Enrique Alba and José M. Troya

Departamento de Lenguajes y CC.CC., Universidad de Málaga
Campus de Teatinos, 29071, Málaga, Spain
eat@lcc.uma.es

Gaining New Fields of Application for OOP: the Parallel Evolutionary Algorithm Case

ABSTRACT

Object Oriented Programming (OOP) is continuously gaining new domains of application. We address in this work a study of several important design and implementation issues in one of such new domains: parallel evolutionary algorithms (PEAs). These algorithms are heuristics aimed at performing search, optimization, and machine learning tasks. We will identify the potential and actual advantages of using OOP in such a field of application, as well as we propose a class design for solving complex real-world problems with PEAs. Besides the methodological and practical outcomes, some results showing the efficiency and flexibility of the resulting OOP-PEA systems are offered herein. We conclude that OOP allows quick PEA prototyping, integration of new techniques within the PEA, and easy cooperation with other techniques in parallel, all of this without reducing the efficiency of the resulting PEA.

Introduction

Evolutionary algorithms (EAs) are a large family of stochastic heuristics that have been successfully applied for solving many search, optimization, and machine learning problems [3]. Unlike most other optimization techniques, EAs maintain a population of encoded tentative solutions that are *competitively* manipulated by applying some *variation operators* to find a global optimum.

A sequential EA proceeds in an iterative manner by generating new populations of *individuals* from the old ones (see Figure 1). Every individual is the encoded version (symbol string or tree) of a tentative solution. An evaluation function associates a *fitness* value to every individual indicating its suitability to the problem. The standard EA applies *stochastic operators* such as selection, crossover, and/or mutation on an initially random population in order to compute a generation of new individuals. The mean fitness is improved from generation to generation in this manner.

For non-trivial problems this process might require high computational resources, and thus, a variety of algorithmic issues are being studied to design efficient EAs.

We stress on one of such issues consisting in using parallel models of EAs (PEAs).

```
proc EA
        t:=0;
        generate and evaluate the initial P(t);
        while not termination_condition do
                t:=t+1;
                select individuals for P(t);
                reproduce individuals in P(t);
                evaluate P(t);
        end_while;
        return best_individual_ever_found;
end_proc EA;
```

Figure 1. Pseudocode of an EA.

PEAs have been applied in real-world optimization problems with a considerable success [1] [11]. In fact, these applications have almost completely attracted the interest of the researchers. This means that numerical performance, new operators, new ways of encoding the problem parameters, etc. represent the body of knowledge in this field. Furthermore, this also means that methodological issues and software engineering concepts are rarely stressed in important works of the so called *evolutionary computation* field.

There are, however, many reasons to take care of the software quality, even recognizing that this is not a primary goal for the EA community. In particular, parallel EAs are having a large importance in dealing with complex problems to overcome their requirements of high memory and CPU time [6] [11]. Combining operators, quick prototyping, organizing the experimentation, and related tasks demand the flexibility and abstraction that only OOP can provide.

In this paper we show the advantages of using an object oriented design and implementation. After a background section on PEAs for non-specialists, we will offer some general guidelines to design a hierarchy of useful classes for constructing a PEA. Then, we present existing works in this area, and propose one of our OO models that follows these guidelines. We will show that the resulting OO system is highly efficient on difficult and real-life problems, as well as very flexible and easy to use for novel practitioners. In addition, some hints on the OO implementation of PEAs are offered. Finally, some conclusions are summarized and future implications of this work are pointed out.

Background on Parallel EAs

EAs are a large family of parameterizable search algorithms whose behavior must be specified before solving a problem. They are meta-heuristics that need to be instantiated with concrete techniques to be used in practice. In order to *customize* a PEA to solve our problem we need to undertake the following steps:

1. Defining a mapping for the parameters of the problem to be encoded in the form of a string (or a tree) of symbols. This is called the *genotype*.
2. Designing the evaluation function that assigns a fitness value to each string: its fitness value.
3. Determining the set of operators to be applied.
4. Assigning values to the parameters of the algorithm.

These steps lead to an algorithm that searches for a good sub-optimal solution (hopefully an optimum). Mapping the problem parameters to a string usually yields a binary (or floating point) string. The evaluation function represents the problem; thus, the researcher can add as much problem-knowledge as needed to it. As to point 3, the operators in an EA work by *selecting* the best strings attending to their fitness, *crossing* random slices of two of these selected strings to yield new pairs of strings, randomly changing their contents (*mutation*), and replacing the old pool with the newly computed one (*replacement*).

See Figure 2 for an example iteration in a generational genetic algorithm (GA) that optimizes $f(x)=x^2$ with $x \in [0,255]$. A GA is an EA applying all the above mentioned operations. In this example, every population contains four strings, each one encoding a tentative solution in 8 bits (one single *gene*). Before evaluation, each string needs to be expressed in its *phenotypic* form to be understandable for the problem (binary-to-decimal in this easy problem).

Selection copies the fittest strings to a temporal population (two copies of the second string, and one copy of the first and fourth strings). A single point crossover is performed on half of the population (p_c=0.5). This provokes the exchange of the two last bits between the first and fourth selected strings, while the second and third strings remain unchanged. Mutation is applied at a low rate, and this is why it only changes (flips) the first bit of the third string. Notice the increment in the average fitness in the two successive generations. Also, a solution is found in $P(i+1)$ at x=255, $f(x)$=65,025.

From this example it can be inferred the need of using long strings and large populations for solving complex problems. This is the main reason for using parallel EAs instead of a large sequential EA. In addition, using separate populations have shown to considerably reduce the numerical (not only the physical) effort to locate a solution [1] [6]. The algorithm terminates when a solution is found (if the solution is known), or after a pre-defined number of iterations. With any fixed effort, the algorithm can locate more than one solution to the problem, which is a very interesting feature of this class of meta-heuristics.

str#	*genotype*	*phen.*	*fitness*	*pr. sel.*
1	**11001001**	201	40,401	0.28
2	**11111110**	254	64,516	0.45
3	**00000001**	1	1	7e10^{-6}
4	**11000011**	195	38,025	0.26

***P*(*i*)**

avg. fitness=35,735.75

Select	*after selection*	*Cross*	*after crossover*	*Mutat.*	*after mutation*
	111111-10		**111111-11**		**11111111**
	11111110		**11111110**		**11111110**
	11001001	p_c=0.5	**(1)1001001**	p_m=1/32	**(0)1001001**
	110000-11		**110000-10**		**11000010**

str#	*genotype*	*phen.*	*fitness*	*pr. sel.*
1	**11111111**	255	65,025	0.37
2	**11111110**	254	64,516	0.37
3	**01001001**	73	5,329	0.03
4	**11000010**	194	37,636	0.22

***P*(*i*+1)**

avg. fitness=43,126.50

Figure 2. A generation in a sequential GA optimizing $f(x)=x^2$.

Utilizing parallel EAs usually requires the single (*panmictic*) population to be partitioned either in several sub-populations where island EAs are run with sparse string exchanges (*migration or distributed EAs*), or in the form of neighborhoods (*cellular EAs*). Let us see these three EA types in Figure 3. In distributed EAs, additional parameters controlling when migration occurs and how migrants are selected/incorporated in the target islands are needed. In cellular EAs the overlapped small neighborhoods help in exploring the search space. These two kinds of EAs provide a better sampling of the search space (exploration) and very often they improve the numerical and runtime behavior of the algorithm [8] [13] [14].

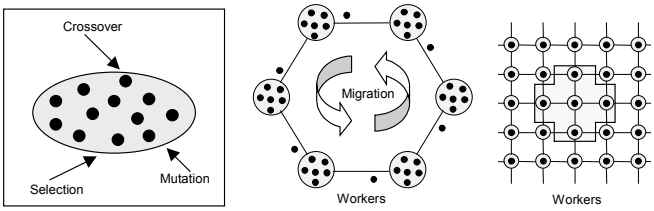


Figure 3. A panmictic EA has all its strings -black points- in the same population (a). Structuring the population usually leads to distinguish between distributed (b) and cellular (c) EAs.

OOP and Parallel EAs

After this brief description of the kind of algorithms we are dealing with, the open question is how OOP can help. To answer this question we must revisit the customization steps.

First, applying a PEA goes through an initial phase of prototyping, in which several (different) algorithms with different operators, encoding, probabilities, and alternative evaluation functions are sought: considering *more than one alternative* is very usual. An OO design can help in allowing a quick redefinition of the classes involved, while not merging different proposals. This helps to *identify* the best algorithm, and it looks more appropriate than, e.g., using an unrelated set of functions in C.

Second, because of their stochastic behavior, the results must be averaged, since one single execution is not statistically relevant. In this sense, some kind of high level approach to the *experimentation* is very useful. Explicitly including initialization facilities in the constructors of the OO classes (e.g., for reading configuration files) is very easy. Also, future extensions will have a clear point in the class hierarchy for improving the initial problem definition.

Among the general benefits of using the OO methodology for a PEA implementation we can outline the following ones:

- *Reusing* the software is, at present, difficult: a PEA software tool is either too specific (and thus useless for other problems) or else it is too general (and thus inefficient). OOP can help in reusing software, in the fast building of *prototypes*, and also in the quality of this software.
- A *common architecture* of classes for an evolutionary system could be created and enriched by different people with experience in the field. *Comparisons* should then be readily simple and meaningful.

An object oriented implementation of a PEA can also help in combining algorithms with different parallel granularities (computation/communication ratio), topologies, operators, or cooperation with other techniques. It offers structured facilities to develop new models of evolution.

In general, adding/changing/removing new operators, problems, encoding, etc. can be clearly undertaken when an OO design is used, since each of these tasks can be performed by a different object class. In imperative implementations, designing separate functions for the operators is not safe when combining operations, it is error prone, and the design does not abstract the algorithm itself. Some systems using a C++ implementation exist, but they usually fail in including this kind of operator pool facility.

Some examples of OO programmed PEAs are [1] GALOPPS, GAlib, EVOLVER, TOLKIEN, and GAME. From a software design point of view, GAME maybe the best reference. The rest of systems provide interesting ready-to-run algorithms with varying degrees of flexibility, but with no special contributions to the OOP domain.

General OOP Requirements

The traditional imperative aspect of an EA can drastically change by means of an OO design and implementation. In this section we are going to undertake a discussion on the form and composition of a class hierarchy suited for a PEA.

With practically the same amount of effort used to specify an imperative *pseudocode*, we can develop a c*ustomized-working code* for a problem in terms of object orientation. The last solution is much more flexible and rich in the data structures it uses, and also in the operations it can perform (also in future extensions to other problems).

In order to have object classes suited for future addition of operators we create two separate classes: a class for the `individuals` and a class for the `operator_pool`. We define the operator pool as a set of operators whose composition can dynamically change. The idea of an *ordered* list of operators, each one having its own set of parameters (not just an application probability), can be very helpful in controlling the search of the PEA, and, in fact, it is an ongoing necessity for solving modern search problems.

The work with individuals must be separated from their actual contents. Therefore, `genotype` and `individual` should exist as separate classes in order to allow for an independent manipulation. Also, a `problem` will serve to separate the search engine from the actual problem we are solving with it. Some `population` and s`tatistics` classes must be defined as well, since many operations deal with the population object, and since statistics must be gathered, both for the user and for the algorithm behavior.

Finally, we must include a `communication` class in this general hierarchy for transferring data among processors in the case of parallel EAs. Usually, an asynchronous exchange of information is faster than a synchronous communication in PEAs [2] [9]. Having a `communication` class allow us to easily hide their implementation differences and the physical details.

Also, some classes must perform the parallel interactions among EAs. A central monitor is normally needed to deal with the user: get global and internal statistics and status, kill a process, save and load populations on line, etc. All this work can be encapsulated into a `monitor` class, in order to declare instances that control a cluster of EA sub-algorithms.

A *socket* interface could be initially considered as a fast and widely available tool for implementing the communication. Other candidates could be PVM or MPI. But, on the other hand, the Java language incorporates such a fast message passing interface with the advantage of being platform independent. This makes Java a primary choice for future implementations of PEAs. However, Java must be programmed carefully in numerical applications, in special to avoid unnecessary object clones (slow execution).

C++ allows to fulfill all these requirements. But, since we are dealing with parallel algorithms, the necessity of a separate communication API is a drawback of C++, especially if heterogeneous systems are considered.

Parallelization is easy and flexible with OOP. The numerical behavior can be hidden from the communication system. However, some kinds of algorithms such as cellular EAs [6] [8] [14] cannot be easily derived from a hierarchy which has been targeted for coarse grain computations. Since cellular EAs (cEAs) locate one string in every node of a 2D grid, and since the interactions among strings in the near neighborhood are intense, an approach using operating system *processes* to model every string is not adequate.

Cellular EAs should instead be programmed by adding some spatial disposition to the population in each parallel

node. This means to include a `neighborhood` class in the hierarchy. As we will see, this could be a good approach, since it allows running parallel loosely coupled grids.

If a single cellular EA is going to be programmed and the full grid is to be partitioned among processors, then the implementation would significantly change from the case of distributed algorithms (discussed so far) [8]. In fact, *threads* should be a more adequate approach for cEAs. The parallel EA would be then composed of interacting threads, possibly running in different processors (many of them in the same processor), each one behaving like an individual in the cEA.

Besides these new ideas, cooperation or competition among different kind of EAs or even different non-evolutionary algorithms can be achieved with OOP. Such an OO distributed EA could deal with the requirements of the new kinds of modern optimization problems.

Related Works: GAME

GAME is one of the first systems including OOP ideas [11]. It has a very simple design for dealing with a potentially wide range of EAs (Figure 4). It incorporates some basic classes for manipulating genes (`gene`, `Bgene`) and other –generic– strings of symbols (`element`, `gstring`, `POOL`). The mapping from a *binary* genotype to the phenotype has been directly provided since it is widely used (`BinaryEncodeDecode`), although there exist some other mechanisms for more sophisticated mappings (`ENCODE_DECODE_CLASS`). The population is handled as a string pool by using a pool descriptor (`POOL_DESCRIPTOR`). New operators can be applied on pools of this type, but there is no special way of structuring these operators.

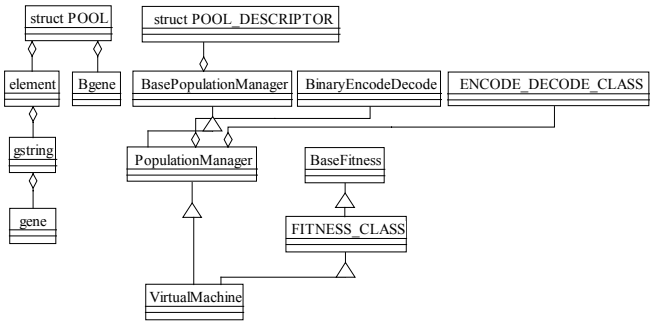


Figure 4. Class design of GAME in OMT.

The predefined manipulations on a population are performed on instances of the classes `PopulationManager` & `BasePopulationManager`. In addition, two classes for manipulating the fitness values are defined in GAME (`FITNESS_CLASS`, `BaseFitness`) in order the users to customize the fitness function for easing the selection of fittest strings. The virtual machine class (`VirtualMachine`) is intended to allow the definition of the reproductive cycle of a general EA: initial generation, applying operators, EA termination, etc.

The GAME class hierarchy is very generic, but it is good only in helping with the basic data structures of an EA (genotype and population). However, the claimed parallel extensions, graphical interface, and other advances are not being used in practice. The reason is that this generality is a drawback in GAME. In addition to some internal unreported errors (e.g., losses of memory), implementing a particular EA requires a considerable effort using GAME. Also, the base classes for individuals slow down the implementation of actual algorithms (memory is continuously de/allocated). Besides, the experimentation and addition of operators are not defined in its class hierarchy.

Therefore, in despite the important contributions of GAME to the OO design of EAs, its utilization has been very limited. The design of such general EAs environments forces the designer to provide a practical tradeoff between generality, efficiency, and easy utilization. This is the goal we will address in the next section by proposing *xx*GA.

Our Proposal: *xx*GA

By following the outlined OO guidelines we have developed *xx*GA. This software package is intended to help in developing quick algorithmic prototypes to solve complex problems by means of parallel distributed GAs, a subclass of the EA family. We want to stress that we are not interested in one actual *implementation*. In fact, we have designed and successfully used many other different systems under the presented philosophy. On the contrary, we are interested in the problems arising when using OOP in EAs and PEAs.

Figure 5 shows a description of the functional blocks of our parallel EA. It can be seen that a set of parameters and operators are given to solve the problem (upper half of the figure). The system uses a composition of parallel algorithms to provide a separate (*politypic*) evolution of populations. All this system is implemented by using OOP.

Figure 5. Functional blocks and OOP in *xx*GA.

One of our contributions is to consider that operators are not unrelated each other in the algorithm. Operators are grouped in a single class, thus including explicitly in the design the algorithmic concept of *reproductive plan*: a set of operations that transform the present population of individuals into a new one with higher average fitness.

All the operators have in common the necessity of parameters for its control (e.g., probabilities), an order of application, and so on. See Figure 6 and notice that a single error control can be added to this class for improving the quality of the resulting software. Besides that, our proposal allows an easy change in the behavior of the actual algorithm during the search: dynamic changes in the parameters, order of application, or composition of the reproductive plan. This is usual when applying EAs to complex problems.

We propose two classes: the class `Hybrid_Operator` for general purpose operators, and the class `Operator_Pool` for including evolutionary operators. See in Figure 7 the OMT model of *xx*GA.

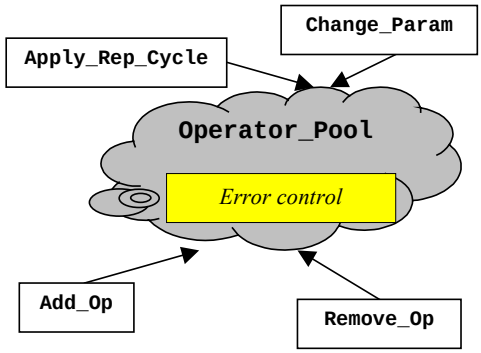


Figure 6. Reproductive cycle as an object class (`Operator_Pool`).

Notice in the class hierarchy of Figure 7 that the functional blocks are included as related object classes in the design. For example, the representation used in the individuals is taken from the binary (`Bit`) or real (`Real`) gene –parameter– classes. In order to use another kind of genotype (e.g., using integer permutations for combinatorial optimization) we only need to write a new class redefining some minor details on their generation and manipulation.

An `Individual` is composed by one *chromosome* plus a fitness value. A chromosome is an array of `Gene` values that can be added or not as a separate class.

Solving a new problem only requires to extend the behavior of the base class `Problem` in order to implement the objective function (see `F1`, `F2` and `F3` as examples of defining new problems for mathematical function optimization). An `evaluate` method must be defined for any new problem. Additional methods for problem configuration or monitoring can be easily included in such derived classes. We have done so for difficult learning problems like training a neural network with evolutionary techniques (classes `FFN`, `Pattern`, and `BPN`).

By defining the problem, the set of operators, and the representation used in the strings we can customize a `Population` ready to be used by the `GA` class. The population utilizes `Random` objects to generate random values and to make stochastic decisions. Since we are considering also structured populations (cellular EAs), a `Neighborhood` class with the most common topologies (mesh, ring, tree, etc.) is included.

Besides, an `Island` class is needed to include all the communication services, such as sending/receiving strings from/to the other parallel islands. Operations can be performed both sync and asynchronously, and some mechanisms to interact with the global monitor are provided. Since each island performs a basic sub-algorithm, we inherit from the GA basic class. One single instance of class `Monitor` is needed per cluster of GAs, which is intended to perform global tasks, such as spawning the islands, gathering on-line statistics, and terminating the search.

All the communication details are hidden inside the `Comm` class. The system uses its own communication protocol between the island and the monitor, and among the islands, in order to perform a distributed search. Tables gathering information about active sub-algorithms, search status, and global statistics are maintained during the search.

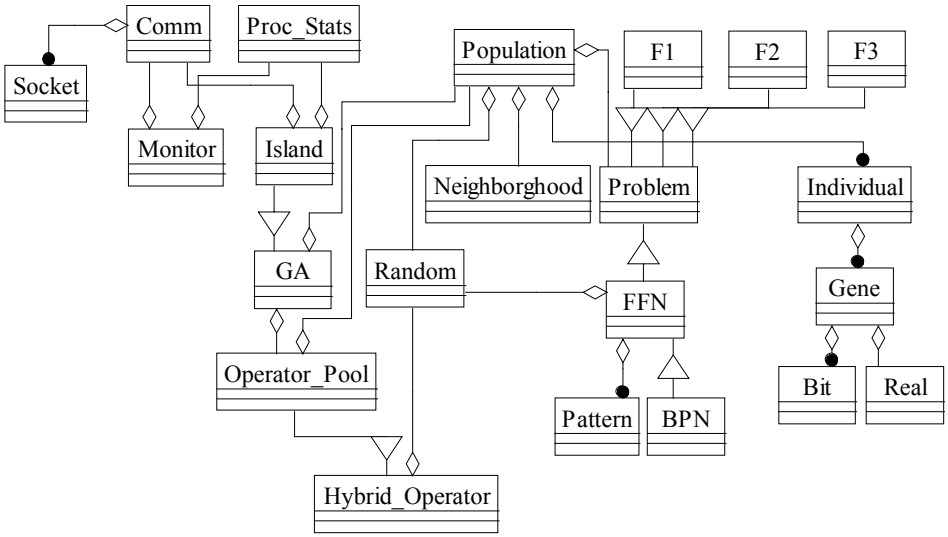


Figure 7. Class hierarchy of *xx*GA in OMT.

The present implementation uses a simple OO extension of the BSD `Socket` interface for the communication tasks. This grants access to many interesting low and high level communication options. Changing to other communication libraries such as MPI or PVM is reduced to changing the implementation of the methods in the `Comm` class.

Since in this research field the interpretation of results is very important, gathering statistics on the numerical search (average fitness, present best result, etc.) is used in any EA. Also, monitoring computational resources such as available memory and search time is very interesting. All these functions are embedded in our class `Proc_Stats`.

The whole system can be configured with a new parameter set to solve a given problem by using configuration files that are interpreted in the constructors of the associated classes. Reporting configuration errors is included as a part of each class in this hierarchy. This means that recompilation is seldom needed for a problem.

OOP Keeps Efficiency High

In addition to the expected theoretical advantages of an OO implementation, we are very interested in showing that the resulting algorithms keep the same levels of performance of existing implementations. In general, the kind of implementation does not influence the numerical behavior of the algorithm; only the execution time could be negatively modified by an OO implementation. C++ is efficient, while Java demands a careful implementation to avoid memory wasting. Also, the interpreted nature of Java is a small drawback, although new Java machines are very optimized.

Therefore, we are going to see how the efficiency is very high in sequential and parallel EAs which are implemented by using an OO methodology. To achieve this goal, first we are going to present results with the C++ implementation of *xx*GA on various problems.

In Figure 8 we show the average speedup with *xx*GA for solving the subset sum problem (NP-complete) with 128 elements [7], and for training a neural network (NN) that checks the parity of a four bit input [2].

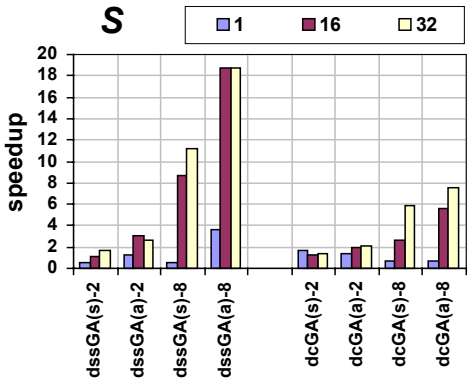
All the algorithms have been executed in a cluster of UltraSPARC 1 workstations with an ATM LAN. Any result is averaged over 100 runs of the considered algorithm.

For the subset sum problem we measure the speedup of a sequential GA versus a distributed GA with 2 islands in 2 processors, and versus 8 islands on 8 processors. Results are very good, specially when using a steady-state (dssGA) [12] reproductive plan (left part of the graph of Figure 8a). We label with **s**/**a** the distributed algorithms making sync/async migrations of strings.

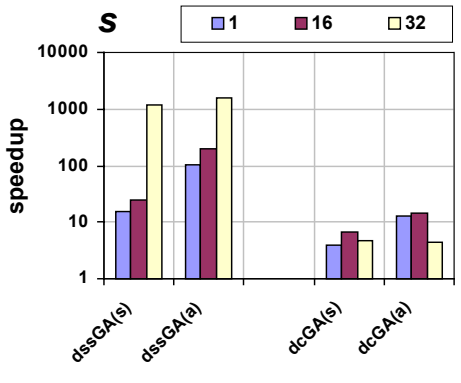
We report results herein with three degrees of coupling: high (1), medium (16) and low (32). These numbers represent the number of isolated generations performed between migrations of one random string to a neighbor island. See how a high coupling (sequential-like) affects the search process usually enlarging its cost. Notice also that linear and even super-linear speedup is possible since this kind of stochastic algorithms has a non-zero probability of finding a solution for any time $t>0$ [10].

The right part of the graph in Figure 8a reports results when the islands have a cellular GA inside each one (dcGA). The speedup is almost linear. Changing the reproductive loop from steady-state (dssGA) to cellular (dcGA) has been quite easy with our class design. Also, the experimentation tasks have been simplified thanks to the high level of abstraction: performing 100 executions on each configuration, adding the problems, defining the kind of operators (selection, two point crossover, and mutation), etc.

Finally, Figure 8b shows the speedup when training the parity4 NN with 8 processors. In this case we report the difference between a sync/async execution. Details on synchronization are hidden to the algorithms, and thus changing from one to the other is easy and safe. We can notice the advantages of using the mentioned asynchronous execution as well as the high speedup when a medium-low coupling (16 and 32) is being used. Distributed GAs are much faster than sequential GAs with one single population for both problems. Actually, comparing versus a single-population for a speedup value is not an orthodox measure in parallel EAs, but it is still useful to give a general idea.



(a)



(b)

Figure 8. Speedup when solving the subset sum NP-complete problem with 2 and 8 processors (a), and speedup when training a NN with 8 processors (b), both with sync –s– and async –a– distributed GAs.

Some Hints for Designing PEAs with OOP

In general, having a class hierarchy similar to that in Figure 7 is a good starting point for devising a PEA. But, in addition, some hints can be offered to implementors in order to avoid common pitfalls.

Our first advice refers to the `Random` class. This class is used for generating random individuals and for making stochastic decisions in the algorithm. Since the random number generator is common for all the application, the constructor of `Random` must not initialize the seed explicitly. If so, many `Individual` object instances with the same contents will be created, all of them resetting the seed to the same value. This occurs even if a clock dependent value is used for the seed, since many individuals could be generated very quickly. The hint consists in making a `static` initialization to ensure a single seed.

Our second hint deals with the implementation of the population, since this is a very important decision. In Figure 9 we can see that using a dynamic list for the population of individuals is more efficient than using a static vector. This can sound common sense if we think about the large amount of insertions and deletions that a typical EA performs, but it is good to confirm it in practice, since the system is more complex than merely handling a population appropriately.

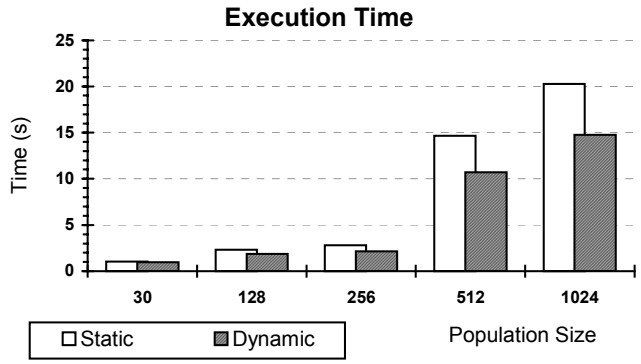


Figure 9. Static versus dynamic implementation of the population in C++ (Knapsack problem with 20 objects).

For example, when selection uses a sorted population (e.g., when applying selection by ranking [4]) it would be better to use some kind of vector operations in order to ease the frequently undertaken sorting steps. See such an scenario implemented in Java in Figure 10 with three different reproductive plans (steady-state, generational, and cellular) for solving a difficult deceptive problem proposed to mislead the search of an EA (MMDP8) [5]. Although it is straightforward to implement it with vectors, it is not recommended to do so, since it results in unnecessarily large execution times. Even if the selection operator does not use a sorted population, it still shows long runtimes. The main source of slowness can be found in the necessity of working with duplicates of individuals (and not with their references) in many points of the algorithm. Optimization on the memory used in the representation classes (`Gene`, `Chromosome`, `Individual`) is also very important when using Java. This is related to our third comment.

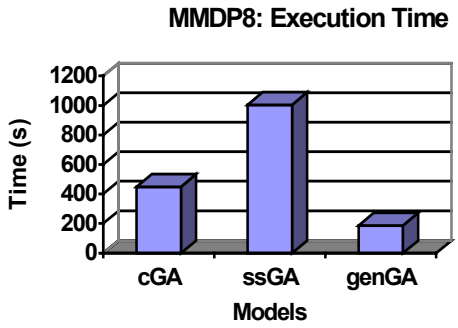


Figure 10. Time for solving MMDP8 in Java.

Our third hint deals with savings in memory. Reducing the memory spent in the basic classes for storing the parameter values (`Gene`, `Bit`, `Real`, `Chromosome`, `Individual`), is a very important issue. Devoting pointers to a container class at this level (Figure 11b) means to spend as much memory space in the pointer as for the value itself. In C++, some non-elegant solutions are usually undertaken. The reason is the necessity of efficiency in time and memory, since millions of individuals and temporal variables will be created and destroyed during the search.

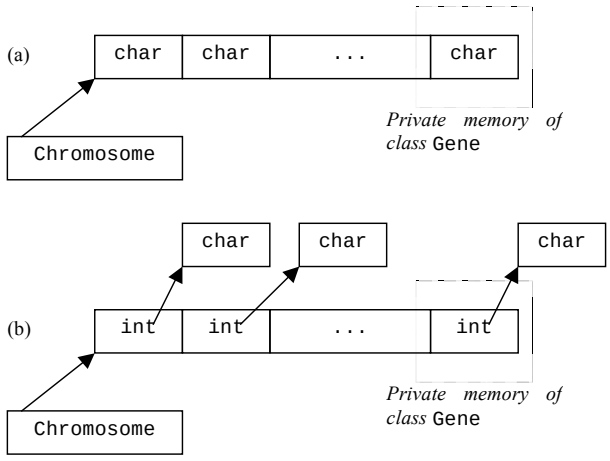


Figure 11. Memory savings in the lower container classes are very important in an algorithm that continuously generates and deletes millions of instances of such classes. Example with C++.

Our fourth and final hint is useful when utilizing sockets for implementing the communication classes. When doing a `Socket` class to abstract the raw socket library for the PEA, it sounds reasonable to *open* the socket in the constructor and to *close* it in the destructor. However, there exists a single global socket system for an application, whether OO, imperative, functional or whatever its implementation is.

Using *temporal* `Socket` variables in the methods of this class could lead e.g. to open a socket in a *receiving* method and to close it when exiting this method, which is an error, since the socket should have remained open for future calls to read and write operations on it. The hint consists in not closing the socket in the destructor. An additional method for closing the socket should be provided in order for the user to explicitly close the socket when it becomes useless.

Conclusion

In this paper we have discussed the necessity of a structured and flexible design in sequential and parallel evolutionary algorithms. Software quality is often missed in this domain since researchers are worried about solving the target application. Also, many non-computer scientists are unaware of the advantages that software engineering can bring to their future applications and experimentation phases, making them easier and methodological.

We have proposed a class hierarchy for parallel EAs and also we have shown that OOP is far from having a negative impact, neither in the numerical, nor in the real time needed for solving the problem.

Finally, we have offered some hints to help interested researchers in profiting from the numerous advantages of using OOP in their application and theoretical studies.

Many considerations are nowadays becoming increasingly interesting in relation to OOP and parallelism, especially when using Java as the implementation language. Java multi-platform parallelism and heterogeneity at virtually "zero cost" is a very appealing feature for future research with parallel algorithms (work in progress).

References

1. Alba E., Troya J.M., "A Survey of Parallel Distributed Genetic Algorithms", *Complexity* **4**(4):31-52, 1999.

2. Alba E., Troya J.M., "Analyzing Synchronous and Asynchronous Parallel Distributed Genetic Algorithms", *Future Generation Computer Systems*, 17(4):451-465, January 2001.

3. Bäck T., Fogel D., Michalewicz Z. (eds.), *Handbook of Evolutionary Computation*, Oxford Univ. Press, 1997.

4. Goldberg D.E., Deb K., "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms", in Rawlins G.J.E. (ed.), *Foundations of Genetic Algorithms*, Morgan Kaufmann, 69-93, 1991.

5. Goldberg D.E., Deb K., Horn J., "Massively Multimodality, Deception and Genetic Algorithms", in Männer R., Manderick B. (eds.) *Procs. of PPSN*2, North-Holland, 37-46, 1992.

6. Gordon V.S. and Whitley D., "Serial and Parallel Genetic Algorithms as Function Optimizers", in Forrest S. (ed.) *Procs. of the Fifth ICGA*, 177-183, 1993.

7. Jelasity M., "A Wave Analysis of the Subset Sum Problem", in Bäck T. (ed.) *Procs. of the Seventh ICGA*, 89-96, 1997.

8. Manderick B. and Spiessens P., "Fine-Grained Parallel Genetic Algorithms" in Schaffer J.D. (ed.) *Procs. of the Third ICGA*, 428- 433, 1989.

9. Munetomo M., Takai Y., Sato Y., "An Efficient Migration Scheme for Subpopulation-Based Asynchronous Parallel GAs", *HIER-IS-9301*, Hokkaido University, 1993.

10. Shonkwiler R., "Parallel Genetic Algorithms" in Forrest S. (ed.) *Procs. of the Fifth ICGA*, 199-205, 1993.

11. Stender J., *Parallel Genetic Algorithms: Theory and Applications*, IOS Press, 1993.

12. Syswerda G., "A Study of Reproduction in Generational and Steady-State Genetic Algorithms", in Rawlins G. (ed.) *Foundations of GAs,* 94-101, 1991.

13. Tanese R., "Distributed Genetic Algorithms", in Schaffer J.D. (ed.) *Procs. of the Third ICGA*, 434-439, 1989.

14. Whitley D., "Cellular Genetic Algorithms", in Forrest S. (ed.) *Procs. of the Fifth ICGA*, 658, 1993.