Problem 4, Multi-modal and Spacial Distribution Problems

Victor Montiel Argaiz
victor.montielargaiz@gmail.com

March 15, 2012

Contents

Abstract

This paper explores different methods of parallelization of EC which achieve better quality solutions in shorter execution times for hard optimization problems. Using these techniques we can easily fine tune our algorithm to find an optimal balance between *exploration* and *exploitation*, that is, between the creation of diversity and its destruction by means of selection, to achieve optimal results. The methods investigated are *master-slave*, which employs a single population but distributed across different processor units, *fine-grained* architectures, which still uses a unique population but spacial-distributed where interaction between individuals is restricted to the overlapping of some neighborhoods and *coarse-grained* architectures where we maintain several populations which eventually communicate with each other using *migration*. Finally, *hierarchical* architectures add one level of complexity by combining the three above methods in higher-dimension models which take the best of each paradigm. These methods can equally be applied to *multi-criteria* and *multi-constrain* problems where a global optimum does not exist, but a set of locally optimal solutions are needed instead.

1 Introduction

The simplest Evolutionary Computing (EC) strategies always consider *panmictic populations* where all individuals are potential partners and parents of the next generation. This implies that mating and recombination amongst any individual is possible, although not equally probable, without any physical, spacial, social or genetic restriction. However, in real life, these restrictions emerges usually in natural populations. For example, physical restrictions exist within species, where the physical aspect of on of the partners may influence the probability of mating. Spacial constraints appear for instance in terrestrial animals, such as many mammals, where relationships only exists between individuals which coexists within the same habitat, such as island. Social segregation exists still in many ways in many countries, where high-class individuals do not tend to mixture with lower-class people.

The main problem with *panmixia* is that in many occasions the random mating within the whole population might converge to a unique kind of genotype, possibly the best adapted within the population, which, after a few generations, ends up by dominating the entire population, choking the diversity of the community. This effect, known as *genetic drift*, can turn out to be harmful for the individuals, given that a sudden change in environment may completely devastate the highly uniform and specialized population which cannot quickly adapt to the new ecosystem.

When applying Evolutionary Computing (EC) to mathematical problems we realize that complex instances, for example, VLSI routing design, as real life ones, are not usually as simple to solve as finding a global maximum which remains static *ad eternum*. Many times indeed, the objective to be optimized is not well defined or it might be fuzzy. Other times, the environment changes so quickly that there is no competitive edge in achieving a great degree of adaptation to the ecosystem, but what it is required is a fast response to drastic perturbation of the habitat. For all those situations, population would rather have diverse enough individuals locally adapted to some particular characteristics of the milieu, instead of a unique, global and homogeneous adaptation, probably not even being optimal.

In this document, we address several techniques to avoid this problem of lack of diversity, which, at the end, leads to *premature convergence* and poor quality results. At the same time, these techniques can be naturally applied to face problems that by their very nature, accept multiple solution, not necessary optimal. Finally, and given the structure of the different EC methods, these techniques are easily parallelizable, which implies, not only better quality results, but also important speed-up with respect to the serial implementation.

2 Master-Slave Architecture

The first approach to tackle the problem of multi-objectivity and parallelism is the *Master-Slave* architecture [4]. In EC strategies where fitness evaluation or mutation operators are a bottle-neck for the performance, the simplest solution to speed-up the calculation is to split the population in N groups and dispatch the evaluation and possibly also some of the genetic operations of each subgroup to different *slave* processors.

At the beginning of each generation, a *Master* processor assigns to each *slave* a group of individuals. Each *slave* machine evaluate the fitness function on each of them and sends the results back to the *master*. It is also possible that unary genetic operators, such as *mutation* can take place within the *slave* processing. Once the *master* processor have received all individual evaluations, it performs *recombination* and *selection* of the survival for the next generation, according to the schema chosen for the algorithm. Parallel *recombination* and *selection* are also feasible, still treating the whole population as a unity, but implementation difficulties usually are not worth the results achieved.

Master-Slave architecture is thus simply a method which possibly allows some speed-up in the execution of the algorithm taking profit from parallel architectures widely available nowadays. Observe that, from the evolutionary point of view, no real change in *genetic operations* have been made, and the population stills acts as a *panmictic group* and hence, we should not expect improvements in the quality of the final solution with respect the sequential implementation.

When designing *Master-Slave* schema, we have to carefully consider the traffic between *master* and *slave*, since *master* processor can easily become the *bottle-neck* of the architecture if the number of *slaves* is above a certain limit, or the information to be sent to each subgroup is large, for example, because the individual representation takes a lot of bits. In that cases, communication overhead might be a serious issue which degrade the performance.

On the other hand, the architecture does not perform very well on heterogeneous architectures, where each *slave* processor has different computation capacities. If some *slaves* are much faster than others, keeping the workload balanced between nodes is a hard task, and the speed-up which could be potentially attained with the parallel implementation can be easily eroded on synchronous implementations, where faster *slaves* will waste a lot of idle time waiting for the slowest ones.

In summary, this architecture does not provide with different or better quality results, it is not a truly multi-objective strategy, but instead it is only used to speed-up computation of the well-known serial versions of the algorithm. The implementation can be advantageous as long as the processor used are all homogeneous, and the cost of communication between *master* and *slave* is small. However, if the communication traffic is elevated or processors are very different from each other, the cost of implementing an *asynchronous* version of the algorithm probably does not pay-off the result, and we would better implement one of the strategies explained below.

3 Diffusion Model

The *diffusion model*, a.k.a *cellular EA* or *finely grained model* is the first model which clearly states a division within [3]. The algorithm hold a **single spacial distributed** population which is structured into *demes*, which sometimes are of just one individual. Spacial structure is usually two dimensional toroidal grids, and mating between individuals is limited to a small neighborhood around the *deme*. These neighborhoods usually overlaps, so that the interaction amongst individual can be disseminated, step by step, to the entire population. Selection methods are usually the same we use on other sequential EC strategies. This step by step transmission between *demes* because of the overlapping is what creates the diffusion effect of the best genotypes, similar to the diffusion process by which heat is transmitted or

patterns on the skin of the animals are formed.

The overlapping neighborhoods provide thus an implicit method of *migration*, achieving similar effects to the explicit migration method we will study later on the *island model*.

3.1 Neighborhood size and shape

As we mentioned above, the size of each *deme* within the *diffusion model* strategy is small, usually just one individual. Thus, when designing such an strategy, the possible configurations of the algorithm are mainly determined by choosing the neighborhood limits, by fixing its size and shape. The effect of these choices are not well understood yet from the theoretical point of view, and usually they have to be tested empirically for each problem. Reference [7] gives a good insight of these effects.

Two common configuration of neighborhoods are the so-called LN and CN. The first one, LN, defines a linear neighborhood as the *demes* reachable from the central point, taking at maximum n steps in a fixed axial direction (North, South, West, East), such as the number of individuals within the neighborhood is N. The CN scheme allows for neighborhoods comprising *demes* in the closest $n-1$ points to the central point, in any direction, such as the number of individuals of the neighborhood is N. Both examples are represented in figure (1).

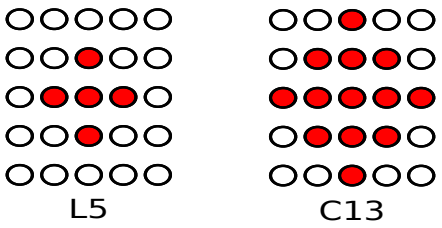

Figure 1: Neighborhood shape schema

In order to quantify the effect of spacial distribution structures on the selection pressure, which itself affects the diversity of the population, we can use the *Growth Curve Analysis*. This analysis measures the selection pressure which a given method induces on a population where only reproduction takes place, so that results are not influenced by several mutation or recombination methods. The idea is to plot the proportion of the best individual within the population through the generation, so that we can visualize the growth rate of it and how it takes over the whole population. The slope in the curve will represent the selection pressure induced by the method. These analysis have been largely studied in genetics, specially for usual methods such as linear ranking and fitness proportional, and in most cases the logistic curve emerges as the underlying growth model.

When applying this analysis to spacial-distributed methods, we observe that the method still shows an approximate logistic curve, but with lower growth rates, meaning that, overall, spacial-distributed population shows less selection pressure, increasing thus diversity. These results are a combination of the selection applied to each of the local neighborhood and the propagation time necessary to spread the best individual globally through the entire population. This propagation time will depend on the size and the shape of the neighborhoods, as we are explaining in the paragraph below.

The relationship of the size of the neighborhood with the selection pressure is rather intuitive, as the size of the neighborhood with respect to the population size increases, it creates a larger overlap area, and hence the propagation time decreases. Decreasing the propagation time induces more selection pressure, which, at the same time, leads to a higher growth rates for the best individual. However, if we explore different neighborhoods with the same size but with different shape, we can see that the shape plays an important role, since we can get very different results using different spacial configuration.

With this idea in mind, [7] analyzed different configurations and concluded that the major factor determining the growth rate for a same size neighborhood was the *equivalent radius*, which is defined as the spacial dispersion of a point pattern with respect to the center of the neighborhood, according to formula (1).

$$radius = \sqrt{\frac{\sum (x_i - \bar{x})^2 + \sum (y_i - \bar{y})^2}{n}} \qquad \bar{x} = \frac{\sum x_i}{n}, \bar{y} = \frac{\sum y_i}{n} \tag{1}$$

With this result, we can then adapt the selection pressure of our diffusion model by simple adjusting the *equivalent radius* of the neighborhood, by modifying the size and shape of the population, according to expression (1). Moreover, [7] goes further and try to approximate the logistic curve explaining the growth model from the relationship derived above, so that we can easily fine tune our *diffusion models* to either replicate the performance of the equivalent sequential algorithm (but with a lower execution time due to parallelism) or improve results by finding a growth rate which maintain a better balance between exploration, via genetic operators and exploitation, via selection pressure.

3.2 Implementation

Fine-Grain parallelism are specially suited for massively parallel computers, such as Single Instruction, Multiple Data (SIMD) processors like GPUs. Since the size of the *deme* is very small, usually being made of just one individual, the *genetic operators* as well as the *mating* and *survival* selection are easily implemented on these architectures. Observe that the gain of the speed is based on the fact that these architectures have a shared-memory easily accessed by all the cores within the processor, being the communication process between *demes* very efficient. Also, being the number of individuals within the *demes* very small, population usually fits within the cache memory of the processor, speeding-up further the execution of each generation. The difficulty might reside, given that we use shared-memory, in defining a sequence of read-write operations easily parallelizable which avoid data corruption. When the shape of the neighborhood is very different from a square, the definition of such a read-writing scheme may be far from trivial.

4 Island Model

Island model, also known as *Coarse-Grain Parallelism*, is based on the theory of *punctuated equilibrium*, which states that populations are usually partitioned into several different subgroups, which evolve

themselves isolated for some generations. Periodically, individuals from one subgroup communicate with other subgroups, by means of, for example, *migration*, adding diversity to the group receiving the new individual. Under this theory, evolutionary process can be seen, at subgroup level, as a process with slow progress where no substantial changes appear on the individuals, until, all of a sudden, a foreigner genotype from some other subgroup comes in with a new feature which is quickly adopted by the population and make the evolutionary process advance rapidly, until reaching again a static equilibrium. Real evolutionary process takes places thus on a few generations after the new individual comes into a subgroup adding some diversity.

This model, as explained in [6], is an *implicit method* which favors, but not forces, the creation of different *demes*, or local populations, sufficiently isolated, by physical distance or any other analogy, enabling the differentiation between groups of individual and thus increasing the diversity within the whole population, which is the aim of these methods.

4.1 Partitioning

For the *Island Model* paradigm, we usually divide the whole population $\wp$, with a number of individuals $|\wp| = M$, into N subgroups $\{\wp_1, \wp_2, \dots \wp_N\}$, each containing $\mu = \frac{M}{N}$. For the sake of simplicity, subpopulation are usually homogeneous in terms of size, although experiments with different size subgroups have also been studied.

One of the first decisions when planing to implement an *island model* is the partitioning parameter μ. As we saw in previous problem set, the behavior of the population is very dependent on the size. Thus, the choice of μ is very important for the evolution dynamics at subgroup level. It is commonly thought that there exits a threshold value of μ below which poor results are achieved, but that, once the μ is above the threshold value, the behavior of the overall algorithm, considering all subgroups, is less sensitive to the partitioning. Hence, the problem lies in finding this threshold value.

In order to better implement the sense of *distance* or *differentiation* amongst individuals within the population, we can have different *fitness evaluation functions* or even *selection* processes both for mating and survival for each subgroup. Observe though that having a different *fitness evaluation* function is not always possible, since some optimization problems require the function to be optimized to be the same at each subgroup. However, other loosely defined problems can make use of this option, for example, when trying to optimize functions which consist on linear weighting of several factors, in which case we can use different weighting schema to express different preferences for different persons.

4.2 Migration process

Once the partitioning in *subgroups* has been made, the next decision is the intercommunication or migration policy amongst the subpopulations. Several aspects have to be considered here, the first being the *topology* of the communication. Usually the problem is modeled as a graph problem, where each *deme* is the vertex of the graph and the edges indicate which subgroup can communicate with each other. Usually, edges are undirected, that is, communication is symmetric and if one group is linked to another, communication is possible in the two ways. However, although less frequent, schema with asymmetric communication are also possible. Common choices for the *topology* are meshes, rings or its generalization for $n > 2$ dimensions, such as hypercubes and torus.

Once the *topology* has been defined, we have to set the migration parameters, such as *migration frequency* and *migration magnitude*, that is, the number of generations of isolated evolution before a communication between subgroups takes places, a.k.a *epochs*, and the number of individuals that actually migrate from one group to another.

The decision above determine the degree of isolation or interaction amongst the subgroups. Observe that, the more connected is the graph and the more frequent is the migration, the more our *island model* looks like a single, large, *panmictic population*, obtaining the behavior of *premature convergence*, *lack of diversity* or *statis* that we want to avoid using these paradigms. At the other extreme, when the graph is poorly connected and the frequency of migration is low, the subgroups stop interacting with each other and the behavior is similar to what we would obtain by running a batch of N executions with a population of μ individual.

Thus, since at both extreme cases we will end up the the *premature convergence* problem, achieving results which usually get stuck at local maxima, we should find a configuration in the middle of these extreme points which will favor *diversity* and thus *exploration* of new and different solution at each *deme*.

Furthermore, we have to define exactly the terms of migration between two groups. First of all, we have to define the *selection* of the migrants at each *deme*. Selection methods reviewed in previous problems can be used, for example, to implement the migration policy. Random selection, random selection based on a fitness or ranking probability schema, best individuals are some of the different selection configurations which can be used. Once we have the *migrants*, we have to define the *assimilation* process on the host population. Here several alternatives exist. We can, for example, remove the selected *migrants* from the source population and paste them on the *host population*. Other alternative is just to make a copy, maintaining the individuals in both the *source* and the *host* population. In any case, we have to perform, after migration, a *survival selection* process which maintain constant and equal to μ the number of individuals at each subgroup.

[2] gives good insight about all parameters which can be set up for the *migration policy*. It specifically provides mathematical expressions to measure the intensity of the selection depending on the selection method we use on the *migration policy* to choose the *migrants* as well as the frequency of these migrations. One important result is that the selection method for the *migrants* affect more to the *selection intensity* of the algorithm than the *replacement method* used on the host population which receives the new individual. So, defining experiments such as *best migrants replacing worst individuals in the host population*, *best migrants replacing random individuals in the host population*, the two method obtained similar results. However, when studying the effect of *random migrants replacing worst individuals*, we remarked that increase in convergence speed is not important in this case.

Finally, until now, we have considered static *topologies* where the graph describing the possible migration flows does not change during the execution of the algorithm. Adding some complexity, as described in [3], we can equally define dynamic *topologies* where the migrants can flow from one group to another, without fixed restrictions, based for example on estimations of in which subgroup the new migrant would provide with more diversity and thus better results. To estimate the target subgroup, we could use distance functions in order to measure the differences between two subpopulations.

4.3 Implementation

Although the actual (possibly parallel) implementation of this paradigm can be done in several ways, *Coarse-Grain Parallelism* is best suited for distributed systems, where the population is divided into subgroups and assigned each of them to a machine within a cluster. Evolution takes places at subgroup level, being each of them assigned to a different machine, and, after an extended number of generations, subgroups communicate using the message-passing interfaces typical from these systems. Observe that, since time between *epoch* tends to be large, the CPU-demanding work is done at each machine, and communication process does not take up a lot of resources.

Of course, any other logical structure different from the distributed system architecture can be im-

plemented. For example, each population can be run within a *operating system process*, all of them being executed on the same machine, and communicating using Inter Process Communication (IPC) methods.

On the other hand, the definition of the *epochs* can condition the performance and architecture of the model, if we choose a fixed number of *generations* between *epochs*, model is homogeneous and can be easily paralelizable. However, if we set the limits of each *epochs* the moment when the subpopulation reaches *statis* or *equilibrium*, the workload at each machine could become unbalanced, since one group will converge sooner than other, and the parallel performance can be seriously affected. In this case, we should consider to design an *asynchronous* version of the algorithm, which is much more arduous to implement.

5 Hierarchical Models

The three previous methods of Distributed Genetic Algorithm (DGA) use the concept of *spacial separation* or *migration* to generate better quality solutions. *Hierarchical models*, as described in [5], combine the best features of each of them by creating a new level of hierarchy and grouping several DGA in an architecture which communicates with each other as on the *island model*. So, a Hierarchical Distributed Genetic Algorithm (HDGA) connects several DGA whose nodes are at the same time smaller DGA

In the hierarchical model we distinguish thus two types of migrations, the local ones, within a given DGA and the global ones, between different DGA. At the same time, the configuration of each of the DGA does not need to be homogeneous, but each DGA can have its own strategy, based either on *island* or *diffusion* models, and with different configuration parameters, as well as genetic operators. However, although any combination is possible, HDGA are usually built by connecting *coarse-grained* algorithms on the higher levels and either *fine-grained*, *coarse-grained* or *master-slave* in the lowest level.

The underlying idea of HDGA is the same as in DGA, by combining different separated populations, we can obtain better quality solutions within a shorter period of time. [5] and [3] mention many examples and test-suites where HDGA have been proven successful to improve the performance of simple DGA. However, because of the degree of complexity achieved in these architectures, and because they are a relatively new models, HDGA are nowadays reduced to the most complex problems where the extraordinary development effort is worth it.

The number of combinations in terms of topology, selection, genetic operators and synchronization across the different subpopulation is the same as we saw for the more simple versions of DGA.

6 Critics

The above methods tackle the problem of speeding up computation of *EC* strategies using mainly two concepts, that of parallelism by dividing the evaluation and genetic operation between several processors, and creating implicit methods which increase the diversity of the population avoiding the *genetic drift* effect leading to *premature convergence*, therefore decreasing the number of generation needed to achieve convergence to the solution.

On the other hand, additionally to the increase of speed of convergence, the breeding in parallel of several groups of individuals can be advantageous for *multi-objective* problems where, instead of a global optimal solution, a set of locally optimal individuals are preferred. In this case, the set of optimal individuals can be naturally defined as the best individual of each *deme* or *subpopulation* within the different paradigms.

Each paradigm has its pros and cons and depending the time we are willing to spend in the development of the strategies, the underlying hardware architecture we are using and the complexity and nature of the problem to solve, we should decide between one or another.

Master-Slave paradigm does not really provide better quality solution, but only a speed-up in terms of the time of execution. Observe that the population in this method is still the same, and the recombination and selection operators are applied in the same way as they would be on the equivalent sequential algorithm. The only advantage of the method is that, by using several machines to evaluate the individuals, the consumed time can be largely decreased, specially in problems with complex evaluation functions. The advantage of the method is that it is easy to implement. Once we have designed the sequential algorithm it is almost trivial to parallelize it just by adding a communication interface between the master and the slaves. The remainder piece of code does not need any modification. The negative point of the method is that it does not really improves the quality of the final solution, and that, if the communication overhead between slave and master is big, the equivalent serial performance of the parallel algorithm can be seriously affected. This communication overhead can be due to a great number of slave processor or to the big size of the individual in terms of the bits required in the memory representation.

Fine-grained paradigm has been widely used and developed with the increasing availability of low-cost highly parallel architectures such as *GPUs*. Since the algorithm still uses a single population, the configuration of the parameters is reduced to the size and shape of the neighborhood, which will determine the speed of diffusion of the best solution. The increase in speed-up is notorious, not only because the number of generations to achieve converge is lower because of the bigger diversity, but because the hardware architectures, based on shared-memory typically within the same chip, and the small *demes* fits very well on the CPU-cores of the GPUs. The negative point is that many of this highly-parallel on-chip architectures have a limited amount of memory, and performance is eroded when we use this method with very large populations, of, for example, million of big-sized individuals, which might not fit within the physical memory. Additionally, highly parallel architecture still requires programming skills very different from the usual sequential programming.

Coarse-Grained architecture employs several populations, usually ran in different processors, which induce an increase of diversity. The paradigm is inspired on the *punctuated equilibrium theory*, and the increase on exploitation arises both from the isolated evolution within each population and from the infusion of new individuals by means of *migration*. Thus, the increase in execution speed is done, not only by the parallelization evaluation of the different executing, but also because of a decrease of the generation needed to convergence. Once we have developed a *master-slave* architecture, in general, as long as we can work with *synchronous* algorithms, it is easy to translate the code to a *coarse-grained* algorithm, just by adding the *migration* logic.

The negative point of these strategies is the quantity of parameters we have to fine tune in order to obtain the best of it. In concrete, we have to carefully choose the *topology* and *interconnection* amongst the population, as well as the number of subpopulations and its size. As for *migration*, we have to look for the optimal *migration rate* and *size*, in terms of number of individuals exchanged. Typically, there exist a *threshold* value for the *migration rate* under which the population are very isolated, and the migration does not provide with any benefit, and above it the *migration rate* is so high that the algorithm acts as a single population, decreasing the diversity. Although general guidelines are known for these parameters, optimal configuration can vary a lot from problem to problem, and thus, we have to spend many time looking for how to best fine tune our algorithm.

Finally, the time span of the *epochs* or time between *migrations* can be set fixed in terms of a number of generation or can be dynamically adjusted depending on when each subpopulation reaches *equilibrium*. In the first case, fixing the number of generation between epochs, we take the risk that the subpopulation

achieves convergence very soon, thus, wasting computational resources that could be saved triggering the migration before in time, or it could be the other way around, the case where a population has not still converged and the quality of the individuals chosen as migrants are not good enough. In the second case, waiting for the subpopulation to achieve *equilibrium*, we take the risk that each subpopulation takes a different amount the time to complete, and thus, an *asynchronous* implementation of the algorithm must be developed, which is a much more arduous task.

As for the *Hierarchical* architecture, the above pros and cons prevail, since the paradigm is just a combination of different models under a specific topology and communication scheme. In general, the implementation of this architecture is by far the most complex, given the extra level of complexity we add by increasing the number of layers. Also, the number of parameters to configure increases with the number of levels and hierarchies, which can demand big time just to find the optimal configuration. Although they have been proven to beat the previous more simple architectures, their relatively poor adoption by practitioners as well as well as their complexity of implementation, make them an option only for the most demanding problems.

As we have seen, the two main advantages of the above methods are the production of better quality solutions for complex optimization methods, and the decrease in computational time required to achieve that results. This decrease in computational time stem from both, the generation of better individuals which make the algorithm to converge faster, and the use of parallel architectures to solve the problem. Some authors [1] suspects that many of these configurations can even achieve *super-linear* speed-ups in performance, that is, that the execution of these algorithms on n processor runs more than n times faster than the sequential version of the algorithm. However, the merit of this potential speed-up might not be due only to the configuration of the parallel architecture, but also may have to do with the implementation and the hardware configuration. Specifically, on genetic algorithms we do extensive use of search and sorting, many of them are $O(n^2)$, with n the number of individuals. Dividing the population in M groups of μ individual, with $\mu \ll n$, the algorithm can greatly increase the performance when being executed on a smaller population. On the other hand, the execution of smaller population in different machines can take advantage of hardware configurations, which increase the execution time of the genetic operators. For instance, the large population on the sequential algorithm might not fit in the CPU cache, and for a single generation, many cache faults might occur, which delays the execution since the individuals must be fetched from the main memory. When executing the parallelized version of the algorithm, the smaller population might fit on the cache, thus, not provoking cache faults and speeding the execution by an important factor.

7 Conclusion

The family of algorithms above presented are able to generate better quality results in shorter periods of time, many times showing super-linear speedups with respect to the sequential algorithm. These improvement is mainly due to the use of techniques aimed, implicitly or explicitly, to increase the diversity within the population, avoiding *premature convergence*, and achieving better individuals in fewer generations.

The underlying idea of using several differentiated or partially isolated populations not only improve results, but they can also be applied to multi-objective and multi-modal problems where more than one solution is desired, instead of getting a single local optimum.

Usually, the implementation of these techniques require more work than in the sequential version of the algorithm, and the best resulting architectures are also the most challenging to develop. As a result, we have to carefully choose the architecture of our program according to the complexity of the problem to be solved, since, for not so complex problem, it might be beneficial either execute a fine-tuned sequential version of the algorithm (with bigger population or more intelligent selection methods) or to implement

one of the simplest algorithm presented on this paper.

As in other parallel architectures, the use of *asynchronous* or *synchronous* communication is also a point to keep in mind, being much more complex to implement the first solution, although, the benefits are a better use of CPU-resources.

8 Acronyms

DGA Distributed Genetic Algorithm 8

EC Evolutionary Computing 2

HDGA Hierarchical Distributed Genetic Algorithm 8

IPC Inter Process Communication 8

SIMD Single Instruction, Multiple Data 5

References

[1] Enrique Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82:7–13, 2002.

[2] Erick Cantu-Paz. Migration policies, selection pressure, and parallel evolutionary algorithms.

[3] Erick Cantú-Paz. A survey of parallel genetic algorithms. *CALCULATEURS PARALLELES, RESEAUX ET SYSTEMS REPARTIS*, 10, 1998.

[4] Adrian Grajdeanu. Parallel models for evolutionary algorithms, 2003.

[5] F. Herrera, F. Herrera, M. Lozano, M. Lozano, C. Moraga, and C. Moraga. Hierarchical distributed genetic algorithms. Technical report, International Journal of Intelligent Systems, Vol, 1997.

[6] W N Martin J Lienig and J P Cohoon. Parallel genetic algorithms based on punctuated equilibria, July 02 1997.

[7] Jayshree Sarma and Kenneth De Jong. An analysis of the effects of neighborhood size and shape on local selection algorithms. pages 236–244. Springer-Verlag, 1996.