A Functional Approach to Parallelize Particle Swarm Optimization

Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio

Resumen— **Bioinspired methods are typically used to face complex problems requiring long execution times. Many of these methods are based on using several simple entities which search for a reasonable solution (somehow) independently. This is the case of Particle Swarm Optimization (PSO), where many simple particles search for the optimum solution by using both their local information and the information of the best solution found so far by any of the other particles. Particles are partially independent, and we can take advantage of this fact to parallelize PSO programs. Unfortunately, it can be tricky and time-consuming for the programmer to provide good parallel implementations for each concrete PSO program. In this paper we introduce a parallel skeleton that, given a sequential PSO implementation, automatically provides a corresponding parallel implementation of it. The skeleton is defined by using a parallel functional programming language.**

Palabras clave— **Particle Swarm Optimization, Parallel Programming, Skeletons, Functional Programming.**

I. Introduction

The higher-order nature of functional languages, where programs (functions) can be treated as any other kind of program data, make them very useful to implement generic programming solutions which can be trivially reused for other problems. This is even clearer in parallel programming. Since programs can trivially refer to themselves, the coordination of parallel subcomputations can be defined by using the same constructions as in the rest of the program. This enables the construction of *skeletons* [1], which are generic implementations of parallel schemes which can be invoked in run-time to immediately obtain a parallelization of a given sequential program. The skeleton defines how subcomputations coordinate in parallel, and the skeleton user just has to define (part of) the subcomputations.

During the last years, several parallel functional languages have been proposed (see e.g. [2], [3], [4], [5], [6]). In this paper we show how to use one of them to simplify the development of parallel versions of Evolutionary Computation methods [7], [8], [9]. In particular, we use the language Eden [3], [10] to create a generic skeleton implementing the parallelization of Particle Swarm Optimization (PSO) [11], [12], [13], [14], [15], which is one of the most essential methods of Evolutionary Computation. However, the main ideas presented in the paper can also be applied to deal with other evolutionary methods. In fact, this work continues our project to develop Eden skeletons for several kinds of Evolutionary Computation methods, which began in [16] by the introduction of an skeleton to parallelize genetic algorithms [17]. By constructing a library of Evolutionary Computation skeletons, we will provide programmers with a quick procedure to compare, in the same environment, different parallel evolutionary strategies for a given target problem. In particular, programmers (even those working with other languages) will be endowed with a tool to (a) very quickly write different parallel evolutionary solutions for a given target problem; (b) test them; and (c) choose the best one based on empirical results. After that, the programmer may keep using Eden, or she can implement the parallel strategy which provided best results or had the best performance in her favorite parallel language.

We have developed several Eden skeletons to parallelize Particle Swarm Optimization. Due to the lack of space, only two of them are presented in this paper. However, these two parallel generic solutions illustrate how skeletons allow us to quickly compare not only different parallel evolutionary computation methods, but also different strategies to parallelize the *same* evolutionary computation method. In particular, one of the skeletons delays the updating of the best solution found so far to some specific times, thus reducing the communication overhead and improving the performance in many situations.

The rest of the paper is structured as follows. Next we introduce the main features of the functional language Haskell. In Section III we briefly describe the Haskell-based parallel language Eden. In Section IV we introduce the Particle Swarm Optimization method. Then, in Section V, we present how to develop a higher-order sequential Haskell function dealing with particle swarm optimization algorithms. Afterwards, in Section VI we introduce a simple parallel version of such higher-order function, whereas this skeleton is modified in Section VII. Finally, Section VIII contains our conclusions and lines for future work.

II. Introduction to Haskell

In this section we briefly introduce the main peculiarities of the functional language Haskell, which is the base of the parallel language Eden.

One advantage of pure functional languages is that the absence of side-effects allow them to offer a clear semantic framework to analyze the correctness of

Dpto. Sistemas Informáticos y Computación, Facultad Informática, Universidad Complutense de Madrid, Spain. E-mails: prabanal@fdi.ucm.es,{isrodrig,fernando}@sip.ucm.es

programs. In particular, the semantics of the parallel language we will use is clearly defined in [18], and it is simple to relate it with the concrete parallel programs developed by the user [19]. The core notion of functional programming is the mathematical function, that is, a program is a function. Starting with simple basic functions and by using functional composition, complex programs are created. Haskell [20] is the *de facto* standard of the lazy-evaluation[1] functional programming community. It is a strongly typed language including polymorfism, higher-order programming facilities and lazy order of evaluation of expressions.

As it can be expected, the language provides large libraries of predefined functions, and also predefined data types for the most common cases, including lists. Let us remark that Haskell provides polymorfism. Thus, data types can depend on other types. New functions can be defined by analyzing cases on the structure of the data types. For instance, the following function computes the total number of elements of any list (of any concrete type) by using *pattern matching*:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

The first line of the definition is optional, and it represents the type declaration of the function: Given a list of any concrete type `a`, it returns an integer. The rest of the definition represents the definition of the behavior of the function: If it receives an empty list, then it returns 0; otherwise, it adds 1 to the length of the tail of the list.

Other powerful characteristic of Haskell is higher-order. It means that functions can be arguments of functions. For instance, the following predefined function `map` receives as input a function `f` and a list, and then it applies function `f` to every element of the list:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Notice that the type declaration of function `map` indicates that its first parameter has type `a->b`, denoting that the first parameter is a function that receives values of type `a` and returns values of type `b`. The second parameter is a list of elements of type `a`, and the result is again a list, but in this case of elements of type `b`.

Notice that in higher-order languages like Haskell, it is also possible to deal with partially applied functions. For instance, `map` can take as its functional argument a partial application of function `(+)`:

```
mapPlusOne :: [Int] -> [Int]
mapPlusOne xs = map (1+) xs
```

Thus, it adds one to each element of the list.

[1]Lazy evaluation is an evaluation strategy which delays the evaluation of each expression until the exact moment when it is actually required.

III. Introduction to Eden

Eden [10], [19] 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.

Let us remark that, 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.

A. Eden Skeletons

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 as higher order functions. Next, we illustrate, by using a simple example, how skeletons can be written in Eden. More complex skeletons can be found in [10].

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] `using` spine
  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. The `spine` strategy (see ([21] for details) is used to eagerly evaluate the spine of the process instantiation list. In this way, all processes are immediately created. Otherwise, they would only be created on demand.

Let us remark 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.

IV. Introduction to Particle Swarm Optimization

Particle Swarm Optimization [11], [12], [13], [14], [15] (from now on PSO) is a metaheuristic inspired on the social behavior of flocks of birds when flying and on the movement of shoals of fish. A population of entities moves in the search space during the execution of the algorithm. These entities are very simple and perform local interactions (including interactions with the environment). The result of combining simple behaviors is the emergence of complex behaviors and the ability to get good results as a team.

It is assumed that the flock searches for *food* in an area and that there is only one piece of food in the area (the problem objective). It is also assumed that birds in the flock do not know where the food is, but they know the distance to it, so a good strategy to find the food is to follow the bird that is closest to it. PSO emulates this behavior to solve optimization problems. Each solution or particle is a bird in the search space that is always moving and never dies. In fact, we can consider that the swarm is a multi-agent system where particles are simple agents that move through the search space and store (and possibly communicate) the best solution found.

Each element of the swarm has a fitness value, a position and a velocity vector that directs its flight. The movement of particles is partially guided by the best solution found by all particles. Concretely, a particle sw_i is composed of four vectors: (a) The vector x_i, which stores the current position (location) of the particle in the search space; (b) the vector v_i, which stores the gradient (direction) according to which the particle moves; (c) the p_i vector, which stores the location of the best solution found by the particle so far; and (d) the g vector, which stores the location of the best solution found by the swarm ant it is common to all entities. It also has three fitness values: (i) The $x_i^{fitness}$, which stores the fitness of the current solution (x_i vector); (ii) the $p_i^{fitness}$, which stores the fitness of the best local solution (p_i vector); and (iii) the $g^{fitness}$, which stores the fitness of the best global solution (g vector).

The movement of each particle sw_i depends on the best solution it has found since the algorithm started, p_i, and the best solution found by all particles in the entire cloud of particles, g. The speed v_i of sw_i is changed at each iteration of the algorithm to bring it closer to positions p_i and g.

Given a swarm sw where S is the number of particles in sw, the goal is to minimize or maximize a function $f : \mathbb{R}^n \to \mathbb{R}$ (hereafter we will assume that the goal is minimizing it). A basic PSO algorithm follows:

(Initialization)

```
For i = 1,...,S Do
   initPosition(i)
   initBestLocal(i)
   If i = 1 Then initBestGlobal()
   If improvedGlobal(i) Then
      updateBestGlobal(i)
   initVelocity(i)
End For
```

(While loop)

```
While not endingCondition()
   For i = 1,...,S Do
      createRnd(rp,rg)
      updateVelocity(i,rp,rg)
      updatePosition(i)
      If improvedLocal(i) Then
         updateBestLocal(i)
      If improvedGlobal(i) Then
         updateBestGlobal(i)
   End For
End While
```

We comment on the algorithm step by step. First, there is an initialization where all $sw_i \in sw$ (first `For` loop) are initialized as it is described next. In `initPosition(i)`, the position of particle sw_i is randomly created and its position x_i is chosen from the range $[bot, up]$ where bot and up are the lower and upper bounds of the search space. Additionally, we compute $x_i^{fitness} = f(x_i)$. In the `initBestLocal(i)` phase, we set $p_i = x_i$ and $p_i^{fitness} = x_i^{fitness}$. The next instruction, `initBestGlobal()`, is only executed when $i = 0$ and initializes $g = x_1$ and $g^{fitness} = x_1^{fitness}$. If `improvedGlobal()`$= True$, that is, if $p_i^{fitness} < g^{fitness}$ then the best global solution is updated in `updateBestGlobal(i)` where

the following operations are performed: $g = p_i$ and $g^{fitness} = p_i^{fitness}$. Finally, in the `initVelocity(`$i$`)` phase, the particle's velocity is randomly established in the range $[-|up - bot|, |up - bot|]$.

After the initialization, the core of the PSO method is executed until the `endingCondition()` is satisfied (`While` loop). The `endingCondition()` can limit the number of iterations performed, the time consumed or the fitness adequation. In the body of the `While` loop (the `For` loop), all particles are updated. The first step in the `For` loop, `createRnd(rp,rg)`, creates two random values (rp and rg) in the range [0,1], that will be used in the next phase. Then, in `updateVelocity(`i`,rp,rg)`, the velocity of sw_i is updated in the following way: $v_i = \omega \cdot v_i + \phi p \cdot rp \cdot (p_i - x_i) + \phi g \cdot rg \cdot (g - x_i)$ where ω, ϕp and ϕr are parameters that are experimentally set (see [22], [23], [15] for further details) and control the behavior and efficacy of the PSO method. The next step, `updatePosition(`i`)`, updates the position of sw_i: $x_i = x_i + v_i$ and the fitness value $x_i^{fitness} = f(x_i)$. At the end of the loop, if `improvedLocal()`$= True$, that is, if $x_i^{fitness} < p_i^{fitness}$ then the best local solution is updated in `updateBestLocal(`i`)` as follows: $p_i = x_i$ and $p_i^{fitness} = x_i^{fitness}$. In the last step, if `improvedGlobal()` is satisfied then the global solution is updated in the `updateBestGlobal(`i`)` phase as we explained above.

V. Generic PSO Algorithms in Haskell

Let us remark that Haskell functions are first-class citizens. Thus, they can be used as parameters of other functions. In particular, in order to provide a general scheme to deal with PSO algorithms, we need to use as parameter the concrete fitness function used in the problem. In addition to the fitness function, we also need to consider other parameters like the number of particles to be used, the number of iterations to be executed by the program, the boundings of the search space, and the parameters ω, ϕp and ϕr described in Section IV. 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. Taking into account these considerations, the type of the higher-order Haskell function dealing with PSO algorithms is the following:

```
pso :: RandomGen a => a          --Random generator
       -> WPGparams              --ω, φ p and φ r
       -> Int                    --Number of particles
       -> Int                    --Number of iterations
       -> (Position -> Double)   --Fitness function
       -> Boundings              --Search space bounds
       -> (Double,Position)      --Result: Best fitness
                                 --        and position
```

Note that the type `Position` should be able to deal with arbitrary dimensions. Thus, the simplest and more general solution to define a position is to use a list of real numbers, where the length of the list is equal to the number of dimensions of the search space. Analogously, the type `Boundings` should contain a list of lower and upper bounds for each of the dimensions. Thus, it can be described by a list of pairs of real numbers. Finally, the type `WPGparams` is even simpler, as it only contains a tuple of three real numbers describing the concrete values of parameters ω, ϕp and ϕr:

```
type Position = [Double]
type Boundings = [(Double,Double)]
type WPGparams = (Double, Double, Double) --ω,φ p,φ r
```

Once the input parameters are defined, it is time to define the body of the function. First, we use function `initialize` to randomly create the list of initial particles. Its definition (not shown) is simple, as it is only necessary to create `np` particles randomly distributed inside the search space defined by the boundings `bo`. Once particles are initialized, we use an independent function `pso'` to deal with all the iterations of the PSO algorithm. As in the case of the main function `pso`, the auxiliary function `pso'` will also needs a way to introduce randomness, in order to have appropriate random values for rp and rg in the range [0,1]. This problem is solved by function makeRss, that creates all the random values needed.

```
pso sg wpg np it f bo
  = obtainBest (pso' rss wpg it f bo initParticles)
  where initParticles = initialize sg np bo f
        rss = makeRss np (randomRs (0,1) sg)

-- Particle: Best local value, best global value,
-- current position and speed, best local position,
-- best global position
type Particle = (Double,Double,Position,Speed,
                 Position,Position)
type Speed = [Double]
```

Note that each particle contains its position, its velocity (that will also be randomly initialized), the best solution found by the particle, and the best solution found by any particle.

Function `pso'` has a simple definition. It runs recursively on the number of iterations. If there is not any iteration to be performed then we just return as result the input particles. Otherwise, we apply one iteration step (by using an auxiliary function `oneStep`) and then we recursively call our generic scheme with one iteration less:

```
pso' _ _ 0 _ _ pas = pas
pso' (rs:rss) wpg it f bo pas
  = pso' rss wpg (it-1) f bo (oneStep rs wpg f bo pas)
```

Let us now concentrate on how to perform each step. First, each particle is updated. This is done by function `updateParticle`, that applies the equations described in Section IV to update the velocity and position. Moreover, this function also checks if the fitness of the new position is better than the best solution found so far. Thus, the output of the function is a list of pairs (Bool,Particle) where the

boolean value is True iff the new position of the particle is best than the best position found globally so far. Once all the particles have been updated, we check if any of them have improved the best solution or not. If some particles have improved the best solution, we sort the new best solutions and select the new best one. Finally, all the particles are updated with the information about the new best global position. The next code shows how each step is done:

```
oneStep :: [(Double,Double)] -> WPGparams ->
           (Position -> Double) -> Boundings
            -> [Particle] -> [Particle]
oneStep rs wpg f bo pas
  | null newBests = newPas
  | otherwise = map (updateGlobalBest newBvBp) newPas
 where
  newBsPas = zipWith (updateParticle wpg f bo) rs pas
  newPas   = map snd newBsPas
  newBests = qsort (map snd (filter fst newBsPas))
  newBvBp  = obtainBest newBests

updateParticle :: WPGparams -> (Position -> Double)
                  -> Boundings -> (Double,Double)
                  -> Particle  -> (Bool,Particle)
updateParticle (w,p,g) f bo (rp,rg)
               (blv,bgv,po,s,blp,bgp)
  | newFit < bgv = (True,(newFit,newFit,newPos,
                          newSpeed,newPos,newPos))
  | newFit < blv = (False,(newFit,bgv,newPos,
                           newSpeed,newPos,bgp))
  | otherwise    = (False,(blv,bgv,newPos,
                           newSpeed,blp,bgp))
  where newSpeed = w*&s +& p*rp*&(blp-&po)
                        +& g*rg*&(bgp-&po)
        newPos = limitRange bo (po +& newSpeed)
        newFit = f newPos
```

where `+&` and `-&` perform the sum and difference on arbitrary large vectors, while `*&` multiplies all the elements of a vector by a single real number. Their definition is completely trivial:

```
infixl 7 *&
x *& xs = map (x*) xs

infixl 6 -&
xs -& ys = zipWith (-) xs ys

infixl 6 +&
xs +& ys = zipWith (+) xs ys
```

Note that the higher-order nature of Haskell makes it simple to define a generic function dealing with PSO algorithms, as the fitness function can be passed as input parameter to the main `pso` function. Moreover, it is also easy to define fitness functions over arbitrary large dimensions. For instance, let us consider the following function:

$$f(x) = \sum_{i=1}^{n} -x_i sin(\sqrt{|x_i|})$$

This function was considered in [24] as a specially hard to optimize function, because the number of local minima increases exponentially with the number of dimensions of the searching space. In particular, they used 30 dimensions using as boundings for each dimension the interval $[-500, 500]$. We can trivially codify such example in Haskell as follows:

```
fit :: Position -> Double
fit xs = sum (map fit' xs)
  where fit' xi = -xi * sin (sqrt (abs xi))

bo :: Boundings
bo = replicate 30 (-500,500)
```

VI. A Simple Parallel Skeleton

It is well known that it is only possible to obtain good speedups in a parallelization if we can identify time-consuming tasks that can be executed indepentdly. In the case of PSO, the most obvious work that can be divided into independent tasks is the update of velocities and positions of the list of particles, as well as the computation of the fitness function for the new positions. Although the time needed to execute these taks will usually be small, it will be executed many times and, in practice, most of the execution time of the program will be devoted to these tasks.

As we have to apply the same operations to a list of particles, this can be done in Eden by using the skeleton `map_par`. Thus, the only piece of code that have to be modified is the call to function `updateParticle` inside function `oneStep`. Now, instead of sequentially calling to `updateParticle` for each particle, we use the parallel skeleton `map_par` to create an independent process to deal with the operation for each particle:

```
oneStep rs wpg f bo pas
  | null newBests = newPas
  | otherwise = map (updateGlobalBest newBvBp) newPas
 where
  newBsPas= map_par (uncurry (updateParticle wpg f bo))
                    (zip rs pas)
  newBsPas = zipWith (updateParticle wpg f bo) rs pas
  newPas   = map snd newBsPas
  newBests = qsort (map snd (filter fst newBsPas))
  newBvBp  = obtainBest newBests
```

In the most common case, the number of particles will be much higher than the number of available processors. Thus, it would be more efficient to create only as many processes as processors available, and to fairly distribute the population among them. This can be done by substituting `map_par` by a call to `map_farm`:

```
...
  newBsPas=map_farm noPe
                    (uncurry (updateParticle wpg f bo))
                    (zip rs pas)
...
```

where `noPe` is an Eden variable equals to the number of available processors in the system, while `map_farm` implements the idea of distributing a large list of tasks among a reduced number of processes. The implementation firstly distributes the tasks among the processes, producing a list of lists where each inner list will be executed by an independent process. Then, it applies `map_par`, and finally it collects the

results joining the list of lists of results into a single list of results. Notice that, due to the laziness, these three tasks are not done sequentially, but in interleaving. As soon as any worker computes one of its outputs, it sends this subresult to the main process, and it continues computing the next element of the output list. Notice that the communications are asynchronous, so that it is not necessary to wait for acknowledgments from the main process. When the main process has received all the needed results, it finishes the computation. The Eden source code of this skeleton is shown below, where not only the number of processors `np` but also the distribution and collection functions (`unshuffle` and `shuffle` respectively) are also parameters of the skeleton:

```
map_farmG np unshuffle shuffle f xs
  = shuffle (map_par (map f) (unshuffle np xs))
```

Different strategies to split the work into the different processes can be used provided that, for every list `xs`, `(shuffle (unshuffle np xs)) == xs`. In our case, we will use a concrete version of `map_farmG` called `map_farm` where the functions used to unshuffle/shuffle distribute the tasks in a round-robin way.

VII. A Parallel Skeleton reducing Communications

One of the advantages of skeleton-based parallel programming is that several different parallel implementations can be provided for a single sequential specification. Thus, depending on the concrete problem to be solved and depending on the concrete parallel architecture available, the programmer can choose the implementation that better fits his necessities. In this section we show a different and more sofisticated implementation of the PSO skeleton.

In many situations, the implementation presented in the previous section will obtain poor speedups. In particular, in case the processors of the underlying architecture do not use shared memory, lots of communications will be needed among processes, dramatically reducing the speedup. The solution to this problem implies increasing the granularity of the tasks to be performed by each process. In particular, we can increase the granularity by splitting the list of particles into as many groups as processors available. Then, each group evolves in parallel independently during a given number of iterations. After that, the processes communicate among them to compute the new best overall position, 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 are performed.

In order to implement in Eden a generic skeleton dealing with this idea, we need to pass new parameters to function `pso`. In particular, we need a new parameter `pit` to indicate how many iterations have to be performed independently in parallel before communicating with the rest of processes. Besides, the parameter `it` will now indicate the number of parallel iterations to be executed, that is, the total number of iterations will be `it * pit`. Finally, we can add an extra parameter `nPE` to indicate the number of processes that we want to create (typically, it will be equal to the number of available processors). Thus, the new type of function `pso` will be as follows:

```
psoPAR ::
  RandomGen a => a   -- Random generator
    -> WPGparams     -- ω, φ p and φ r parameters
    -> Int           -- Number of particles
    -> Int           -- Iterations per parallel step
    -> Int           -- Number of parallel iterations
    -> Int           -- Number of parallel processes
    -> (Position -> Double) -- Fitness function
    -> Boundings            -- Search space bounds
    -> (Double,Position)    -- Result: Best fitness
                            --         and position
```

Before explaining how to implement the main function `pso`, let us consider how to define the function describing the behaviour of each of the processes of the system. In addition to receive a parameter for creating random values, each process needs to receive a parameter with ω, ϕp and ϕr values, the number of iterations to be performed in each parallel step, the fitness function, and the boundings of the search space. Besides, it also receives through an input channel a list containing the initial particles assigned to the process. Finally, it also receives through another channel a list containing the best overall positions found by the swarm after each parallel 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 value through its second input channel right before starting to compute a new parallel step. The output of the process is a list containing the best solution found after each parallel step of the process. Summarizing, the type of the processes is the following:

```
psoP ::
  RandomGen a => a        -- Random generator
  -> WPGparams            -- ω, φ p and φ r parameters
  -> Int                --Iterations per parallel step
  -> (Position -> Double) -- Fitness function
  -> Boundings            -- Search space bounds
  -> ([Particle],         -- Initial local particles
      [(Double,Position)])-- Best overall solutions
  -> [(Double,Position)]  -- Out: Best local solutions
```

Note that we return both the best fitness values and the corresponding positions. Let us now consider how to implement the process definition. It is defined recursively on the structure of the second input channel. That is, when the main function finishes sending the list of new best values through the input channel, the process also finishes its execution. Each time a new value is received, we update the local particles with the information about the new global best solution (if it is better than the best local solution). Then, we use exactly the same function `pso'` used in the sequential case, so we can perform `pit` iterations over the current particles. Finally,

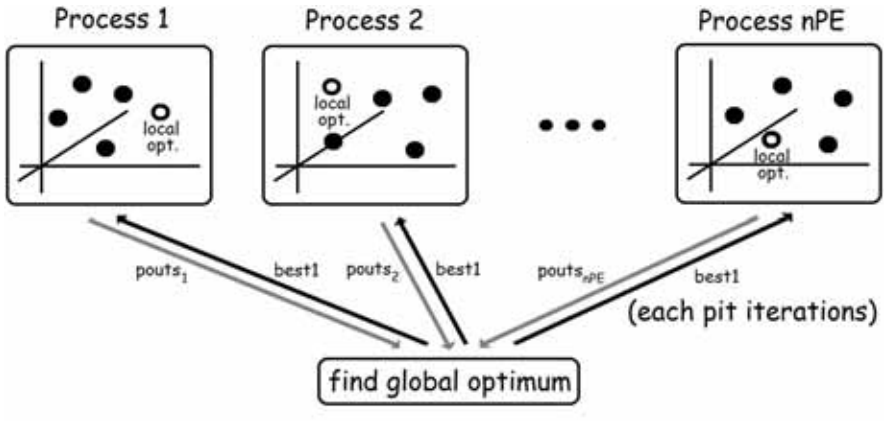

Fig. 1. Basic structure of the second skeleton.

we send through the output channel the best value `newOut` computed locally in this parallel step, and we perform the recursive call to go on working in the next parallel step.

```
psoP sg wpg pit f bo (pas,[]) = []
psoP sg wpg pit f bo (pas,newBest:newBests)
  = newOut : psoP sg2 wpg pit f bo (newPas,newBests)
 where
   rss = makeRss (length pas) (randomRs (0,1) sg1)
   (sg1,sg2)=split sg
   pas' = if newBest < oldBest
            then map (updateGlobalBest newBest) pas
            else pas
   newPas = pso' rss wpg pit f bo pas'
   newOut = obtainBest newPas
   oldBest = obtainBest pas
```

Finally, we have to define the behaviour of the main function `pso`. The first step it must perform is the same as in the sequential case, that is, particles are initialized using the same sequential function `initialize`. Then particles are fairly unshuffled among `nPE` lists, generating the list of lists of particles `pass`, so that each inner list can be sent to a different process. As a technical functional detail, we also need to create a list of random generators `sgs`, one for each process.

The core of function `pso` is the creation of `nPE` independent processes. Each of them receives its corresponding element from the list of random generators `sgs`, and the corresponding list from the list of lists of initial particles `pass`. Moreover, all the processes also receive through their last input channel the list `best1` of the best overall solutions found after each parallel step (see Figure 1). The first element of `best1` corresponds to the best value obtained from the list of initial random particles, while the rest of elements of the list are obtained from the output of the processes (`pouts`). Note that Haskell uses lazy evaluation, so that it can easily deal with mutually recursive definitions like the ones relating `pouts` and `bests1`. Finally, note also that `pouts` returns the list of outputs of each process. Thus, each list of `pouts` contains outputs from a single process, while in order to compute `bests` we need to deal with all the first outputs of all the processes together in a single list. Hence, to compute `bests` we first transpose the list of lists `pouts` (by using function `transp`). Then, each list is sorted and the best value is obtained. By doing so, each element of `bests` contains the best overall value computed after each parallel step.

The source code dealing with the creation of the processes is the following:

```
psoPAR sg wpg np pit it nPE f bo = last bests   where
  initParticles = initialize sg np bo f

  pass = unshuffle nPE initParticles
  sgs  = tail (generateSGs (nPE+1) sg)

  pouts :: [ [(Double,Position)] ]
  pouts = [process (psoP (sgs!!i) wpg pit f bo)
           #(pass!!i,bests1) |i<-[0..nPE-1]]
             `using` spine

  bests,bests1 :: [(Double,Position)]
  bests  = map (head.sort) (transp pouts)
  bests1 = take it (obtainBest initParticles : bests)
```

Let us remark that in order to convert a sequential PSO algorithm into the corresponding parallel program, the programmer only has to change a call to function `pso` by a call to function `psoPAR` indicating appropriate values for parameters `pit`, `it` and `nPE`. In fact, the only programmer effort will be related to selecting a reasonable value for `pit`.

Note that the programmer does not need to deal with the details of the parallelization. In fact, it is not even necessary that the programmer understands the details of the parallel implementation, provided that he understands the type interface of function `psoPAR`. Actually, the only function to be developed by the programmer is the fitness function. Moreover, it is important to recall that Eden programs can interact with other programming languages. In particular, C code can be encapsulated inside a Haskell function. Hence, Eden can be used as a coordination language dealing with the parallel structure of the program, while the core of the fitness function could be implemented in a computation language like C.

Let us finally comment that it is possible to provide more versions of the PSO skeleton (not shown due to lack of space), so the programmer can select the one that better fits his necessities. For instance, it is easy to change our skeleton to provide another version where the number of iterations to be performed in each parallel step is variable. Let us note that, during the first phases, the global optimum changes faster than during the last iterations. Thus, during the last phases we could use a larger number of independent parallel iterations, as synchronization seems to be less important when the global optimum does not change significantly.

VIII. Conclusions and Future Work

We have shown how to use the parallel functional language Eden to provide parallel versions of PSO algorithms by means of Eden skeletons. The programmer only has to provide an appropriate fitness function and adjustment parameters, while all the low-level details of the parallelization are done automatically by the skeleton and the underlying runtime system. Moreover, different parallel implementations of PSO can be provided to the user (in the paper we have shown two of them), so he can select the one that better fits his necessities.

In our previous work [16], we showed how to use Eden skeletons to deal with the parallelization of genetic algorithms. In fact, our aim is to provide a library of generic Haskell versions of the most common bioinspired metaheuristics, as well as a library containing the corresponding Eden skeletons for each metaheuristic. By doing so, we will simplify the task of using these metaheuristics in functional settings, and we will also simplify the task of improving their performance by means of parallelizing them. Moreover, as several metaheuristics will be provided in the same environment, the programmer will be able to check more easily what metaheuristics fits better for each concrete problem.

In addition to extend our library to deal with other evolutionary computation methods (like Ant Colony Optimization or River Formation Dynamics), as future work we are particularly interested in studying hybrid systems combining different metaheuristics.

Acknowledgments

This research has been partially supported by projects TIN2009-14312-C02-01 and UCM-BSCH GR35/10A - group number 910606.

Referencias

[1] M. Cole, "Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, pp. 389–406, 2004.

[2] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones, "GUM: a portable parallel implementation of Haskell," in *Programming Language Design and Implementation, PLDI'96*. 1996, pp. 79–88, ACM Press.

[3] U. Klusik, R. Loogen, S. Priebe, and F. Rubio, "Implementation skeletons in Eden: Low-effort parallel programming," in *Implementation of Functional Languages, IFL'00*. 2001, LNCS 2011, pp. 71–88, Springer.

[4] N. Scaife, Horiguchi S., G. Michaelson, and P. Bristow, "A parallel SML compiler based on algorithmic skeletons," *Journal of Functional Programming*, vol. 15, no. 4, pp. 615–650, 2005.

[5] S. Marlow, S. L. Peyton Jones, and S. Singh, "Runtime support for multicore Haskell," in *International Conference on Functional Programming, ICFP'09*. 2009, pp. 65–78, ACM Press.

[6] G. Keller, M.T. Chakravarty, R. Leshchinskiy, S.L. Peyton Jones, and B. Lippmeier, "Regular, shape-polymorphic, parallel arrays in Haskell," in *International Conference on Functional Programming, ICFP'10*. 2010, pp. 261–272, ACM.

[7] A.E. Eiben and J.E. Smith, *Introduction to Evolutionary Computing*, Springer, 2003.

[8] K. de Jong, "Evolutionary computation: a unified approach," in *Genetic and Evolutionary Computation Conference, GECCO'08*. 2008, pp. 2245–2258, ACM.

[9] R. Chiong, Ed., *Nature-Inspired Algorithms for Optimisation*, vol. 193 of *Studies in Computational Intelligence*, Springer, 2009.

[10] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio, "Parallelism abstractions in Eden," in *Patterns and Skeletons for Parallel and Distributed Computing*, F. A. Rabhi and S. Gorlatch, Eds. 2002, pp. 95–128, Springer.

[11] J. Kennedy and R.C. Eberhart, "Particle swarm optimization," in *IEEE International Conference on Neural Networks*. 1995, vol. 4, pp. 1942–1948, IEEE Computer Society Press.

[12] Y. Shi and R.C. Eberhart, "A modified particle swarm optimizer," in *IEEE International Conference on Evolutionary Computation*. 1998, pp. 69–73, IEEE Computer Society Press.

[13] M. Clerc and J. Kennedy, "The particle swarm - explosion, stability, and convergence in a multidimensional complex space," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 1, pp. 58–73, 2002.

[14] K.E. Parsopoulos and M.N. Vrahatis, "Recent approaches to global optimization problems through particle swarm optimization," *Natural Computing*, vol. 1, no. 2-3, pp. 235–306, 2002.

[15] M.E.H. Pedersen, *Tuning & Simplifying Heuristical Optimization*, Ph.D. thesis, University of Southampton, School of Engineering Sciences, 2010.

[16] A. de la Encina, M. Hidalgo-Herrero, P. Rabanal, and F. Rubio, "A parallel skeleton for genetic algorithms," in *Int. Work-Conference on Artificial Neural Networks, IWANN 2011*. 2011, vol. 6692 of *LNCS*, pp. 388–395, Springer.

[17] D.E. Goldberg, *Genetic Algorithms in Search, Optimisation and Machine Learning*, Addison-Wesley, 1989.

[18] M. Hidalgo-Herrero and Y. Ortega-Mallén, "Continuation semantics for parallel Haskell dialects," in *First Asian Symposium on Programming Languages and Systems (APLAS'03)*. 2003, LNCS 1058, pp. 303–321, Springer.

[19] M. Hidalgo-Herrero, Y. Ortega-Mallén, and F. Rubio, "Analyzing the influence of mixed evaluation on the performance of Eden skeletons," *Parallel Computing*, vol. 32, no. 7-8, pp. 523–538, 2006.

[20] S. L. Peyton Jones and J. Hughes, "Report on the programming language Haskell 98," Tech. Rep., Feb. 1999, `http://www.haskell.org`.

[21] P. W. Trinder, K. Hammond, H-W. Loidl, and S. L. Peyton Jones, "Algorithm + Strategy = Parallelism," *Journal of Functional Programming*, vol. 8, no. 1, pp. 23–60, 1998.

[22] Y. Shi and R.C. Eberhart, "Parameter selection in particle swarm optimization," in *Evolutionary Programming*. 1998, vol. 1447 of *LNCS*, pp. 591–600, Springer.

[23] T. Beielstein, K.E. Parsopoulos, and M.N. Vrahatis, "Tuning PSO parameters through sensitivity analysis," Tech. Rep., 2002.

[24] X. Yao, Y. Liu, and G. Lin, "Evolutionary programming made faster," *IEEE Trans. Evolutionary Computation*, vol. 3, no. 2, pp. 82–102, 1999.