by computing the length of the list. In the implementation, function `dePAR` has to be modified to split each list of candidates according to their relative speeds. That is, `pins` is now created by taking into account the `speeds` parameter:

```
dePARh sg p nc pit it speeds f bo = obtainBestCandidate (last poutsFlat)
 where nPE = length speeds
       initCandidates = initializeCandidates sg nc bo f
       sgs = tail (generateSGs (nPE+1) sg)
       pouts=[process (deP (sgs!!i) p pit f bo)#(take it (pins!!i))
              |i<-[0..nPE-1]]
       poutsFlat = flatXsss pouts
       pins = redistrRelative speeds initCandidates poutsFlat
```

The redistribution considering the relative speed is done by using function `shuffleRelative`, an auxiliary function that first computes the `percentage` of

tasks to be assigned to each process, and then distributes the tasks by using function `splitWith`. It is worth to comment that we do not need to change anything else in the skeleton. In particular, the definition of the process `deP` itself remains unchanged.

6 Experimental Results

In this section we illustrate the usefulness of the library by performing some experiments. Let us remark that the higher-order nature of the language simplifies the development of tools to analyze properties of the different metaheuristic. In particular, we can write new higher-order functions whose parameters are again higher-order functions dealing with different metaheuristics. For instance, we can compare a list of metaheuristics `mths` for the same input problem (given by a concrete `fitness` function and the `bounds` of the search space) by using a higer-order function as follows:

```
compare::[(Position->Double)->Boundings->(Double,Position)]
          -> (Position->Double) -> Boundings -> [Double]
compare mths fitness bounds
  = map (fst . ($ (fitness,bounds)) . uncurry) mths
```

Note that the higher-order function receives as second and third parameters the fitness function and the boundaries of a concrete problem, while the first input is a list of metaheuristics to be compared, where each of them is again a higher-order function that receives a fitness function and the boundaries of the search space. Let us remark that the metaheuristics can have more parameters than those appearing in function `compare`. For instance, Differential Evolution has more parameters: the number of candidates, number of iterations, etc. However, as functions are first class citizens of the language, any metaheuistic can be partially applied. As an example, we can partially apply metaheuristic `deSEQ` to use a concrete random generator, concrete adjustment parameters (F, CR), a concrete number of candidates (75) and a concrete number of iterations (2000) by writing the following expression

```
deSEQ sg (0.47,0.88) 75 2000
```

Its type is exactly `(Position->Double) -> Boundings -> (Double,Position)`. That is, we can use it as one element of the first input list of function `compare`. For instance, we can compare three different configurations of function `deSEQ` for a single problem `ackley` by writing the following:

```
compare [deSEQ sg (0.47,0.88) 75 2000, deSEQ sg (0.47,0.88) 100 1500,
         deSEQ sg (0.32,0.76) 75 2000]
        ackleyFitness ackleyBounds
```

That is, we are comparing three different configurations. The first and the second one use the same values for F and CR, but the first one uses 75 candidates and 2000 iterations, while the second one uses 100 candidates and 1500

iterations. The third configuration uses different values for F and CR, while the number of candidates and iterations is the same as in the first configuration. We can also generate larger lists of configurations by combining parameters using comprehension lists:

```
compare [deSEQ sg (f,cr) nc ni | f<-[0.47,0.32], cr<-[0.88,0.76],
                                 nc<-[75,100], ni<-[1500,2000]]
        ackleyFitness ackleyBounds
```

As it can be expected, we can easily compare the results obtained by both sequential and parallel metaheuristics. For instance

```
compare ([deSEQ sg (0.47,0.88) 75 2000]
          ++[dePAR sq (0.47,0.88) 75 50 40 n|n<-[1..4]])
        ackleyFitness ackleyBounds
```

compares the sequential version with four parallelizations varying the number of processes to be used from 1 to 4, while

```
compare [dePAR sq (0.47,0.88) 75 pit (div 2000 pit) 4 | pit<-[50,100,200]]
        ackleyFitness ackleyBounds
```

compares three parallel implementations, all of them using 4 processes and 2000 iterations, but varying the size of each global step from 50 to 200 iterations. Obviously, the comparison can also include different metaheuristics as follows:

```
compare [deSEQ sg (0.47,0.88) 75 2000, deSEQ sg (0.47,0.88) 100 1500,
         beesSEQ sg 3000 100 1500, psoSEQ sg (-0.16,1.89,2.12) 100 1500]
        ackleyFitness ackleyBounds
```

where we compare two configurations of Differential Evolution, one configuration of Artificial Bee Colony, and another configuration of Particle Swarm Optimization. Our library provides a larger set of functions implementing different kinds of comparisons. For instance, the previous function is extended to execute each metaheuristic n times and to compute average and standard deviation results. We also allow to receive as input not only a problem, but a list of problems, and we analyze the results obtained for all of them, and so on.

In order to show the information we can obtain by using these tools, we compare the results obtained by three different metaheuristics on a given benchmark. In particular, we compare Particle Swarm Optimization, Artificial Bee Colony, and Differential Evolution by using as benchmark a well-known set of functions defined in [16], where we have removed the last six functions of such benchmark because they are simple low-dimensional functions with only a few local minima.

In order to fairly compare the three metaheuristics, for each function we used exactly the same number of fitness evaluations. This number of function evaluations is the same as that defined in [16]. Regarding the tuning parameters of each of the metaheuristics, we use values available in the literature. In particular, the parameters of PSO are taken from [10], in the case of ABC we

use [1], and in the case of DE we follow [9]. The results shown in Table 1 were obtained after computing the average of 50 executions for each metaheuristic. Note that for each metaheuristic we can find a concrete problem where it obtains the best results. However, the metaheuristic that obtains more often the best result in this concrete benchmark is ABC. In fact, by using [8] we can perform an statistical analysis to quantify the differences among the metaheuristics. In particular, aligned Friedman test can be used to check whether the hypothesis that all methods behave similarly (the null hypothesis) holds or not. Let us consider $\alpha = 0.05$, a standard significance level. From results given in Table 1, we calculate that the p-value for aligned Friedman is 0.0027, which allows to reject the null hypothesis with a high level of significance (the p-value is much lower than 0.05). So, the test concludes that the results of ABC, PSO, and DE are not considered similar. Ranks assigned by this test to ABC, PSO, and DE are respectively 19.5, 26.31, and 27.69 (smaller ranks denote better methods).

Regarding the speedups, all of them obtain reasonable speedups taking into account that the effort needed to use the skeletons is negligible: the programmer only changes a call to the sequential higher-order function by a call to the parallel skeleton. Anyway, the speedup obtained by PSO is slightly better (around 10%). The reason is that in each global step PSO only communicates the best position found by each island, while in ABC and DE it is communicated the whole set of bees/candidates computed in the last iteration. Thus, larger communications reduces the speedup.

Table 1. Average optimality comparison among metaheuristics

Funct	Name	Dim	PSO	ABC	DE
$f_1(x)$	Sphere model	30	$1.02 \cdot 10^{-4}$	$3.87 \cdot 10^{-9}$	$6.17 \cdot 10^{-4}$
$f_2(x)$	Schwefel's problem 2.22	30	$8.29 \cdot 10^{-3}$	$1.74 \cdot 10^{-7}$	$2.84 \cdot 10^{-5}$
$f_3(x)$	Schwefel's problem 1.2	30	$1.93 \cdot 10^{-5}$	$3.58 \cdot 10^{3}$	$3.33 \cdot 10^{4}$
$f_4(x)$	Schwefel's problem 2.21	30	$1.45 \cdot 10^{-3}$	1.39	$6.42 \cdot 10^{-1}$
$f_5(x)$	Generalized Rosenbrock's function	30	26.57	0.13	24.01
$f_6(x)$	Step function	30	0	0	0
$f_8(x)$	Generalized Schwefel's problem 2.26	30	-9686.99	-12569.49	-12044.01
$f_9(x)$	Generalized Rastrigin's function	30	$6.97 \cdot 10^{-8}$	0	$9.95 \cdot 10^{-2}$
$f_{10}(x)$	Ackley's function	30	$2.41 \cdot 10^{-3}$	$9.37 \cdot 10^{-5}$	15.53
$f_{11}(x)$	Generalized Griewank function	30	$4.69 \cdot 10^{-3}$	$1.12 \cdot 10^{-10}$	$2.53 \cdot 10^{-4}$
$f_{12}(x)$	Generalized penalized function I	30	$6.33 \cdot 10^{-3}$	$8.96 \cdot 10^{-11}$	$5.45 \cdot 10^{-5}$
$f_{13}(x)$	Generalized penalized function II	30	$4 \cdot 10^{-2}$	$8.7 \cdot 10^{-9}$	$3.99 \cdot 10^{-3}$
$f_{14}(x)$	Shekel's foxholes function	2	1.65	496.58	476.85
$f_{15}(x)$	Kowalik's function	4	$1.22 \cdot 10^{-3}$	$4.69 \cdot 10^{-4}$	$3.075 \cdot 10^{-4}$
$f_{16}(x)$	Six-hump camel-back function	2	-1.0316	-1.0316	-1.0316
$f_{17}(x)$	Branin function	2	0.398	0.398	0.398

7 Conclusions and Future Work

In this paper we have shown the usefulness of the functional programming paradigm to develop generic solutions to deal with swarm intelligence metaheuristics. In particular, we have shown how to develop parallel skeletons for a given metaheuristic, namely Differential Evolution, but the same ideas can be used to deal with any metaheuristic. The higher-order nature of the language simplifies the development of generic functions comparing the results obtained with different configurations.

The results obtained with our library show that the effort needed to use our skeletons is negligible. However, the obtained speedup is good. Anyway, we do not claim to obtain optimal speedup, but *reasonable* speedups at very low programming effort.

As future work, we want to use our library to deal with NP-complete problems appearing in the context of marketing strategies (see e.g. [13]).

References

1. Akay, B., Karaboga, D.: Parameter tuning for the artificial bee colony algorithm. In: Nguyen, N.T., Kowalczyk, R., Chen, S.-M. (eds.) ICCCI 2009. LNCS, vol. 5796, pp. 608–619. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04441-0_53
2. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**, 389–406 (2004)
3. Das, S., Suganthan, P.N.: Differential evolution: a survey of the state-of-the-art. IEEE Trans. Evol. Comput. **15**(1), 4–31 (2011)
4. Dorigo, M., Birattari, M.: Ant colony optimization. In: Sammut, C., Webb, G.I. (eds.) Encyclopedia of Machine Learning, pp. 36–39. Springer, Heidelberg (2010)
5. Karaboga, D., Görkemli, B., Ozturk, C., Karaboga, N.: A comprehensive survey: artificial bee colony (ABC) algorithm and applications. Artif. Intell. Rev. **42**(1), 21–57 (2014)
6. Kennedy, J., Eberhart, R.C.: Particle swarm optimization. In: IEEE International Conference on Neural Networks, vol. 4, pp. 1942–1948. IEEE Computer Society Press (1995)
7. Loogen, R.: Eden – parallel functional programming with haskell. In: Zsók, V., Horváth, Z., Plasmeijer, R. (eds.) CEFP 2011. LNCS, vol. 7241, pp. 142–206. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32096-5_4
8. Parejo, J.A., García, J., Ruiz-Cortés, A., Riquelme, J.C.: Statservice: herramienta de análisis estadístico como soporte para la investigación con metaheurísticas. In: MAEB 2012 (2012)
9. Pedersen, M.E.H.: Good parameters for differential evolution. Technical report HL1002, Hvass Laboratories (2010)
10. Pedersen, M.E.H.: Tuning & simplifying heuristical optimization. Ph.D. thesis, University of Southampton, School of Engineering Sciences (2010)
11. Rabanal, P., Rodríguez, I., Rubio, F.: Using river formation dynamics to design heuristic algorithms. In: Akl, S.G., Calude, C.S., Dinneen, M.J., Rozenberg, G., Wareham, H.T. (eds.) UC 2007. LNCS, vol. 4618, pp. 163–177. Springer, Heidelberg (2007). doi:10.1007/978-3-540-73554-0_16

12. Rabanal, P., Rodríguez, I., Rubio, F.: Parallelizing particle swarm optimization in a functional programming environment. Algorithms **7**(4), 554–581 (2014)
13. Rodríguez, I., Rabanal, P., Rubio, F.: How to make a best-seller: optimal product design problems. Appl. Soft Comput. **55**, 178–196 (2017)
14. Rubio, F.: Programación funcional paralela eficiente en Eden. Ph.D. thesis, Universidad Complutense de Madrid (2001)
15. Rubio, F., de la Encina, A., Rabanal, P., Rodríguez, I.: Eden's bees: parallelizing artificial bee colony in a functional environment. In: ICCS 2013, pp. 661–670 (2013)
16. Yao, X., Liu, Y., Lin, G.: Evolutionary programming made faster. IEEE Trans. Evol. Comput. **3**(2), 82–102 (1999)