Implementing Parallel Genetic Algorithm Using Concurrent-functional Languages

J. Albert-Cruz[1], J. J. Merelo[2], L. Acevedo-Martínez[1] and Paloma de las Cuevas[2]
[1]*Centro de Estudios de Matemática Computacional, Universidad de Ciencias Informáticas, La Habana, Cuba*
[2]*Dept. Arquitectura y Tecnología de los Computadores, Universidad de Granada, Granada, Spain*

Keywords: Parallel Evolutionary Algorithms, Functional Languages, Concurrent Languages, Implementation, Modeling.

Abstract: The spread of multiprocessor and multi-core architectures have a pervasive effect on the way software is developed. In order to take full advantage of them, a parallel implementation of every single program would be needed, but also a radical reformulation of the algorithms that are more appropriate to that kind of implementation. In this work we design and implement an evolutionary computation model using programming languages with built-in concurrent concepts. This article shows the advantages of these paradigms in order to implement a parallel genetic algorithm (pGA) with an island pools based topology in the concurrent-functional oriented programming languages: Erlang, Scala, and Clojure. Some implementation decisions are analyzed and the results of the solution of a study case are shown.

1 INTRODUCTION

Genetic algorithms (GA) (Goldberg, 1989) are currently one of the most used meta-heuristics to solve engineering problems. Furthermore, parallel genetic algorithms (pGAs) are useful to find solutions of complex optimizations problems in adequate times (Luque and Alba, 2011); in particular, problems with complex fitness. Some authors (Alba and Troya, 2001) state that using pGAs improves the quality of solutions in terms of the number of evaluations needed to find one. This reason, together with the improvement in evaluation time brought by the simultaneous running in several nodes, have made parallel and distributed evolutionary algorithms a popular methodology.

Running evolutionary algorithms in parallel is quite straightforward, but programming paradigms used for the implementation of such algorithms is far from being an object of study. Object oriented or procedural languages like Java and C/C++ are mostly used. Even when some researchers show that implementation matters (Merelo-Guervós et al., 2011), parallels approaches in new languages/paradigms is not normally seen as a land for scientific improvements.

New parallel platforms have been identified as new trends in pGAs (Luque and Alba, 2011), however only hardware is considered. Software platforms, specifically programming languages, remain poorly explored; only Ada (Santos, 2002) and Erlang (A. Bienz and Thede, 2011; Kerdprasop and Kerdprasop, 2013) were slightly tested.

This work explores the advantages of some non mainstream languages (not included in the top ten of any most popular languages ranking) with concurrent and functional features in order to develop GAs in its parallel versions. It is motivated by the lack of community attention on the subject and the belief that using concepts that simplify the modeling and implementation of such algorithms might promote their use in research and in practice.

This research is intended to show some possible areas of improvement on architecture and engineering best practices for concurrent-functional paradigms, as was made for Object Oriented Programming languages (Merelo-Guervós et al., 2000), by focusing on pGAs as a domain of application and describing how their principal traits can be modeled by means of concurrent-functional languages constructs. We are continuing the research reported in (Cruz et al., 2013; Albert-Cruz et al., 2013).

The rest of the paper is organized as follows. Next section presents the state of the art in concurrent and functional programming language paradigms and its potential use for implementing pGAs. Our proposal to adapt pGAs to the paradigms using a study case is explained in section 3.1 as well as the experimental results in section 3.2. In section 3.3 we show a sam-

Cruz J., Merelo J., Acevedo-Martínez L. and Cuevas P..
Implementing Parallel Genetic Algorithm Using Concurrent-functional Languages.
DOI: 10.5220/0005036601690175
In *Proceedings of the International Conference on Evolutionary Computation Theory and Applications* (ECTA-2014), pages 169-175
ISBN: 978-989-758-052-9

ple program of canonical island/GA implemented in Scala.

Finally, we draw the conclusions and future lines of work in section 4.

2 FUNCTIONAL AND CONCURRENT PROGRAMMING

Developing correct software quickly and efficiently is a never ending goal in the software industry. Novel solutions that try to make a difference providing new abstraction tools outside the mainstream of programming languages have been proposed to pursue this goal; two of the most promising are the functional and the concurrent.

Concurrent Programming

The concurrent programming paradigm (or concurrent oriented programming (Armstrong, 2003)) is characterized by the presence of programming constructs for managing processes like first class objects. That is, with operators for acting upon them and the possibility of using them like parameters or function's result values. This simplifies the coding of concurrent algorithms due to the direct mapping between patterns of communications and processes with language expressions.

Concurrent programming is hard for many reasons, the communication/synchronization between processes is key in the design of such algorithms. One of the best efforts to formalize and simplify that is the Hoares *Communicating Sequential Processes* (Hoare, 1978), this interaction description language is the theoretical support for many libraries and new programming languages.

When a concurrent programming language is used normally it has a particular way of handling units of execution, being independent of the operation system has several advantages: one program in those languages will work the same way on different operating systems. Also they can efficiently manage a lot of processes even on a mono-processor machine.

Functional Programming

Functional programming paradigm, despite its advantages, does not have many followers. Several years ago was used in Genetic Programming (Briggs and O'Neill, 2008; Huelsbergen, 1996; Walsh, 1999) and recently in neuroevolution (Sher, 2013) but in GA its presence is practically nonexistent (Hawkins and Abdallah, 2001).

This paradigm is characterized by the use of functions like first class concepts, and for discouraging the use of state changes. The latter is particularly useful for develop concurrent algorithms in which the communication by state changes is the origin of errors and complexity. Also, functional features like closures and first class functions in general, allow to express in one expression patterns like *observer* which in language like Java need so many lines and files of source code.

Multi-paradigms Emerging Languages

The field of programming languages research is very active in the Computer Science discipline. To find software construction tools with new and better means of algorithms expression is welcome. In the last few years the functional and concurrent paradigms have produced a rich mix in which concepts of the first one had been simplified by the use of the second ones.

Among this new generation, the languages Erlang and Scala have embraced the actor model of concurrency and get excelentes results in many application domains; Clojure is another one with concurrent features such as promises/futures, Software Transaction Memory and agents. All of these tools have processes like built-in types and scale beyond the restrictions of the number of OS-threads.

3 A CONCURRENT AND FUNCTIONAL APPROACH TO PGAS

In order to design the architecture of a software in the GAs application domain, we mostly identify the main concepts involved and the relations among them. Then, using the concepts of the paradigms and programming techniques chosen, we define the structure from the highest levels of abstraction indicating the data to be processed and their flow. The quality and extensibility of that structure might determine the success or failure of the software development.

On the other hand, to develop an optimal codification of an algorithm is mandatory to know every characteristic of the programming language that is being used.

We used an hybrid pGAs (island topology with a pool based pGA in each node) for show the implementation recommendations. The main pGAs components are listed in Table 1. We chose a classical problem, the *Max-SAT* with 100 variable instances (Hoos and Stutzle, 2000).

Table 1: Parallel GA components.

AG Component	Rol	Description
chromosome	Representing the solution.	binary string
evaluated chromosome	Pair {chromosome, fitness}.	relation thats indicate the value of a individual
population	Set of chromosomes.	list
crossover	Relation between two chromosomes producing other two new ones.	crossover's function
mutation	A chromosome modification.	chromosome's change function
selection	Means of population filtering.	selection's function
pool	Shared population among node's calculating units.	population
island	Topology's node.	
migration	Random event for chromosome interchange.	message
evolution	Execution.	A generation is made
evaluation	Execution.	A fitness calculi is made

3.1 Modeling and Implementing in Concurrent Languages

With the initial domain analysis and using the concurrent and functional concepts properly, the following design and implementation were conceived.

Erlang Modeling

The main concurrent concepts are **actor** and **message**, while the functional ones are **function** and **list**.

We propose to use **actors** (the execution units of the language) for independent processes: islands or evaluators/reproducers; for the communication among the actors we use **messages**, which is the concept available in the pattern for that aim.

To express the pGAs logic we propose to use the functions (functional features for data transformation and computation expression). For the data model we propose use lists and tuples (the basic data structures in the functional paradigm).

Scala Modeling

Scala is a programming language with the same concurrent programming pattern (actors) as Erlang. In the Scala implementation we followed the same criteria utilized in Erlang but with differences for its object support and JVM dependence.

Clojure Modeling

The Clojures main concurrent used concepts are **agent**, **ref** and **atom**; the functional ones are **function** and **list**. Clojure is a language with a very strict control of state changes; it demands a clear identification of the code doing it and that is similar to Erlang where the functional purity is pursued too.

Agents were the concept used for implementing the independent units of execution (reproductors, evaluators, and islands). The communication between agents was made by protocols functions due to the needed flexibility. GAs operations, their logic and constraints, were expressed in functions and protocol implementations and the data was encoded in lists and vectors data structures.

Libraries Developed

We developed libraries in Erlang, Scala and Clojure following the same design concepts and it was tested with the study case. The code is open, under AGPL license, at the following addresses:

Erlang https://github.com/jalbertcruz/erlEA/archive/v1.0.tar.gz

Scala https://github.com/jalbertcruz/sclEA/archive/v1.0.tar.gz

Clojure https://github.com/jalbertcruz/cljEA/archive/v1.0.tar.gz

3.2 Application Over the Study Case

All used languages have functional and concurrent built-in features, with the first ones supporting the second ones. Erlang and Scalas implementations are based in the actor pattern for doing parallel computation. Clojure on the other hand works with the agent concept, a similar model with simplified ways of reading the involved information.

To communicate modules we used languages dependent (and different) data types. The message's structure was tuples for Erlang and Scala, and for agents it was necessary to encapsulate functions on protocols (Clojure variants of Java interfaces). For sharing individuals (the pool) we used functionals

Table 2: Erlang constructions.

Erlang Concept	Role
tuple	Data structure for immutable compound data.
list	Sequence data structure for variable length compound data.
function	Data relations, operations.
actor	Execution unit, process.
message	Communication among actors.
ets	Set of chromosome shared by the pool.
random module	Random number generation.

Table 3: Erlang/AG concepts mapping.

Erlang concept	AG concept mapping
tuple	evaluated chromosome
list	chromosomes and populations
function	crossover, mutation and selection
actor	island, evaluator and reproducer
message	migration
ets	pool

Table 4: Scala concepts.

Scala concepts	Role
tuple	Data structure for immutable compound data.
list	Sequence data structure for variable length compound data.
function	Data relations, operations.
Akka's actor	Execution unit, process.
symbol/message	Communication among actors.
HashMap	Set of chromosome shared by the pool.

Table 5: Scala/AG concepts mapping.

Scala concept	AG concept mapping
tuple	evaluated chromosome
list	chromosomes and populations
function	crossover, mutation and selection
Akka's actor	island, evaluator and reproducer
symbol/message	migration
HashMap	pool

consult/modification data structures: hash-like for Scala/Clojure and the *ets* module in Erlangs case. The data was encoded with compound data structures: lists, vectors, tuples, records, etc. The Table 6 summarizes the differences between the languages.

Results

The design was tested with a population of 1024 individuals on each island (two islands were used), doing 5000 evaluations on a dual-core (4 threads) laptop i7-3520M with Windows 8 and 16 Gb of RAM. In order to find the better combinations of evaluators/reproducers, several of them were tested for each technology (evaluators = 1..30 and reproducers = 1..10). In every combination the number of evaluators is greater than the reproducers because the fitness function is more computational intensive than the reproduction execution. 10 runs were used for each combination and then the times with more dispersion were deleted until the standard deviation (SD) remained below the 5 %.

For a *speedup* analysis, using the ideas presented in (Alba, 2002), a sequential implementation with the same data structures and operator's implementations was made. Speedup is the ratio between $E[T_1]$ (sequential implementation average time) and $E[T_m]$ (parallel implementation average time in m processors), the expected value is $m = 4$ in this case (the number of logical processors in the used hardware).

The results shown in Table 7 indicate for each language the best time for the parallel implementation, the combination of evaluators/reproducers in which the parallel variant was obtained, the time for the sequential implementation, a relative speedup (speedup calculated in relation to his sequential time) and the speedup (relative to the best sequential time of all

Table 6: Language concept used for each pGA component.

	Erlang	**Scala**	**Clojure**
Parallel execution unit	actor	actor	agent
Communication (messages)	tuple	tuple	function (protocol)
pool	*ets*	HashMap	hash-map
DS chromosome	list	list	vector
DS population	list	list	lazy list
Compound data	tuple	tuple/object	record/vector
Runtime environment	Erlang VM	Java VM	Java VM

Table 7: Experiment results for the minimum parallel time of all combinations tested.

Language	**Parallel time ± SD (ms)**	**Workers combination**	**Sequential time (ms)**	**Relative speedup**	**Speedup**
Erlang	2920.40 ± 126	25 evaluators, 1 reproducer	8143.3	2.78	0.55
Clojure	1734.66 ± 28.32	10 evaluators, 1 reproducer	3340.22	1.92	0.92
Scala	563 ± 24.32	6 evaluators, 1 reproducer	1651.8	2.86	2.86

implementations, Scala's in this case). Each worker (evaluators and reproducers) is a unit of execution, and in the used hardware only 4 units (at most) can run at the same time.

Figure 1 shows the running times when one reproducer is used with a variant number of evaluators; Figure 2 shows the same but for two reproducers. In both cases the overall behaviour of Scala is better.

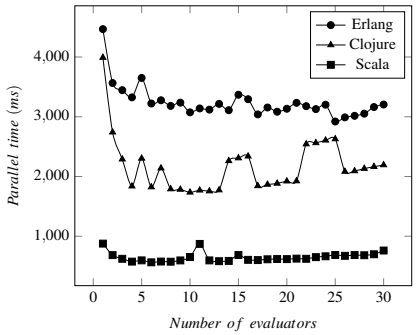

Figure 1: Parallel running times for one reproducer and 0..30 evaluators of hybrid pGAs implementation in Erlang, Scala and Clojure.

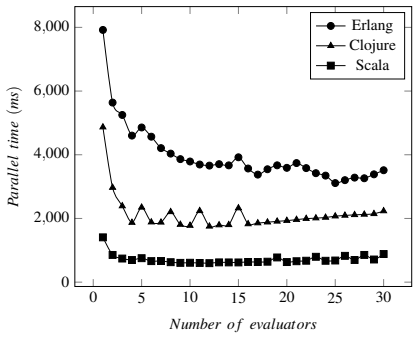

Figure 2: Parallel running times for two reproducers and 0..30 evaluators of hybrid pGAs implementation in Erlang, Scala and Clojure.

The computation complexity of the evaluation function is greater than the reproduction phase and this is why the results when one reproducer was used are better than when two reproducers were used.

Figures 1 and 2 show that the three languages have a good concurrent behaviour: the overhead of managing more logical execution units than the available physical ones did not show any impact on the execution time of the algorithm, even when that number gradually increases.

The Scala implementation is smoother in its results in contrast with Clojure where many peaks were obtained. These two languages use the JVM and the same random library, however there are clear differences in their concurrent models. The results for Scala and Clojure are better with a small number of units of execution: when the number of evaluators grows the efficiency of the algorithm falls. In this sense Erlang have a non-typical behaviour, improving up to 25 evaluators, and then the speed begins to decrease.

Erlang is the language with the worst execution time; but its runtime, in the best case, is able to schedule 52 units of execution (far more than the others). The Erlang processes are scheduled using SMP whith one scheduler per core. Each process is allowed to run until it is paused to wait for input (a message from some other processes) or until it has executed a maximum fixed number of reductions (each VM instruction has associated a number of reductions). This unique way of scheduling processes yields these particular results and will be better used in next studies. Also the speedup obtained in relationship with his sequential time is very good. These two facts point to a possible good scalability.

Clojure's performance is medium, with a speedup

close to 1. The *send* function was always used to compute the expression by the agents therefore a hardware dependent pool of treats was used.

Scala is the language with best results, even when its runtime is the same of Clojure's. It has a particular model of concurrency (actors on a *event-based* dispatcher supported on the Java JSR166 fork-join pool); and of computation (its balance between mutable and immutable state), allowing the best behaviour of the concurrent algorithm. Again is important to note the quality of the concurrent abstractions made by all these technologies in which the number of logical units of executions is greater than the number of the physical ones.

3.3 Sample Program of Canonical Island/GA in Scala

In order to highlight the simplicity of the code based in the built in concurrent concepts of the selected languages. We will describe a canonical island-based GA implementation in Scala. We take and island-based GA implementation because is simpler parallel GA model than hybrid pGAs we use for time evaluations.

Scala inherit keywords from several languages to express programming concepts: from Java take *extends* for express inheritance and from Python/Ruby take *def* to define methods. In order to help the type inference they have Pascal-like syntax for declaring types.

The listing 1 shows a class declaration: an actor class.

```
class Island extends Actor {
  // Set of actors (workers)
  var workers: Set[ActorRef] = _
  def receive = {
     case 'start =>
      // All executing units to work!
      workers.forEach(_ ! 'start)
  }
}
```

Listing 1: Actor declaration.

Island is an *actor*, it have a set of others actors (the workers reproducers and evaluators). All actors classes should have a *receive* method, this block of code is compound of a list of case instructions: one for each kind of message the actor is able to respond. In this method there is only one case: for send a *'start* message to each worker. All actors have a method of name *!* to send messages to them.

```
// One of the worker classes
class Reproducer extends Actor {
  def receive = {
     case('evolve, pool:HashMap, n:Int)
           =>
     val pop = pool.filter((a: (List,
           (Int, Int))) => a._2._2==2).
           keys.toList.map(i =>
                       (i, pool(i)._1))
       val (res, resultData) =
         Reproducer.evolve(
         Reproducer.extractSubpop(pop,n),
           parentsCount = n / 2 })
         // Continue the iteration with
         // res and resultData
 }
}
```

Listing 2: Functional processing of data.

The listing 2 shows the class for the reproducers, in this case the message processed is composed of a tuple of 3 elements. The first statement apply a filter and a transformation (method *map*) over the pool of individuals.

```
// Creating 4 islands
val islands = for(_ <- 1 to 4)
      yield sys.actorOf(Props[Island])

// Puting the migrants destination &
// start each island
for(i <- 0 to 3){
      islands(i) ! ('migrantsDest,
                    islands((i+1)%4))
      islands(i) ! 'start
}
```

Listing 3: Main code.

Finally the listing 3 create islands and start the evolutions.

4 CONCLUSIONS

This work shows the simplicity of the modeling and implementation of a hybrid parallel genetic algorithm in three different concurrent-functional languages. Most of the developed code is open, under AGPL license, at https://github.com/jalbertcruz/. In particular we described a canonical island-based GA implementation in Scala to show its simplicity using the built-in concurrent concepts. Erlang and Clojure are languages that encourage a *non mutable state-all functional* programming style with advantages in the design and correction of the algorithms. The protocols of Clojure allow the principles of OO without the complications of inheritance; its concurrent concepts are specialized and flexible at the same time. The Scala language is multi-paradigm and hybrid in relation with the computation models supported. When a shared data structure is needed this language allows

a more direct access and that could be an advantage, although this has not been shown in our experiments through the scaling capability.

Among the new trends in pGAs are new parallel platforms, the new languages with built-in concurrent abstractions are parallel platforms too, and their use for developing pGAs can be a very good approach for new GA developments. The functional side, which is present in all of them, is a key component to compose software components and simplifying the communication strategies among concurrent activities. In the pGA model used in this work the chosen GA architecture is concurrent-rich but the implementation remains simple thanks of the high level of abstraction of the implementation technologies.

Our experiments show that the performance of Scala is the best and point to Erlang as a very scalable runtime.

In order to complete the methodology we plan to study others concurrent oriented languages such Go, Haskell, and F# as well as going deeper in other concurrent features of the already studied languages.

We are also planning to enrich the experiments with more complex cases of study and to test the libraries in heterogeneous hardware in order to check the scalability of each language.

REFERENCES

A. Bienz, K. Fokle, Z. K. E. Z. and Thede, S. (2011). A generalized parallel genetic algorithm in Erlang. In *Proceedings of Midstates Conference on Undergraduate Research in Computer Science and Mathematics*.

Alba, E. (2002). Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82:7–13.

Alba, E. and Troya, J. M. (2001). Analyzing Synchronous and Asynchronous Parallel Distributed Genetic Algorithms. *Future Generation Computer Systems - Special issue on bioimpaired solutions to parallel processing problems*, 17.

Albert-Cruz, J., Acevedo-Martínez, L., Merelo, J., Castillo, P., and Arenas, M. (2013). Adaptando algoritmos evolutivos paralelos al lenguaje funcional erlang. *MAEB 2013 - IX Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados*.

Armstrong, J. (2003). Concurrency oriented programming in erlang.

Briggs, F. and O'Neill, M. (2008). Functional genetic programming and exhaustive program search with combinator expressions. *Int. J. Know.-Based Intell. Eng. Syst.*, 12(1):47–68.

Cruz, J. A., Guervós, J. J. M., García, A. M., and de las Cuevas, P. (2013). Adapting evolutionary algorithms to the concurrent functional language Erlang. In *GECCO (Companion)*, pages 1723–1724.

Goldberg, D. E. (1989). *Genetic Algorithms in search, optimization and machine learning*. Addison Wesley.

Hawkins, J. and Abdallah, A. (2001). A generic functional genetic algorithm. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, AICCSA '01, pages 11–, Washington, DC, USA. IEEE Computer Society.

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

Hoos, H. H. and Stutzle, T. (2000). Satlib: An online resource for research on sat. In I.P.Gent, H.v.Maaren, T., editor, *SAT 2000*, pages 283–292, www.satlib.org. IOS Press.

Huelsbergen, L. (1996). Toward simulated evolution of machine-language iteration. In *Proceedings of the First Annual Conference on Genetic Programming*, GECCO '96, pages 315–320, Cambridge, MA, USA. MIT Press.

Kerdprasop, K. and Kerdprasop, N. (2013). Concurrent data mining and genetic computing implemented with erlang language. *International Journal of Software Engineering and Its Applications*, 7(3).

Luque, G. and Alba, E. (2011). *Parallel Genetic Algorithms, Theory and Real World Applications*, chapter Parallel Models for Genetic Algorithms, pages 15–30. Springer-Verlag Berlin Heidelberg.

Merelo-Guervós, J.-J., Arenas, M. G., Carpio, J., Castillo, P., Rivas, V. M., Romero, G., and Schoenauer, M. (2000). Evolving objects. In Wang, P. P., editor, *Proc. JCIS 2000 (Joint Conference on Information Sciences)*, volume I, pages 1083–1086. ISBN: 0-9643456-9-2.

Merelo-Guervós, J.-J., Romero, G., García-Arenas, M., Castillo, P. A., Mora, A.-M., and Jiménez-Laredo, J.-L. (2011). Implementation matters: Programming best practices for evolutionary algorithms. In Cabestany, J., Rojas, I., and Caparrós, G. J., editors, *IWANN (2)*, volume 6692 of *Lecture Notes in Computer Science*, pages 333–340. Springer.

Santos, L. (2002). Evolutionary Computation in Ada95, A Genetic Algorithm approach. *Ada User Journal*, 23(4).

Sher, G. I. (2013). *Handbook of Neuroevolution Through Erlang*. Springer.

Walsh, P. (1999). A functional style and fitness evaluation scheme for inducting high level programs. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1211–1216, Orlando, Florida, USA. Morgan Kaufmann.