1 Parallel and Distributed Evolutionary Algorithms: A Review

MARCO TOMASSINI
Institute of Computer Science, University of Lausanne, 1015 Lausanne, Switzerland.
E-mail: Marco.Tomassini@di.epfl.ch. Web: www-iis.unil.ch.

1.1 INTRODUCTION

Evolutionary algorithms (EAs) find their inspiration in the evolutionary processes occurring in Nature. The main idea is that in order for a population of individuals to adapt to some environment, it should behave like a natural system; survival, and therefore reproduction, is promoted by the elimination of useless or harmful traits and by rewarding useful behavior. Technically, evolutionary algorithms can be considered as a broad class of stochastic optimization techniques. They are particularly well suited for hard problems where little is known about the underlying search space. An evolutionary algorithm maintains a population of candidate solutions for the problem at hand, and makes it evolve by iteratively applying a (usually quite small) set of stochastic operators, known as *mutation*, *recombination* and *selection*. The resulting process tends to find globally satisfactory, if not optimal, solutions to the problem much in the same way as in Nature populations of organisms tend to adapt to their surrounding environment.

When applied to large hard problems evolutionary algorithms may become too slow. One way to overcome time and size constraints in a computation is to parallelize it. By sharing the workload, it is hoped that an N-processor system will do the job nearly N times faster than a uniprocessor system, thereby allowing researchers to treat larger and more interesting problem instances. Although an N-times speedup is difficult to achieve in practice, evolutionary algorithms are sufficiently regular in their space and time dimensions as to be suitable for parallel and distributed implementations. Furthermore, parallel evolutionary algorithms seem to be more in line with their natural

counterparts and thus might yield algorithmic benefits besides the added computational power.

This chapter is organized as follows. The next section contains a brief introduction to genetic algorithms and genetic programming. This is followed by a short description of the main aspects of parallel and distributed computer architectures. Next, we review several models of parallel evolutionary algorithms, discussing their workings as well as their respective advantages and drawbacks with respect to the computer or network architecture on which they are executed. In the final sections, we review the state of the theory behind parallel evolutionary algorithms and a few unconventional models. A summary and conclusions section ends the chapter. Parallel genetic algorithms have been reviewed in the past by [6] which also includes an interesting account of the early studies in the field.

1.2 GENETIC ALGORITHMS AND GENETIC PROGRAMMING

Evolutionary algorithms are a class comprising several related techniques. Among them we find chiefly *genetic algorithms*, *genetic programming*, *evolution strategies* and *evolutionary programming*. From the point of view of parallel and distributed computation all such techniques roughly offer the same opportunities. We will therefore study parallel implementations of the two more widespread methodologies i.e., genetic algorithms and genetic programming. The following two sections give a brief introduction to both of these in a classical serial setting. For more details on evolutionary algorithms, the reader can consult the two recent reference texts [4, 17].

1.2.1 An introduction to genetic algorithms

Genetic algorithms (GAs) make use of a metaphor whereby an optimization problem takes the place of the environment; feasible solutions are viewed as individuals living in that environment. An individual's degree of adaptation to its surrounding environment is the counterpart of the objective function evaluated on a feasible solution. In the same way, a set of feasible solutions take the place of a population of organisms.

In genetic algorithms, individuals are just strings of binary digits or of some other set of symbols drawn from a finite set. As computer memory is made up of an array of bits, anything can be stored in a computer can also be encoded by a bit string of sufficient length. Each encoded individual in the population may be viewed as a representation, according to an appropriate encoding, of a particular solution to a problem.

A genetic algorithm starts with a population of randomly generated individuals, although it is also possible to use a previously saved population or a

population of individuals encoding for solutions provided by a human expert or by another heuristic algorithm.

Once an initial population has been created, the genetic algorithm enters a loop. At the end of each iteration a new population will have been created by applying a certain number of stochastic operators to the previous population. One such iteration is referred to as a *generation*.

The first operator to be applied is *selection*: in order to create a new intermediate population of n "parents", n independent extractions of an individual from the old population are performed, where the probability for each individual of being extracted is linearly proportional to its fitness. Therefore, above average individuals will expectedly have more copies in the new population, while below average individuals will risk extinction.

Once the population of parents, that is of individuals that have been selected for reproduction, has been extracted, the individuals for the next generation will be produced through the application of a number of reproduction operators, which can involve just one parent (thus simulating asexual reproduction), in which case we speak of mutation, or more parents (thus simulating sexual reproduction), in which case we speak of recombination. In genetic algorithms two reproduction operators are used: crossover and mutation.

To apply crossover, couples are formed with all parent individuals; then, with a certain probability, called crossover rate p_{cross}, each couple actually undergoes crossover: the two bit strings are cut at the same random position and their second halves are swapped between the two individuals, thus yielding two novel individuals, each containing characters from both parents.

After crossover, all individuals undergo mutation. The purpose of mutation is to simulate the effect of transcription errors that can happen with a very low probability (p_{mut}) when a chromosome is duplicated. This is accomplished by flipping each bit in every individual with a very small probability, called mutation rate. In other words, each "0" has a small probability of being turned into a "1" and *vice versa*.

In principle, the above-described loop is infinite, but it can be stopped when a given termination condition specified by the user is met. Possible termination conditions are: a pre-determined number of generations or time has elapsed or a satisfactory solution has been found or no improvement in solution quality has been taking place for a pre-determined number of generations. At this point, it is useful to recall that evolutionary algorithms are stochastic iterative techniques that are not guaranteed to converge although they usually succeed in finding good enough solutions.

The evolutionary cycle can be summarized by the following pseudo-code:

```
generation = 0
seed population
while not termination condition do
    generation = generation + 1
```

```
    calculate fitness
    selection
    crossover(p_cross)
    mutation(p_mut)
  end while
```

1.2.2 An introduction to genetic programming

Genetic programming (GP) is a more recent evolutionary approach which extends the genetic model of learning to the space of programs. It is a major variation of genetic algorithms in which the evolving individuals are themselves computer programs instead of fixed length strings from a limited alphabet of symbols. Genetic programming is a form of *program induction* that allows to automatically discover programs that solve or approximately solve a given task. The present form of GP is principally due to J. Koza [12].

Individual programs in GP might be expressed in principle in any current programming language. However, the syntax of most languages is such that GP operators would create a large percentage of syntactically incorrect programs. For this reason, Koza chose a syntax in prefix form analogous to LISP and a restricted language with an appropriate number of variables, constants and operators defined to fit the problem to be solved. In this way syntax constraints are respected and the program search space is limited. The restricted language is formed by a user-defined *function set* F and *terminal set* T. The functions chosen are those that are *a priori* believed to be useful for the problem at hand, and the terminals are usually either variables or constants. In addition, each function in the function set must be able to accept as arguments any other function return value and any data type in the terminal set T, a property that is called *syntactic closure*. Thus, the space of possible programs is constituted by the set of all possible compositions of functions that can be recursively formed from the elements of F and T.

As an example, suppose that we are dealing with simple arithmetic expressions in three variables. In this case suitable function and terminal sets might be defined as:

$$F = \{+, -, *, /\}$$

and

$$T = \{A, B, C\}$$

and the following are legal programs: $(+(*AB)(/CD)), and (*(-(+AC)B)A)$.

It is important to note that GP does not need to be implemented in the LISP language (though this was the original implementation). Any language that can represent programs internally as parse trees is adequate. Thus, most GP packages today are written in C, C++ or Java rather than LISP.

Programs are represented as trees with ordered branches in which the internal nodes are functions and the leaves are the terminals of the problem. Thus, the examples given above would give raise to the trees in Figure 1.1.

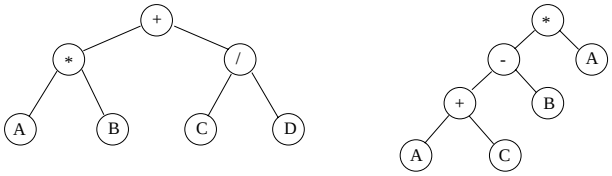

Fig. 1.1 Two GP trees corresponding to the LISP expressions in the text.

Evolution in GP is similar to GAs but different individual representation and genetic operators are used. Once suitable functions and terminals have been determined for the problem at hand, an initial random population of trees (programs) is constructed. From there on the population evolves as with a GA where fitness is assigned after actual execution of the program (individual) and with genetic operators adapted to the tree representation. Fitness calculation is a bit different for programs. In GP we would like to discover a program that satisfies a given number N of predefined input/output relations: these are called the *fitness cases*. For a given program p_i its fitness f_i on the i-th fitness case represents the difference between the output g_i produced by the program and the correct answer G_i for that case. The total fitness $F(p_i)$ is the sum over all N fitness cases of some norm of the cumulated difference:

$$F(p_i) = \sum_{k=1}^{N} \parallel g_k - G_k \parallel . \tag{1.1}$$

Obviously, a better program will have a lower fitness under this definition, and a perfect one will score 0 fitness.

The crossover operation starts by selecting a random crossover point in each parent tree and then exchanging the sub-trees, giving rise to two offspring trees, as shown in Figure 1.2. The crossover site is usually chosen with non-uniform probability, in order to favor internal nodes with respect to leaves. Mutation, when used, is implemented by randomly removing a subtree at a selected point and replacing it with a randomly generated subtree.

One problematic step in GP is the choice of the appropriate language for a given problem. In general, the problem itself suggests a reasonable set of functions and terminals but this is not always the case. Although experimental evidence has shown that good results can be obtained with slightly different choices of F and T, it is clear that the choice of language has an influence on how hard the problem will be to solve with GP. For the time being, there is no guideline for estimating this dependence nor for choosing suitable terminal

and function sets: the choice is left to the sensibility and knowledge of the problem solver.

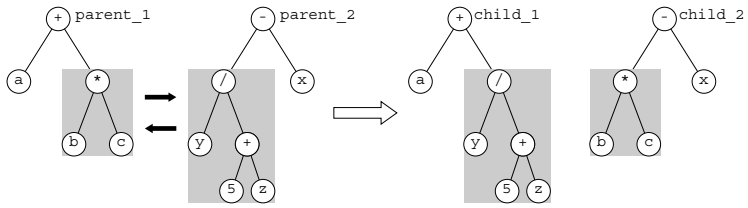

Fig. 1.2 Example of crossover of two genetic programs.

Plain GP works well for problems that are not too complex and that give rise to relatively short programs. To extend GP to more difficult problems some hierarchical principle has to be introduced. In any problem-solving activity hierarchical considerations are needed to produce economically viable solutions. Methods for automatically identifying and extracting useful modules within GP have been discussed by Koza under the name of Automatically Defined Functions (ADF) ([13]), by Angeline and Kinnear [10] and by Rosca [23].

GP is particularly useful for program discovery, i.e. the induction of programs that correctly solve a given problem with the assumption that the form of the program is unknown and that only the desired behavior is given, e.g. by specifying input-output relations. Genetic programming has been successfully applied to a wide variety of problems from many fields, described in [12, 13] and, more recently, in [2,4,10]. In conclusion, GP has been empirically shown to be quite a powerful automatic or semi-automatic program-induction and machine learning methodology.

1.3 PARALLEL AND DISTRIBUTED EVOLUTIONARY ALGORITHMS

Parallel and distributed computing is a key technology in the present days of networked and high-performance systems. The goal of increased performance can be met in principle by adding processors, memory and an interconnection network and putting them to work together on a given problem. By sharing the workload, it is hoped that an N-processor system will give rise to a *speedup* in the computation time. Speedup is defined as the the time it takes a single processor to execute a given problem instance with a given algorithm divided by the time on a N-processor architecture of the same type for the same problem instance and with the same algorithm. Sometimes, a different, more suitable algorithm is used in the parallel case. Clearly, in the ideal case

the maximum speedup is equal to N. If speedup is indeed about linear in the number of processors, then time consuming problems can be solved in parallel in a fraction of the uniprocessor time or larger and more interesting problem instances can be tackled in the same amount of time. In reality things are not so simple since in most cases several overhead factors contribute to significantly lower the theoretical performance improvement expectations. Furthermore, general parallel programming models turn out to be difficult to design due to the large architectural space that they must span and to the resistance represented by current programming paradigms and languages. In any event, many important problems are sufficiently regular in their space and time dimensions as to be suitable for parallel or distributed computing and evolutionary algorithms are certainly among those.

In the next section we present a quick review of parallel and distributed architectures as an introduction to the discussion of parallel and distributed evolutionary algorithms.

1.3.1 Parallel and distributed computer architectures: an overview

Parallelism can arise at a number of levels in computer systems: task level, instruction level or at some lower machine level. Although there are several ways in which parallel architectures may be classified, the standard model of Flynn is widely accepted as a starting point. But the reader should be aware that it is a coarse-grain classification: for instance, even today's serial processors are in fact highly parallel in the way in which they execute instructions, as well as with respect to the memory interface. Even at a higher architectural level, many parallel architectures are in fact hybrids of the base classes. For a comprehensive treatement of the subject, the reader is referred to Hwang's text [11].

Flynn's taxonomy is based on the notion of instruction and data streams. There are four possible combinations conventionally called *SISD* (single instruction, single data stream), *SIMD* (single instruction, multiple data stream), *MISD* (multiple instruction, single data stream) and *MIMD* (multiple instruction, multiple data stream). Figure 1.3 schematically depicts the three most important model architectures.

The SISD architecture corresponds to the classical mono-processor machine such as the typical PC or workstation. As stated above, there is already a great deal of parallelism in this architecture at the instruction level (pipelining, superscalar and very long instruction execution). This kind of parallelism is "invisible" to the programmer in the sense that it is built-in in the hardware or must be exploited by the compiler.

In the SIMD architecture the same instruction is broadcast to all processors. The processors operate in lockstep executing the given instruction on different data stored in their local memories (hence the name: single instruction, multiple data). The processor can also remain idle, if this is appropri-

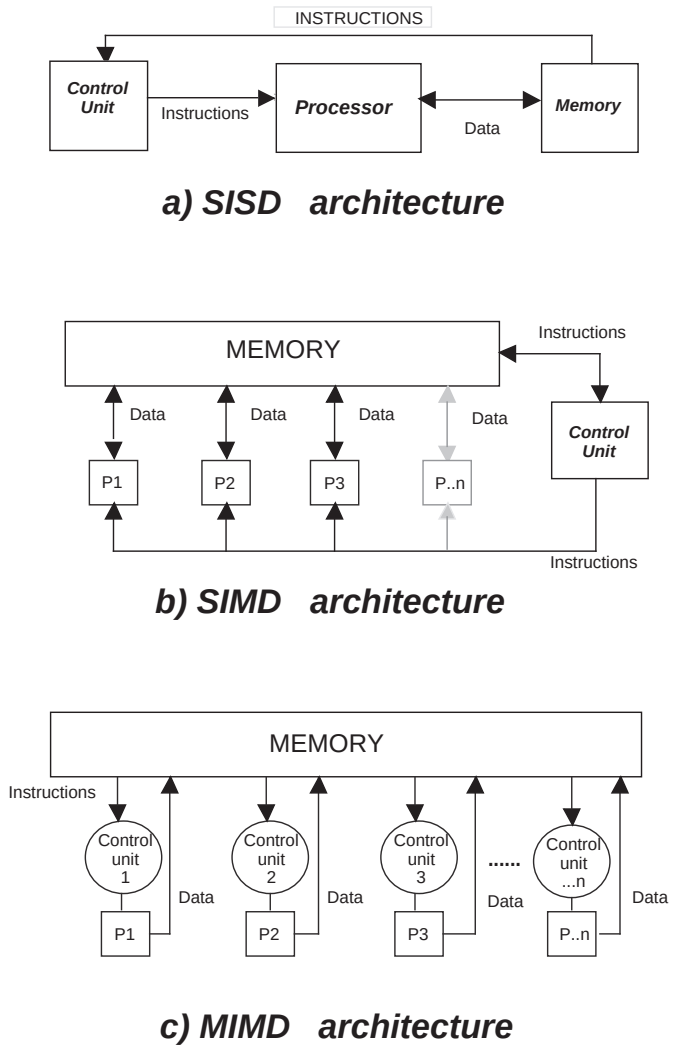

Fig. 1.3 Flynn's model of parallel architectures. The shared memory model is depicted; In the SIMD and MIMD case memory can also be distributed (see text).

ate. SIMD machines exploit *spatial* parallelism that may be present in a given problem and are suitable for large, regular data structures. If the problem domain is spatially or temporally irregular, many processors must remain idle at a given time step since different operations are called for in different regions. Obviously, this entails a serious loss in the amount of parallelism that can be

exploited. Another type of SISD computer is the vector processor which is specialized in the pipelined execution of vector operations. This is the typical component of supercomputers but it is not a particularly useful architecture for evolutionary algorithms.

In the MIMD class of parallel architectures multiple processors work together through some form of interconnection. Different programs and data can be loaded into different processors which means that each processor can execute different instructions at any given point in time. Of course, usually the processors will require some form of synchronization and communication in order to cooperate on a given application. This class is the more generally useful and most commercial parallel and distributed architectures belong to it.

There has been little interest up to now in the MISD class since it does not lend itself to readily exploitable programming constructs. One type of architecture of this class that enjoys some popularity are the so-called *systolic arrays* which are used in specialized applications such as signal processing.

Another important design decision is whether the system memory spans a single address space or it is distributed into separated chunks that are addressed independently. The first type is called *shared memory* while the latter is known as *distributed memory*. This is only a logical subdivision independent of how the memory is physically built.

In shared memory multiprocessors all the data are accessible by all the processors. This poses some design problems for data integrity and for efficiency. Fast cache memories next to the processors are used in order to speedup memory access to often-used items. Cache coherency protocols are then needed to insure that all processors see the same value for a given piece of data.

Distributed memory multicomputers is also a popular architecture which is well suited to most parallel workloads. Since the address spaces of each processor are separate, communication between processors must be implemented through some form of message passing. To this class belong networked computers, sometimes called computer *clusters*. This kind of architecture is interesting for several reasons. First of all, it has a low cost since already existing local networks of workstations can be used just by adding a layer of communication software to implement message passing. Second, the machines in these networks usually feature up-to-date off-the-shelf processor and standard software environments which make program development easier. The drawbacks are that parallel computing performance is limited by comparatively high communication latencies and by the fact that the machines have different workloads at any given time and are possibly heterogeneous. Nevertheless, problems that do not need frequent communication are suitable for this architecture. Moreover, some of the drawbacks can be overcome by using networked computers in dedicated mode with a high-performance communication switch. Although this solution is more expensive, it can be cost-effective with respect to specialized parallel machines.

Finally, one should not forget that the World Wibe Web provides important infrastructures for distributed computation. As it implements a general distributed computing model, this Web technology can be used for parallel computing and for both computing and information related applications. Harnessing the Web or some other geographically distributed computer resource so that it looks like a single computer to the user is called *metacomputing*. The concept is very attractive but many challenges remain. In fact, in order to transparently and efficiently distribute a given computational problem over the available resources without the user taking notice requires advances in the field of user interfaces and in standardized languages, momitoring tools and protocols to cope with the problem of computer heterogeneity and uneven network load. The Java environment is an important step in this direction.

We will see in the next section how the different architectures can be used for distributed evolutionary computing.

1.3.2 Parallel and distributed evolutionary algorithms models

There are two main reasons for parallelizing an evolutionary algorithm: one is to achieve time savings by distributing the computational effort and the second is to benefit from a parallel setting from the algorithmic point of view, in analogy with the natural parallel evolution of spatially distributed populations.

A first type of parallel evolutionary algorithm makes use of the available processors or machines to run independent problems. This is trivial, as there is no communication between the different processes and for this reason it is sometimes called an *embarassingly parallel* algorithm. This extremely simple method of doing simultaneous work can be very useful. For example, this setting can be used to run several versions of the same problem with different initial conditions, thus allowing gathering statistics on the problem. Since evolutionary algorithms are stochastic in nature, being able to collect this kind of statistics is very important. This method is in general to be preferred with respect to a very long single run since improvements are more difficult at later stages of the simulated evolution. Other ways in which the model can be used is to solve N different versions of the same problem or to run N copies of the same problem but with different GA parameters, such as crossover or mutation rates. Neither of the above adds anything new to the nature of the evolutionary algorithms but the time savings can be large.

We now turn to genuine parallel evolutionary algorithms models. There are several possible levels at which an evolutionary algorithm can be parallelized: the population level, the individual level or the fitness evaluation level. Moreover, although genetic algorithms and genetic programming are similar in many respects, the differences in the individual representation make genetic programming a little bit different when implemented in parallel. The next section describes the parallelization of the fitness evaluation while the two following sections treat the population and the individual cases respectively.

1.3.3 Global parallel evolutionary algorithms

Parallelization at the fitness evaluation level does not require any change to the standard evolutionary algorithm since the fitness of an individual is independent of the rest of the population. Moreover, in many real-world problems, the calculation of the individual's fitness is by far the most time consuming step of the algorithm. This is also a necessary condition in order for the communication time to be small in comparison to the time spent in computations. In this case an obvious approach is to evaluate each individual fitness simultaneously on a different processor. A *master* process manages the population and hands out individuals to evaluate to a number of *slave* processes. After the evaluation, the master collects the results and applies the genetic operators to produce the next generations. Figure 1.4 graphically depicts this architecture. If there are more individuals than processors, which is often the case, then the individuals to be evaluated are divided as evenly as possible among the available processors. This architecture can be implemented on both shared memory multiprocessors as well as distributed memory machines, including networked computers. For genetic algorithms it is assumed that fitness evaluation takes about the same time for any individual. The other parts of the algorithm are the same as in the sequential case and remain centralized. The following is an informal description of the algorithm:

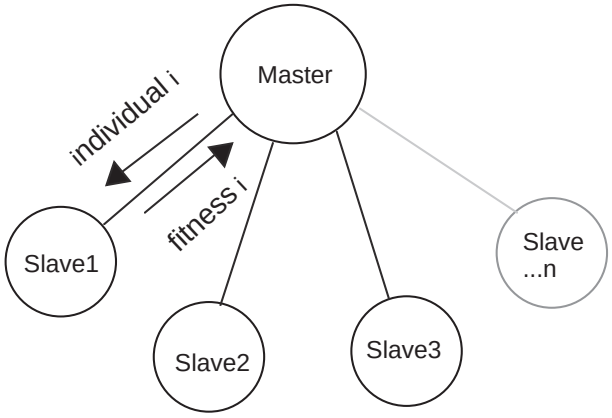

Fig. 1.4 A schematic view of the master-slave model.

```
produce an initial population of individuals
for all individuals do in parallel
   evaluate the individual's fitness
```

```
end parallel for
while not termination condition do
   select fitter individuals for reproduction
   produce new individuals
   mutate some individuals
   for all individuals do in parallel
      evaluate the individual's fitness
   end parallel for
   generate a new population by inserting some new good
   individuals and by discarding some old bad individuals
end while
```

In the GP case, due to the widely different sizes and complexities of the individual programs in the population different individuals may require different times to evaluate. This in turn may cause a load imbalance which decreases the utilization of the processors. Mouloud *et al* [21] implemented a simple method for load-balancing the system on a distributed memory parallel machine and observed nearly linear speedup. However, load balancing can be obtained for free if one gives up generational replacement and permits steady-state reproduction instead (see section 1.3.6).

1.3.4 Island distributed evolutionary algorithms

We now turn to individual or population-based parallel approaches for evolutionary algorithms. All these find their inspiration in the observation that natural populations tend to possess a *spatial* structure. As a result, so-called *demes* make their appearance. Demes are semi-independent groups of individuals or subpopulations having only a loose coupling to other neighboring demes. This coupling takes the form of the slow migration or diffusion of some individuals from one deme to another. A number of models based on spatial structure have been proposed. The two most important categories are the *island* and the *grid* models.

The *island* model [9] features geographically separated subpopulations of relatively large size. Subpopulations may exchange information from time to time by allowing some individuals to migrate from one subpopulation to another according to various patterns. The main reason for this approach is to periodically reinject diversity into otherwise converging subpopulations. As well, it is hoped that to some extent, different subpopulations will tend to explore different portions of the search space. When the migration takes place between nearest neighbor subpopulations the model is called *stepping stone*. Within each subpopulation a standard sequential evolutionary algorithm is

executed between migration phases. Several migration topologies have been used: the ring structure, 2-d and 3-d meshes, hypercubes and random graphs being the most common. Figure 1.5 schematically depicts this distributed model and the following is a high-level algorithmic description of the process:

```
initialize P subpopulations of size N each
generation number := 1
while not termination condition do
   for each subpopulation do in parallel
      evaluate and select individuals by fitness
      if generation number mod frequency = 0 then
         send K<N best individuals to
         a neighboring subpopulation
         receive K individuals from a
         neighboring population
         replace K individuals in
         the subpopulation
      end if
      produce new individuals
      mutate individuals
   end parallel for
   generation number := generation number + 1
end while
```

Here *frequency* is the number of generations before an exchange takes place. Several individual replacement policies have been described in the literature. One of the most common is for the migrating `K` individuals to displace the `K` worst individuals in the subpopulation. It is to be noted that the subpopulation size, the frequency of exchange, the number of migrating individuals, and the migration topology are all new parameters of the algorithm that have to be set in some way. At present there is no rigorous way for choosing them. However, several works have empirically arrived at rather similar topologies and parameter values [9, 27]. However, some new theoretical work is also shedding some light on these issues (see section 1.3.8 below).

1.3.5 Cellular genetic algorithms

In the *grid* or *fine-grained* model [16] individuals are placed on a large toroidal (the ends wrap around) one or two-dimensional grid, one individual per grid

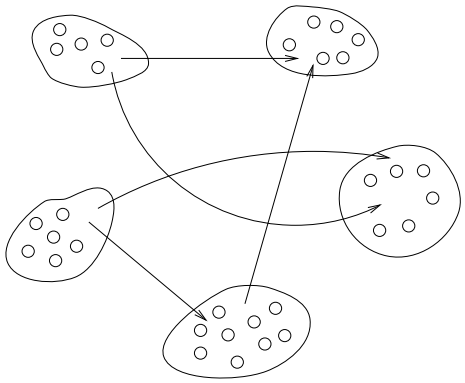

Fig. 1.5 The *island* model of semi-isolated populations.

location (see Figure 1.6). The model is also called *cellular* because of its similarity with cellular automata with stochastic transition rules [28, 29].

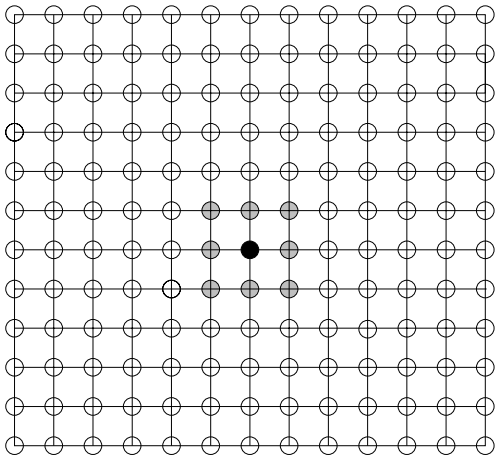

Fig. 1.6 A 2-D spatially extended population of individuals. A possible neighborhood of an individual (black) is marked in gray.

Fitness evaluation is done simultaneously for all individuals and selection, reproduction and mating take place locally within a small neighborhood. In time, semi-isolated niches of genetically homogeneous individuals emerge across the grid as a result of slow individual diffusion. This phenomenon is called *isolation by distance* and is due to the fact that the probability of in-

teraction of two individuals is a fast-decaying function of their distance. The following is a pseudo-code description of a grid evolutionary algorithm.

```
for each cell i in the grid do in parallel
   generate a random individual i
end parallel for
while not termination condition do
   for each cell i do in parallel
      evaluate individual i
      select a neighboring individual k
      produce offspring from i and k
      assign one of the offspring to i
      mutate i with probability p_mut
   end parallel for
end while
```

In the preceding description the neighborhood is generally formed by the four or eight nearest neighbors of a given grid point (see Figure 1.6). In the 1-D case a small number of cells on either side of the central one is taken into account. The selection of an individual in the neighborhood for mating with the central individual can be done in various ways. Tournament selection is commonly used since it matches nicely the spatial nature of the system. Local tournament selection extracts k individuals from the population with uniform probability but without re-insertion and makes them play a "tournament", which is won, in the deterministic case, by the fittest individual among the participants. The tournament may be probabilistic as well, in which case the probability for an individual to win it is generally proportional to its fitness. This makes full use of the available parallelism and is probably more appropriate if the biological methaphor is to be followed. Likewise, the replacement of the original individual can be done in several ways. For example, it can be replaced by the best among itself and the offspring or one of the offspring can replace it at random. The model can be made more dynamical by adding provisions for longer range individual movement through random walks, instead of having individuals interacting exclusively with their nearest neighbors [28]. A noteworthy variant of the cellular model is the so-called *cellular programming algorithm* [26]. Cellular programming has been extensively used to evolve cellular automata for performing computational tasks.

1.3.6 Implementation and experimental observations on parallel EAs

The master-slave global fitness parallelization model can be implemented on both shared memory multiprocessors and distributed memory multicomputers, including workstation clusters, and it is well adapted when fitness calculation takes up most of the computation time. However, it is well know that distributed memory machines have better scaling behavior. This means that performance is relatively unaffected if more processors (and memory) are added *and* larger problem instances are tackled. In this way, computation and communication times remain well balanced. Shared memory machines, on the other hand, suffer from memory acces contention, especially if global memory access is through a single path such as a bus. Cache memory alleviates the problem but it does not solve it.

Although both island and cellular models can be implemented on serial machines, thus comprising useful variants of the standard globally communicating GA, they are ideally suited for parallel computers. From an implementation point of view, coarse-grained island models, where the ratio of computation to communication is high, are more adapted to distributed memory multicomputers and for clusters of workstations. Recently, some attempts have been made at using Web and Java-based computing for distributed evolutionary algorithms [20]. The advantages and the drawbacks of this approach have been discusses in section 1.3.1.

Massively parallel SIMD (Single Instruction Multiple Data) machines such as the Maspar and the Connection Machine CM-200 are appropriate for cellular models, since the necessary local communication operations, though frequent, are very efficiently implemented in hardware. Genetic programming is not suitable for cellular models since individuals may widely vary in size and complexity. This makes cellular implementations of GP difficult both because of the amount of local memory needed to store individuals as well as for efficiency reasons (e.g., sequential execution of different branches of code belonging to individuals stored on different processors).

In general, it has been found experimentally that parallel genetic algorithms, apart from being significantly faster, may help in alleviating the premature convergence problem and are effective for multimodal optimization. This is due to the larger total population size and to the relative isolation of the spatial regions where solutions start to co-evolve. Both of these factors help to preserve diversity while at the same time promoting local search. There are several results in the literature that support these conclusions, see for instance [14,15,18,27]. As in the sequential case, the effectiveness of the search can be improved by permitting hill-climbing, i.e. local improvement around promising search points [18]. It has even been reported that for the island model [1,14] *superlinear* speedup has been achieved in some cases. Superlinear speedup means getting more than N-fold acceleration with N processors with respect to the uniprocessor time. While superlinear speedup is strictly

impossible for deterministic algorithms, it becomes possible when there is a random element in the ordering of the choices made by an algorithm. This has been shown to be the case in graph and tree search problems where, depending on the solution and the way in which the search space is subdivided, parallel search may be more than N-fold effective. The same phenomenon may occur in all kind of stochastic, Monte-Carlo type algorithms, including evolutionary algorithms. The net result is that in the multi-population case oftentimes *fewer* evaluations are needed to get the same solution quality than for the single population case with the same total number of individuals.

For coarse-grain, island-based parallel genetic programming the situation is somewhat controversial. Andre and Koza [1] reported excellent results on a GP problem implemented on a transputer network (the 5-parity problem). More recently, W. Punch [22] performed extensive experiments on two other problems: the "ant" problem and the "royal tree" problem. The first is a standard machine learning test problem [12], while the latter is a contructive benchmark for GP. Punch found that the multiple-population approach did not help in solving those problems. Although the two chosen problems were purposedly difficult for distributed GP approaches, these results point to the fact that the parallel, multi-population GP dynamics is not yet well understood. This is not surprising, given that there is no solid theoretical basis yet even for the standard sequential GP.

Until now only the spatial dimension entered into the picture. If we take into account the temporal dimension as well, we observe that parallel evolutionary algorithms can be *synchronous* or *asynchronous*. Island models are in general synchronous, using SPMD (Single Program Multiple Data), coarse-grain parallelism in which communication phases synchronize processes. This is not necessary and experiments have been carried out with asynchronous EAs in which subpopulations evolve at their own pace and exchange individuals only when some internally measured level of convergence has been attained [19]. This avoids constraining all co-evolving populations to swap individuals at the same time irrespective of subpopulation evolution and increases population diversity. Asynchronous behavior can be easily and conveniently obtained in island models by using *steady-state* reproduction instead of generational reproduction, although generational models can also be made to work asynchronously. In steady-state reproduction a small number of offspring replace other members of the population as soon as they are produced instead of replacing all the population at once after all the individuals have gone through a whole evaluation-selection-recombination mutation cycle [17]. For genetic programming the asynchronous setting also helps for the load-balancing problem caused by the variable size and complexity of GP programs. By independently and asynchronously evaluating each program, different nodes can proceed at different speeds without any need for synchronization [1].

Fine-grained parallel EAs are fully synchronous when they are implemented on SIMD machines and are an example of data-parallelism. Asynchronous

behavior can be easily simulated on a sequential machine but I do not know of any example of asynchronous cellular evolutionary algorithms.

Figure 1.7 schematically depicts the main parallel evolutionary algorithm classes according to their space and time dimensions. The reader should be aware that this is only a first cut at a general classification. Although the main classes are covered, a number of particular and hybrid cases exist in practice, some of which will be described in the next section.

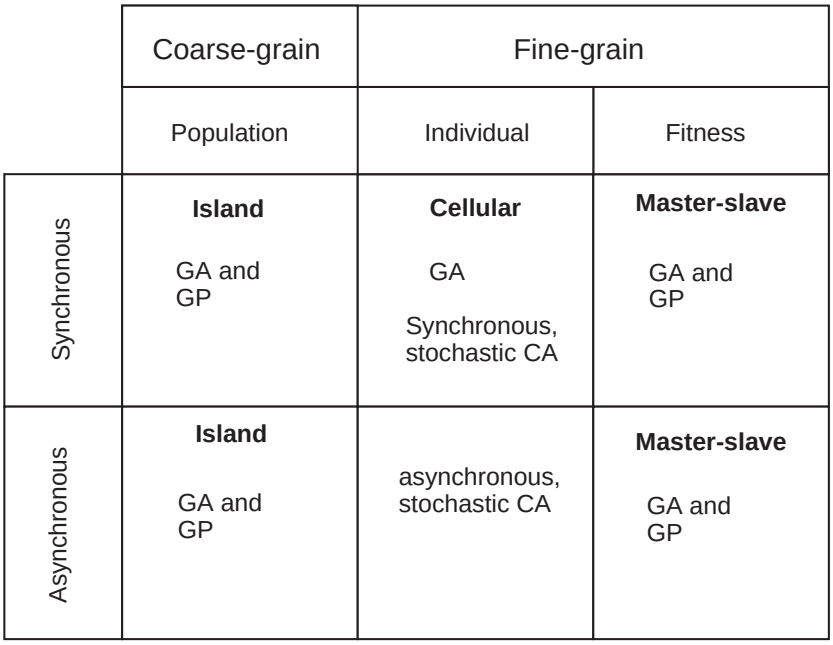

Fig. 1.7 Parallel and distributed algorithms classes according to their space and time behavior.

1.3.7 Non-standard parallel EAs models

There exist proposals for parallel evolutionary algorithms that do not fit into any of the classes that have been described in the previous sections. From the topological and spatial point of view, the methods of parallelization can be combined to give *hybrid* models. Two-level hybrid algorithms are sometimes used. The island model can be used at the higher level, while another fine-grained model can be combined with it at the lower level. for example, one might consider an island model in which each island is structured as a grid of individuals interacting locally. Another possibility is to have an island model

at the higher level where each island is a global parallel EA in which fitness calculation is parallelized. Other combinations are also possible and a more detailed description can be found in [6].

Although these combinations may give rise to interesting and efficient new algorithms, they have the drawback that even more new parameters have to be introduced to account for the more complex topological structure.

So far, we have made the implicit hypothesis that the genetic material as well as the evolutionary conditions such as selection and crossover methods were the same for the different subpopulations in a multi-population algorithm. If one gives up these constraints and allows different subpopulations to evolve with different parameters and/or with different individual representations for the same problem then new distributed algorithms may arise. We will name these algorithms *non-uniform* parallel EAs. One recent example of this class is the *Injection island GA* (iiGA) of Lin *et al* [14]. In an iiGA there are multiple populations that encode the same problem using a different representation size and thus different resolutions in different islands. The migration rules are also special in that migration is only one-way, going from a low-resolution to a high resolution node. According to Lin *et al*, such a hierarchy has a number of advantages with respect to a standard island algorithm.

In biological terms, having different semi-isolated populations in which evolution takes place at different stages and with different, but compatible genetic material makes sense from a biological point of view: it might also be advantageous for evolutionary algorithms. Genetic programming in particular might benefit from these ideas since the choice of the function and terminal sets is often a fuzzy and rather subjective issue.

1.3.8 Theoretical work

Theoretical studies of evolutionary algorithms have been performed mostly for the case of binary-coded, generational genetic algorithms with standard genetic operators and selection [3]. The standard form of evolution strategies and evolutionary programming is also sufficiently well understood. However, the dynamical behavior of genetic programming and non-standard GAs is much less clear, although progress is being made.

Parallel evolutionary algorithms are even more difficult to analyze because they introduce a number of new parameters such as migration rates and frequencies, subpopulation topology, grid neighborhood shape and size among others. As a consequence, there seems to be lacking a better understanding of their working. Nevertheless, performance evaluation models have been built for the simpler case of the master-slave parallel fitness evaluation in the GA case [7] and for GP [21]. As well, some limiting simplified cases of the island approach have been modeled: a set of isolated demes and a set of fully connected demes [5]. These are only first steps towords establishing more principled ways for choosing suitable parameters for parallel EAs.

In another study, Whitley *et al* [30] have presented an abstract model and made experiments of when one might expect the island model to out-perform single population GAs on separable problems. Linear separable problems are those problems in which the evaluation function can be expressed as a sum of independent nonlinear contributions. Although the results are not clear-cut, as the authors themselves acknowledge, there are indications that partitioning the population may be advantageous.

There have also been some attempts at studying the behavior of cellular genetic algorithms. For instance, Sarma and De Jong [25] investigated the effects of the size and shape of the cellular neighborhood on the selection mechanism. They found that the size of the neighborhood is a parameter that can be used to control the selection pressure over the population. In another paper [24] the convergence properties of a typical cellular algorithm were studied using the formalism of probabilistic cellular automata and Markov chains. The authors reached some conclusions concerning global convergence and the influence of the neighborhood size on the quality and speed at which solutions can be found. Unfortunately, their algorithm is not wholly standard since it includes a special recipe for introducing a form of elitism into the population. Recently, Capcarrere *et al* [8] presented a number of statistical measures that can be applied to cellular algorithms in order to understand their dynamics. The proposed statistics were demonstrated on the specific example of the evolution of non-uniform cellular automata but they can be applied generally.

1.4 SUMMARY AND CONCLUSIONS

If one is to follow the biological methaphor, parallel and distributed evolutionary algorithms seem more natural than their sequential counterparts. When implemented on parallel computer architectures they offer the advantage of increased performance due to the execution of simultaneous work on different machines. Moreover, genuine evolutionary parallel models such as the island or the cellular model constitute new useful varieties of evolutionary algorithms and there is experimental evidence that they can be more effective as problem solvers. This may be due to several factors such as independent and simultaneous exploration of different portions of the search space, larger total population sizes, and the possibility of delaying the uniformisation of a population by migration and diffusion of individuals. These claims have not been totally and unambiguously confirmed by the few theoretical studies that have been carried out to date. It is also unclear what is the kind of problem that can benefit more from a parallel/distributed setting, although separable and multimodal problems seem good candidates.

One drawback of parallel and evolutionary algorithms is that they are more complicated than the sequential versions. Since a number of new parameters must be introduced such as communication topologies and migration/diffusion

policies, the mathematical modeling of the dynamics of these algorithms is very difficult. Only a few studies have been performed to date and even those have been mostly applied to simplified, more tractable models. Clearly, a large amount of work remains to be done on these issues before a clearer picture starts to emerge.

The effectiveness of parallel and distributed EAs has also been shown in practice in a number of industrial and commercial applications. Because of lack of space I cannot review these applications here, but it is clear that for some hard problems parallelism can be effective. Some real-life problems may need days or weeks of computing time to solve on serial machines. Although the intrinsic complexity of a problem cannot be lowered by using a finite amount of computing resources in parallel, the use of parallelism often allows to reduce these times to reasonable amounts. This can be very important in an industrial or commercial setting where the time to solution is instrumental for decision making and competitivity. Parallel and evolutionary algorithms have been used successfully in operations research, engineering and manufacturing, finance, VLSI and telecommunication problemes among others. The interested reader can find out more about these applications by consulting the proceedings of the major conferences in the field.

I am indebted to my colleagues B. Chopard, M. Sipper and A. Tettamanzi for stimulating discussions and to R. Gomez for his help with some of the figures appearing in this work.

References

1. D. Andre and J. R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2*, Cambridge, MA, 1996. The MIT Press.
2. P.J. Angeline and K.E. Kinnear Jr. (Eds.). *Advances in Genetic Programming 2*. The MIT Press, Cambridge, Massachusetts, 1996.
3. T. Bäck. *Evolutionary algorithms in theory and practice*. Oxford University Press, Oxford, 1996.
4. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic programming, An Introduction*. Morgan Kaufmann, San Francisco CA, 1998.
5. E. Cantú-Paz. Designing efficient master-slave parallel genetic algorithms. Technical Report 95004, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL, 1997.
6. E. Cantú-Paz. A survey of parallel genetic algorithms. Technical Report 97003, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL, 1997.
7. E. Cantú-Paz and D. E. Goldberg. Modeling idealized bounding cases of parallel genetic algorithms. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 456–462. Morgan Kaufmann, San Francisco, CA, 1997.
8. M. Capcarrere, A. Tettamanzi, M. Tomassini, and M. Sipper. Studying parallel evolutionary algorithms: the cellular programming case. In A. Eiben, T. Bäck, M. Shoenauer, and H.P. Schwefel, editors, *Parallel Problem Solving from Nature*, pages 573–582. Springer, 1998.
9. J. P. Cohoon, S. U. Hedge, W. N. Martin, and D. Richards. Punctuated equilibria: A parallel genetic algorithm. In J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, page 148. Lawrence Erlbaum Associates, 1987.
10. K.E. Kinnear Jr. (Ed.). *Advances in Genetic Programming*. The MIT Press, Cambridge, Massachusetts, 1994.
11. K. Hwang. *Advanced Computer Architecture*. Mc Graw-Hill, New York, 1993.
12. J. R. Koza. *Genetic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

13. J. R. Koza. *Genetic Programming II.* The MIT Press, Cambridge, Massachusetts, 1994.

14. S. C. Lin, W. F. Punch, and E. D. Goodman. Coarse-grain parallel genetic algorithms: Categorization and a new approach. In *Sixth IEEE SPDP*, pages 28–37, 1994.

15. A. Loraschi, A. Tettamanzi, M. Tomassini, and P. Verda. Distributed genetic algorithms with an application to portfolio selection problems. In *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 384–387, Wien, New York, 1995. Springer-Verlag.

16. B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, page 428. Morgan Kaufmann, 1989.

17. Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, Heidelberg, third edition, 1996.

18. H. Mühlenbein, M. Schomish, and J. Born. The parallel genetic algorithm as a function optimizer. *Parallel Computing*, 17:619–632, 1991.

19. M. Munetomo, Y. Takai, and Y. Sato. An efficient migration scheme for subpopulation-based ansynchronously parallel genetic algorithms. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, page 649. Morgan Kaufmann Publishers, San Mateo, California, 1993.

20. P. Nangsue and S. E. Conry. An agent-oriented, massively distributed parallelization model of evolutionary algorithms. In J. Koza, editor, *Late Breaking Papers, Genetic Programming 1998*, pages 160–168. Stanford University, 1998.

21. M. Oussaidene, B. Chopard, O. Pictet, and M. Tomassini. Parallel genetic programming and its application to trading model induction. *Parallel Computing*, 23:1183–1198, 1997.

22. W. Punch. How effective are multiple poplulations in genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogeland M. Garzon, D. Goldberg, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 308–313, San Francisco, CA, 1998. Morgan Kaufmann.

23. J. Rosca and D. Ballard. Discovery of subroutines in genetic programming. In P.J. Angeline and K.E. Kinnear Jr. (Eds.), editors, *Advances in Genetic Programming 2.* Cambridge, Massachusetts, 1996.

24. G. Rudolph and J. Sprave. A cellular genetic algorithm with self-adjusting acceptance thereshold. In *In First IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, pages 365–372, London, 1995. IEE.

25. J. Sarma and K. De Jong. An analysis of the effect of the neighborhood size and shape on local selection algorithms. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 236–244. Springer-Verlag, Heidelberg, 1996.

26. M. Sipper. *Evolution of Parallel Cellular Machines: The Cellular Programming Approach.* Springer-Verlag, Heidelberg, 1997.

27. T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer Science*, page 176, Heidelberg, 1991. Springer-Verlag.

28. M. Tomassini. The parallel genetic cellular automata: Application to global function optimization. In R. F. Albrecht, C. R. Reeves, and N. C. Steele, editors, *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 385–391. Springer-Verlag, 1993.

29. D. Whitley. Cellular genetic algorithms. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, page 658. Morgan Kaufmann Publishers, San Mateo, California, 1993.

30. D. Whitley, S. Rana, and R. B. Heckendorn. Island model genetic algorithms and linearly separable problems. In D. Corne and J. L. Shapiro, editors, *Evolutionary Computing: Proceedings of the AISB Workshop, Lecture notes in computer science, vol. 1305*, pages 109–125. Springer-Verlag, Berlin, 1997.