Journal of Heuristics, 10: 357–380, 2004

ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics*

S. CAHON
N. MELAB
E.-G. TALBI
Laboratoire d'Informatique Fondamentale de Lille, UMR CNRS 8022, Cité Scientifique, 59655 - Villeneuve d'Ascq Cedex, France
email: cahon@lifl.fr
email: melab@lifl.fr
email: talbi@lifl.fr

Submitted in September 2003 and accepted by Enrique Alba in March 2004 after 1 revision

Abstract

In this paper, we present the ParadisEO white-box object-oriented framework dedicated to the reusable design of parallel and distributed metaheuristics (PDM). ParadisEO provides a broad range of features including evolutionary algorithms (EA), local searches (LS), the most common parallel and distributed models and hybridization mechanisms, etc. This high content and utility encourages its use at European level. ParadisEO is based on a clear conceptual separation of the solution methods from the problems they are intended to solve. This separation confers to the user a maximum code and design reuse. Furthermore, the fine-grained nature of the classes provided by the framework allow a higher flexibility compared to other frameworks. ParadisEO is of the rare frameworks that provide the most common parallel and distributed models. Their implementation is portable on distributed-memory machines as well as on shared-memory multiprocessors, as it uses standard libraries such as MPI, PVM and PThreads. The models can be exploited in a transparent way, one has just to instantiate their associated provided classes. Their experimentation on the radio network design real-world application demonstrate their efficiency.

Key Words: metaheuristics, design and code reuse, parallel and distributed models, object-oriented frameworks, performance and robustness

1. Introduction

In practice, combinatorial optimization problems are often NP-hard, CPU time-consuming, and evolve over time. Unlike exact methods, metaheuristics allow to tackle large-size problems instances by delivering satisfactory solutions in a reasonable time. Metaheuristics are general-purpose heuristics that split in two categories: evolutionary algorithms (EA) and local search methods (LS). These two families have complementary characteristics: EA allow

*This work is a part of the current national joint grid computing project ACI-GRID DOC-G (Défis en optimization Combinatoire sur Grilles). It includes research teams from different laboratories: OPAC from LIFL, OPALE from PRISM and O2 and P3-ID from IMAG. The project is supported by the French government.

a better exploration of the search space, while LS have the power to intensify the search in promising regions. Their hybridization allows to deliver robust and better solutions (Talbi, 2002).

Although serial metaheuristics have a polynomial temporal complexity, they remain unsatisfactory for industrial problems. Parallel and distributed computing is a powerful way to deal with the performance issue of these problems. Numerous parallel and distributed metaheuristics (PDM) and their implementations have been proposed, and are available on the Web. They can be reused and adapted to his/her own problems. However, the user has to deeply examine the code and rewrite its problem-specific sections. The task is tedious, error-prone, takes along time and makes harder the produced code maintenance. A better way to reuse the code of existing PDM is the reuse through libraries (Wall, 1999). These are often more reliable as they are more tested and documented. They allow a better maintainability and efficiency. However, libraries do not allow the reuse of design.

An approach devoted to the design and code reuse is based is the framework-based approach (Johnson and Foote, 1988). A framework may be object-oriented (OO), and defined as a set of classes that embody an abstract design for solutions to a family of related problems (Fink, Vo, and Woodruff, 1999). Frameworks are well-known in the software engineering literature. In this paper, we focus on those related to PDM for discrete optimization problems. These allow the reuse of the design and implementation of a whole PDM. They are based on a strong conceptual separation of the invariant (generic) part of PDM and their problem-specific part. Therefore, they allow the user to redo very little code.

Several frameworks for PDM have been proposed in the literature. Most of them focus only either on EA (Arenas et al., 2002; Luke et al., 2002; Costa, Lopes, and Silva, 1999; Goodman, 1994; Gagné, Parizeau, and Dubreuil, 2003) or LS (Di Gaspero and Schaerf, 2001; Michel and Van Hentenryck, 2001). Only few frameworks are dedicated on the design of both EA and LS (Alba and the MALLBA Group, 2002; Krasnogor and Smith, 2000). On the other hand, existing frameworks either do not provide parallelism at all or supply just a little. In this paper, we propose a new framework for PDM, named *Parallel and Distributed Evolving Objects (ParadisEO)*. ParadisEO is an extension of the Evolving Objects (EO) framework devoted to the design of serial EA. The framework has been developed within the scope of an European joint work (Keijzer et al., 2001). The extensions include LS, parallel and distributed computing, hybridization, and other features that are not the focus of this paper. Unlike most of other existing frameworks, it provides a wide range of reusable features and mechanisms related to PDM. First, it allows the design of both EA and LS and provides different reusable mechanisms for their hybridization. Second, for each family it supplies a large variety of variation operators, evaluation functions, etc. Third, it implements the most common parallel and distributed models, and allows the user to use them in a transparent and portable way. Finally, the fine-grained evolving objects of ParadisEO allow a higher flexibility, adaptability and code reuse.

ParadisEO has been experimented on various academic problems (Traveling Salesman Problem, Graph Coloring Problem, etc.) and real-world applications (spectroscopic data mining (Cahon, Talbi, and Melab, 2003), radio network design, etc.). In this paper, we experiment the framework on the radio network design.

The rest of the paper is organized as follows: Section 2 presents a survey of PDM. In Sections 3 and 4 we describe the architecture and implementation of ParadisEO. We also detail the different parallel and distributed models, and hybridization mechanisms implemented in the framework. Section 5 shows and comments some experimental results on the applications quoted above. In Section 6 we present the different research works related to the frameworks of PDM. Section 7 concludes the paper and highlights the main perspectives of the presented work.

2. Parallel and distributed metaheuristics

2.1. Parallel distributed evolutionary algorithms

Evolutionary Algorithms (Holland, 1975) (EA) are based on the iterative improvement of a population of solutions. At each step, individuals are selected, paired and recombined in order to generate new solutions that replace other ones, and so on. As the algorithm converges, the population is mainly composed of individuals well adapted to the "environment", for instance the problem. The main features that characterize EA are the way the population is initialized, the selection strategy (deterministic/stochastic) by fostering "good" solutions, the replacement strategy that discards individuals, and the continuation/stopping criterion to decide whether the evolution should go on or not.

Basically, three major parallel and distributed models for EA can been distinguished (see figure 1): the island (a)synchronous cooperative model, the parallel evaluation of the population, and the distributed evaluation of a single solution.

– *Island (a)synchronous cooperative model.* Different EA are simultaneously deployed to cooperate for computing better and robust solutions. They exchange in an asynchronous way genetic stuff to diversify the search. The objective is to allow to delay the global

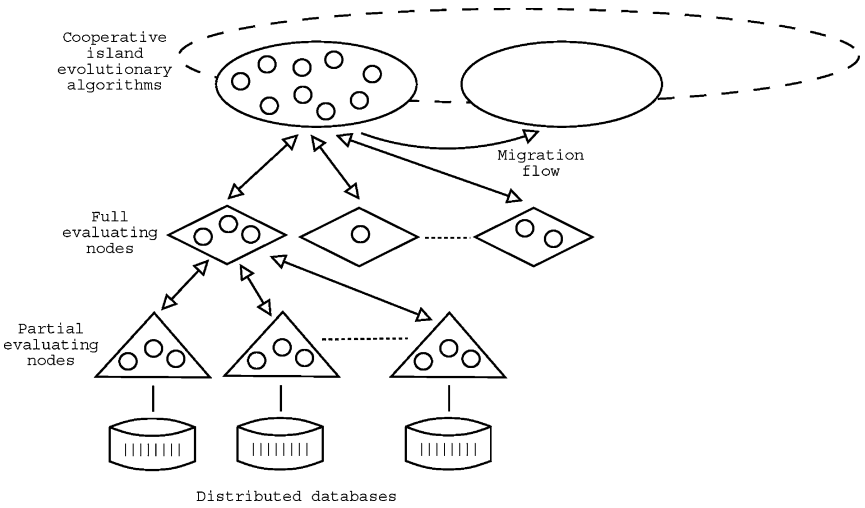

Figure 1. Three major parallel distributed models for EA.

convergence, especially when the EA are heterogeneous regarding the variation operators. The migration of individuals follows a policy defined by few parameters: the migration decision criterion, the exchange topology, the number of emigrants, the emigrants selection policy, and the replacement/integration policy.

– *Parallel evaluation of the population*. It is required as it is in general the most time-consuming. The parallel evaluation follows the centralized model. The farmer applies the following operations: selection, transformation and replacement as they require a global management of the population. At each generation, it distributes the set of new solutions between different workers. These evaluate and return back the solutions and their quality values. An efficient execution is often obtained particularly when the evaluation of each solution is costly. The two main advantages of an asynchronous model over the synchronous model are: (1) the fault tolerance of the asynchronous model; (2) the robustness in case the fitness computation can take very different computation times (e.g. for nonlinear numerical optimization). Whereas some time-out detection can be used to address the former issue, the latter one can be partially overcome if the grain is set to very small values, as individuals will be sent out for evaluations upon request of the workers.
– *Distributed evaluation of a single solution*. The quality of each solution is evaluated in a parallel centralized way. That model is particularly interesting when the evaluation function can be itself parallelized as it is CPU time-consuming and/or IO intensive. In that case, the function can be viewed as an aggregation of a certain number of partial functions. The partial functions could also be identical if for example the problem to deal with is a data mining one. The evaluation is thus data parallel and the accesses to data base are performed in parallel. Furthermore, a reduction operation is performed on the results returned by the partial functions. As a summary, for this model the user has to indicate a set of partial functions and an aggregation operator of these.

2.2. Parallel distributed local searches

In this section we first present the main existing local search methods and their working principles. Afterward, we describe the main existing parallel/distributed models of their design and implementation.

2.2.1. Local searches. All metaheuristics dedicated to the improvement of a single solution are based on the concept of neighborhood. They start from a solution randomly generated or obtained from another optimization algorithm, and update it, step by step, by replacing the current solution by one of its neighboring candidates. Some criterion have been identified to differentiate such searches: the heuristic internal memory, the choice of the initial solution, the candidate solutions generator, and the selection strategy of candidate moves. Three main algorithms of local search stand out: Hill Climbing (HC) (Papadimitriou, 1976), Simulated Annealing (SA) (Kirkpatrick, Gelatt, and Vecchi, 1983) and Tabu Search (TS) (Glover, 1989).

Hill Climbing: HC (Papadimitriou, 1976) is likely the oldest and simplest optimization method. At each step, the heuristic only replaces the current solution by the neighboring

one that improves the objective function. The search stops when all candidate neighbors are worse than the current solution, meaning a local optima is reached. Variants of HC may be distinguished according to the order in which the neighboring solutions are generated (deterministic/stochastic), and according to the selection strategy (choice of the best neighboring solution, the first best, etc.)

Simulated Annealing: SA (Kirkpatrick, Gelatt, and Vecchi, 1983) is sort of stochastic HC. In addition to the current solution, the best solution found since the beginning of the execution is stored. Moreover, few parameters control the progress of the search, which are the temperature and the iteration number. SA enables under some conditions the decrease of a solution. At each step, the neighborhood consists of only one candidate move randomly selected, that replaces the current solution if it is better. Otherwise, the new solution could be accepted with a given probability.

Tabu Search: TS (Glover, 1989) is similar to the steepest deterministic HC. Besides, it manages a memory of the solutions or moves recently applied, which is called the *tabu list*. When a local optima is reached, the search carries on by selecting a candidate worse than the current solution. To avoid the previous solution to be chosen again, and so to avoid cycles, TS discards the neighboring candidates that have been previously applied. Yet, if a candidate is proved to be very good, it could be accepted. This mechanism is called the *aspiration criterion*.

2.2.2. Parallel local searches. Two parallel distributed models are commonly used in the literature: the parallel distributed exploration of neighboring candidate solutions model, and the multi-start model. The models are illustrated by figure 2.

– *Parallel exploration of neighboring candidates*. It is a low-level Farmer-Worker model that does not alter the behavior of the heuristic. A sequential search computes the same results slower. At the beginning of each iteration, the farmer duplicates the current solution

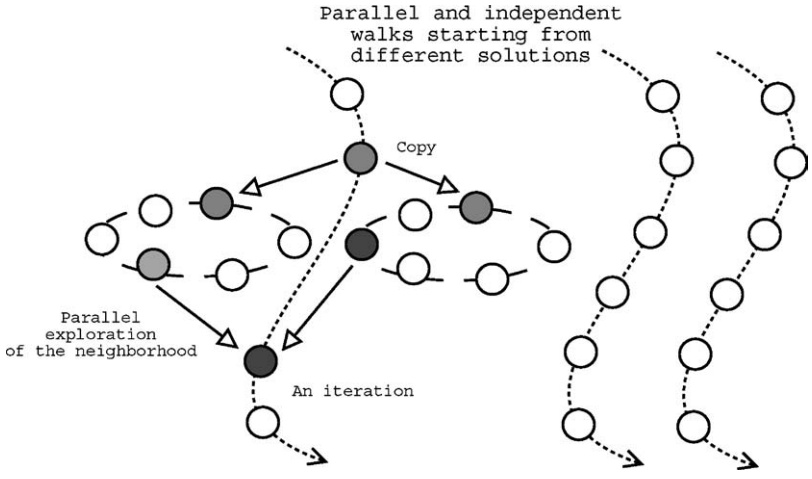

Figure 2. Two parallel models for local search.

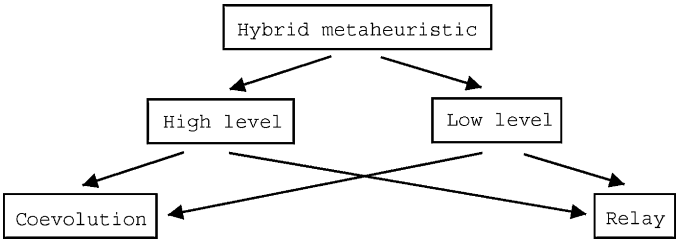

Figure 3. Hierarchical taxonomy of hybrid metaheuristics.

between distributed nodes. Each one manages some candidates and the results are returned to the farmer. The model is efficient if the evaluation of a each solution is time-consuming and/or there are a great deal of candidate neighbors to evaluate. This is obviously not applicable to SA since only one candidate is evaluated at each iteration. Likewise, the efficiency of the model for HC is not always guaranteed as the number of neighboring solutions to process before finding one that improves the current objective function may be highly variable.

– *Multi-start model.* It consists in simultaneously launching several local searches. They may be heterogeneous, but no information is exchanged between them. The results would be identical as if the algorithms were sequentially run. Very often deterministic algorithms differ by the supplied initial solution and/or some other parameters. This trivial model is convenient for low-speed networks of workstations.

2.3. Hybridization

Recently, hybrid metaheuristics have gained a considerable interest (Talbi, 2002). For many practical or academic optimization problems, the best found solutions are obtained by hybrid algorithms. Combinations of different metaheuristics have provided very powerful search methods. In Talbi (2002), E.-G. Talbi has distinguished two levels and two modes of hybridization (figure 3): Low and High levels, and Relay and Cooperative modes.

The low-level hybridization addresses the functional composition of a single optimization method. A function of a given metaheuristic is replaced by another metaheuristic. On the contrary, for high-level hybrid algorithms the different metaheuristics are self-containing, meaning no direct relationship to their internal working is considered. On the other hand, relay hybridization means a set of metaheuristics is applied in a pipeline way. The output of a metaheuristic (except the last) is the input of the following one (except the first). Conversely, co-evolutionist hybridization is a cooperative optimization model. Each metaheuristic performs a search in a solution space, and exchange solutions with others.

3. ParadisEO goals and architecture

The "EO" part of ParadisEO means *Evolving Objects*. EO is a C++ LGPL open source framework[1] and includes a paradigm-free Evolutionary Computation library (EOlib)

dedicated to the flexible design of EA through evolving objects superseding the most common dialects (Genetic Algorithms, Evolution Strategies, Evolutionary Programming and Genetic Programming). Furthermore, EO integrates several services including visualization facilities, on-line definition of parameters, application check-pointing, etc. ParadisEO is an extended version of the EO framework. The extensions include local search methods, hybridization mechanisms, parallelism and distribution mechanisms, and other features that are not addressed in this paper such as multi-objective optimization and grid computing. In the next sections, we present the motivations and goals of ParadisEO, its architecture and some of its main implementation details and issues.

3.1. Motivations and goals

A framework is normally intended to be exploited by as many users as possible. Therefore, its exploitation could be successful only if some important user criteria are satisfied. The following criteria are the major of them and constitute the main objectives of the ParadisEO framework:

- *Maximum design and code reuse*. The framework must provide for the user a whole architecture design of his/her solution method. Moreover, the programmer may redo as little code as possible. This objective requires a clear and maximal conceptual separation between the solution methods and the problems to be solved, and thus a deep domain analysis. The user might therefore develop only the minimal problem-specific code.
- *Flexibility and adaptability.* It must be possible for the user to easily add new features/metaheuristics or change existing ones without implicating other components. Furthermore, as in practice existing problems evolve and new others arise these have to be tackled by specializing/adapting the framework components.
- *Utility.* The framework must allow the user to cover a broad range of metaheuristics, problems, parallel distributed models, hybridization mechanisms, etc.
- *Transparent and easy access to performance and robustness.* As the optimization applications are often time-consuming the performance issue is crucial. Parallelism and distribution are two important ways to achieve high performance execution. In order to facilitate its use it is implemented so that the user can deploy his/her parallel algorithms in a transparent manner. Moreover, the execution of the algorithms must be robust to guarantee the reliability and the quality of the results. The hybridization mechanism allows to obtain robust and better solutions.
- *Portability.* In order to satisfy a large number of users the framework must support different material architectures and their associated operating systems.

3.2. ParadisEO architecture

The architecture of ParadisEO is multi-layer and modular allowing to achieve the objectives quoted above (see figure 4). This allows particularly a high flexibility and adaptability, an easier hybridization, and more code and design reuse. The architecture has three layers identifying three major categories of classes: *Solvers*, *Runners* and *Helpers.*

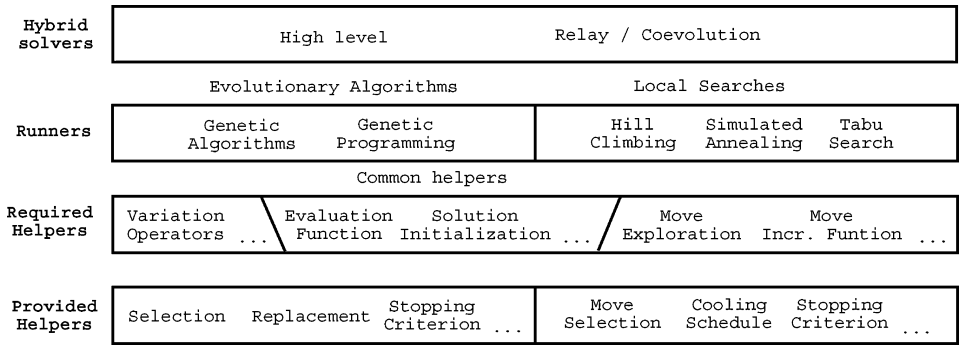

Figure 4. ParadisEO architecture.

– *Helpers.* Helpers are low-level classes that perform specific actions related to the evolution or search process. They are split in two categories: *Evolutionary helpers* (EH) and *Local search helpers* (LSH). EH include mainly the transformation, selection and replacement operations, the evaluation function and the stopping criterion. LSH can be generic such as the neighborhood explorer class, or specific to the local search metaheuristic like the tabu list manager class in the Tabu Search solution method. On the other hand, there are some special helpers dedicated to the management of parallel and distributed models 2 and 3, such as the communicators that embody the communication services.

  Helpers cooperate between them and interact with the components of the upper layer i.e. the runners. The runners invoke the helpers through function parameters. Indeed, helpers have not their own data, but they work on the internal data of the runners.
– *Runners.* The *Runners* layer contains a set of classes that implement the metaheuristics themselves. They perform the run of the metaheuristics from the initial state or population to the final one. One can distinguish the *Evolutionary runners* (ER) such as genetic algorithms, evolution strategies, etc., and *Local search runners* (LSR) like tabu search, simulated annealing and hill climbing. Runners invoke the helpers to perform specific actions on their data. For instance, an ER may ask the fitness function evaluation helper to evaluate its population. An LSR asks the movement helper to perform a given movement on the current state. Furthermore, runners can be serial or parallel distributed.
– *Solvers.* Solvers are devoted to control the evolution process and/or the search. They generate the initial state (solution or population) and define the strategy for combining and sequencing different metaheuristics. Two types of solvers can be distinguished. *Single metaheuristic solvers* (SMS) and *Multiple metaheuristics solvers* (MMS). SMSs are dedicated to the execution of only one metaheuristic. MMS are more complex as they control and sequence several metaheuristics that can be heterogeneous. They make use of different hybridization mechanisms presented in Section 2.3. Solvers interact with the user by getting the input data and delivering the output (best solution, statistics, etc).

According to the generality of their embedded features, the classes of the architecture split in two major categories: *Provided* classes and *Required* classes. Provided classes embody the factored out part of the metaheuristics. They are generic, implemented in the framework, and ensure the control at run time. Required classes are those that must be supplied by the user. They encapsulate the problem-specific aspects of the application. These classes are fixed but not implemented in ParadisEO. The programmer has the burden to develop them using the OO specialization mechanism.

4. ParadisEO implementation

At each layer of the ParadisEO architecture (figure 4) a set of classes is provided. Some of them are devoted to the development of serial metaheuristics, and others are devoted to manage transparently parallelism, distribution and hybridization. The ParadisEO framework implementation is LGPL open source.[2]

4.1. A brief overview of UML notation

In this section, we give a brief description of the UML language (Booch, Rumbaugh, and Jacobson, 1999) focusing only on the diagrams and notations that are used in this paper. UML stands for Unified Modeling Language. Today, it is accepted by the Object Management Group (OMG) as the standard for modeling object oriented programs. UML defines nine types of diagrams. In this paper, we focus only on class diagrams.

Class diagrams are the backbone of UML, they describe the static structure of our framework. We give an overview of the following diagrams: classes and active classes, composition and aggregation, generalization, templates and template parameters.

- *Classes and active classes.* Classes designate an abstraction of entities with common characteristics. They are illustrated (figure 5(a)) with rectangles divided into compartments. The partitions contain respectively the class name (centered, bold-ed and capitalized), the list of attributes, and the list of operations. In order to make the figures readable only the first partition is represented. Active classes (figure 5(b)) allow to initiate and control the flow of activity, while passive classes store data and serve other classes.
- *Composition and aggregation.* An aggregation relationship is an ownership between *Class A*, the whole (or aggregate), and *Class B*, its part. *Class A* plays a more important role than *Class B*, but the two classes are not dependent on each other. Aggregation is illustrated in figure 5(c) by a hollow diamond that points toward the whole class.

  Composition is a special type of aggregation that denotes a strong ownership between *Class A* and *Class B*. It is illustrated in figure 5(d) by a filled diamond that points toward the whole class.
- *Generalization.* Generalization is another name of inheritance or an "is a" relationship. This is represented by figure 5(e).

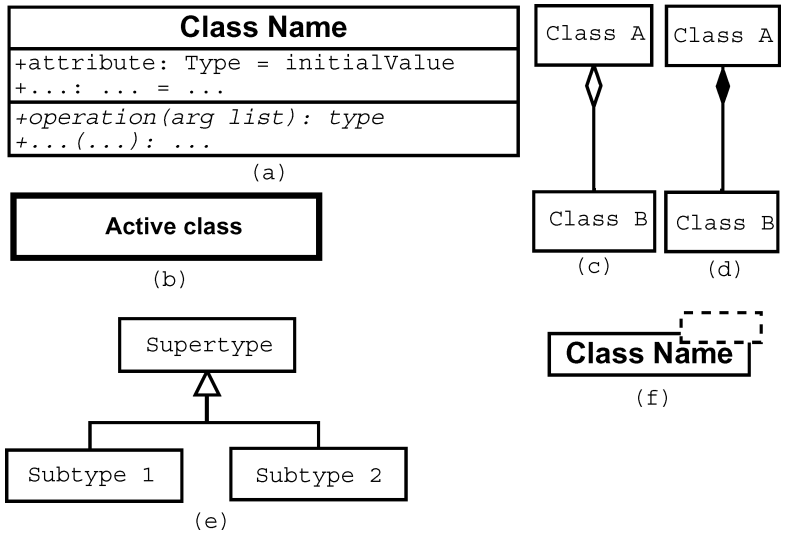

Figure 5. Basic UML class diagram symbols and notations.

– *Templates and template parameters.* Templates are generic classes that are represented by dashed rectangles. When they serve as parameters for a given class they are represented at the top-right corner of that class (figure 5(f)).

4.2. Implementation of serial metaheuristics

The classes for the serial metaheuristics are summarized in figure 6 using the UML notation. The solvers are not represented as there is only one solver that triggers the execution of a given runner.

A serial EA is mainly composed of a reproduction class (*eoBreed*), an evaluation function (*eoPopEvalFunction*), a replacement class (*eoReplacement*) and a continuator class (*eoContinue*). Note that all the classes use a template parameter *Evolving Object Type or EOT* that defines the type of the individuals of the EA population. It must be instantiated by the user.

A serial LS is composed of generic classes and other classes that are specific to each local search method. Generic classes include *eoMoveInit* that allows to initialize a given movement, *eoNextMove* that enables the exploration of the neighborhood, and *eoMoveIncrEval* that computes the fitness value of a solution if a given movement is applied. Specific classes allow to implement the specific features of LS quoted in Section 2.2.1. For instance, *eoTabuList* manages the Tabu list involved in the TS method. Note that all the classes use a template parameter *Move Type or MOVET* that defines the movement type, to be instantiated by the programmer. On the other hand, let us quote that the local search runner classes *eoHC*, *eoSA* and *eoTS* inherit from *eoMonOp*. Such design allows to make easier the low-level hybridization between EA and LS. For instance, the mutation operator of a given GA can be instantiated by a local search runner.

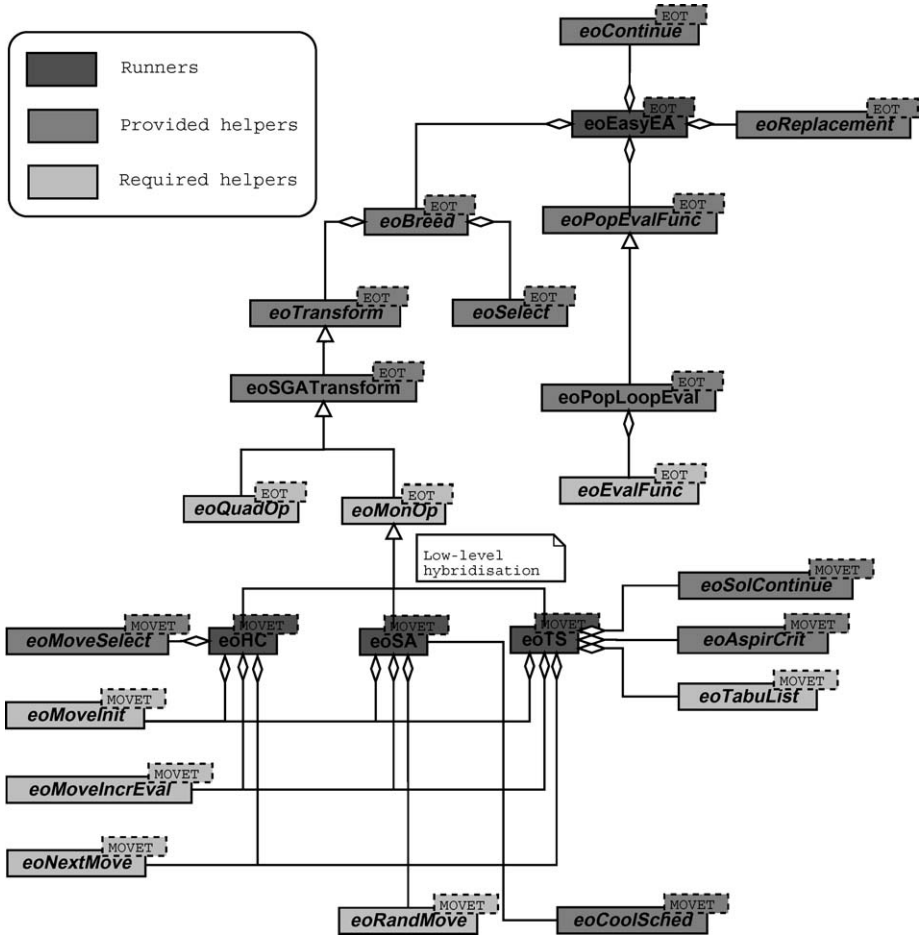

Figure 6. ParadisEO main classes.

4.3. Implementation of parallel and distributed evolutionary algorithms (PDEA)

Figure 7 illustrates the representation of the main classes composing a PDEA in ParadisEO. The area delimited by dashed lines contains the classes implementing the parallel distributed models.

The island model (model 1) component is a specialization of the continuator class. The models 2 and 3 components are obtained from *eoPopEvalFunction* through the inheritance mechanism. The user has just to instantiate these components to be deployed in a parallel/distributed environment. More implantation details on these models are given below.

– The migration decision maker represents the kernel of the island asynchronous cooperative model. It uses the user-defined parameters to take migration decisions, perform migrations according to the selection and replacement policies.

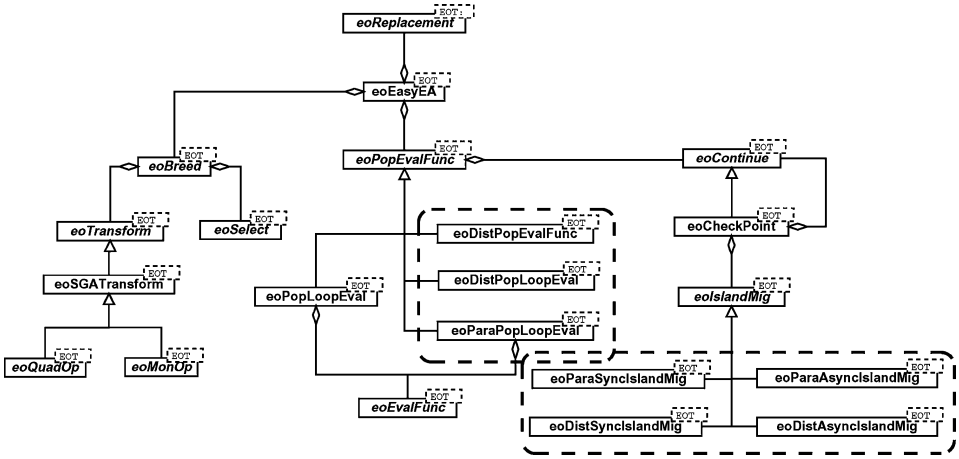

Figure 7. Main classes of a PDEA in ParadisEO.

The deployment of the algorithm requires to choose the technical means to be used to perform the migrations of the individuals according to the hardware support. On shared memory multiprocessors the implementation is multi-threaded and the migrations are memory copy-driven. Threads access in a concurrent way the populations of the different EA stored on the shared memory. On distributed memory machines the mechanism is communication driven. Migrations are performed by explicit message passing.

Figure 8 illustrates an UML-based representation of the core classes that compose both the parallel and distributed asynchronous migration components.

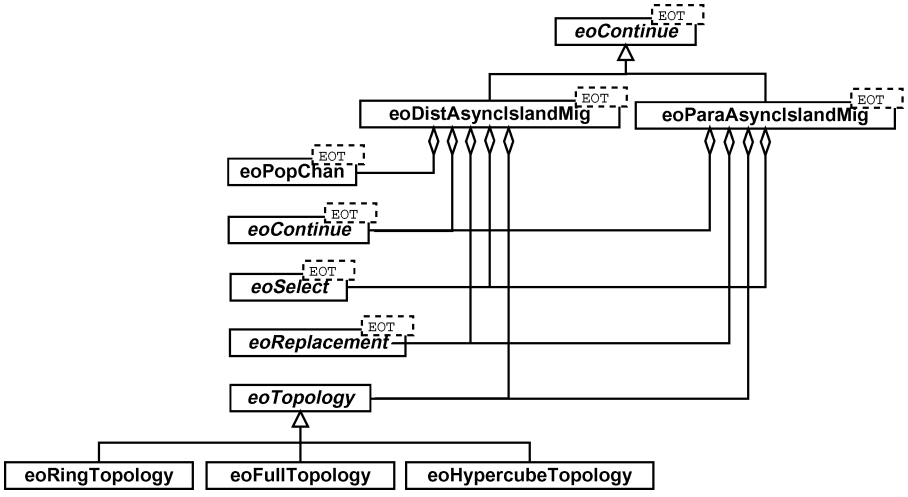

Figure 8. Core classes of the asynchronous migration manager.

Each migration criterion quoted above is represented by a class diagram. A migration component is a kind of a continuator (*eoContinue*) as migration operations are decided to be performed when the stopping criterion is evaluated. The parallel and distributed asynchronous migration components are illustrated respectively by *eoParaAsyncIslandMig* and *eoDistAsyncIslandMig*. They are both composed of a migration decision component (*eoContinue*), a selection component (*eoSelect*) and a replacement component (*eoReplacement*). One has to note that the components (*eoTopology*) and (*eoPopChan*) designate respectively the exchange topology and "buffers" for send/receive of individuals. They are not required for the parallel model because it is assumed that the EA cooperate through a shared recipient in a producer/consumer way.

Note that the specification of the synchronous migration component is quite similar. Instead of providing a continuator, the user just fixes the span between two migration operations. This span is simply expressed by a number of iterations, that should be the same for all EA. A migration consists in migrating some individuals to the neighboring populations, and then waiting an arrival of immigrants before resuming the evolution.

– In the parallel/distributed population evaluation model the population is always stored on a single processor. However, the assignment of the individuals to the processors for evaluation depends on the hardware architecture. On shared memory multi-processors threads pick up (copy) individuals from the global list in a concurrent way. On distributed memory processors the population is divided into sub-populations according to the user-defined granularity parameter. These sub-populations are then assigned to the processors using communication messages. When an evaluator returns its results to the farmer, that evaluator will immediately be sent another sub-population if there still are individuals to be evaluated. The performance of the model hence strongly depends on the grain size. Very fine-grained sub-populations induce a huge amount of communication, leading to a small acceleration over sequential programs. Conversely, very coarse-grained sub-populations imply a lower degree of parallelism, and thus again lead to a smaller acceleration. Some compromise has to be found between both extreme cases.

  As it is illustrated in figure 9, the instantiation of the model 2 is very simple. The parallel version (*eoParaPopLoopEval*) of the model needs only the evaluating operator class (*eoEvalFunc*), and the maximum number of concurrent processes that can be simultaneously launched. In the distributed version of the model, two classes are required: the farmer class (*eoDistPopLoopEval*) and the worker class (*eoDistPopLoopSolver*). At run

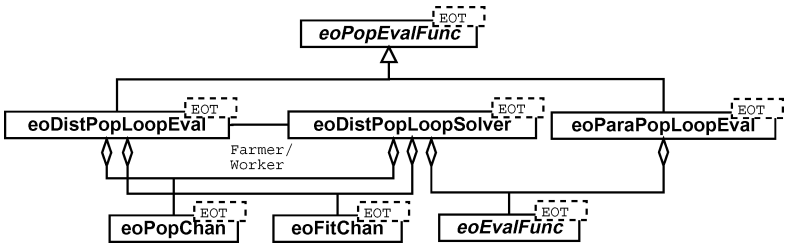

Figure 9. Core classes of model 2.

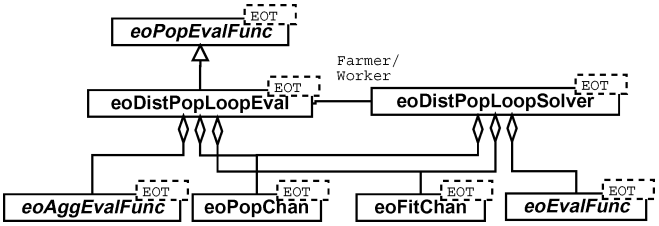

Figure 10. Core classes of model 3.

time, the farmer object distributes individuals among some solvers objects that request some work as soon as they are idle. Communication is performed through two channels: *eoPopChan* for sending/receiving individuals, and *eoFitChan* other to transmit computed fitnesses.

– The distributed evaluation of a single solution model (model 3) is well-suited for distributed memory machines. The data of the evaluation function have to be distributed among different processors. A distribution strategy has to be defined. One has to note here that the problem is application-dependent. Up to date, in ParadisEO it is supposed that the data are already distributed.

  The computation i.e. the user-specified partial functions are sent in explicit messages to the workers which apply them on their local data.

  The UML diagram sketched in figure 10 illustrates the core classes of model 3. It is quite similar to the model 2 illustrated in figure 9. The main difference is noticeable in the farmer class that includes an aggregation function component (*eoAggEvalFunc*) to aggregate partial fitnesses.

In order to meet the portability objective the implementation of the parallel/distributed model is based on different standards. Indeed, the Posix Threads multi-programming library is used to enable a portable execution on shared memory multi-processors. Moreover, the *de facto* communication libraries PVM and MPI allow a portable deployment on networks of heterogeneous workstations.

4.4. Implementation of parallel and distributed local searches (PDLS)

Figure 11 shows the main classes that compose together a PDLS. The area with a dashed line border puts forward classes that are related to parallelism or distribution.

Only the parallel exploration of neighboring candidates is considered in this diagram of classes. The model is detailed in figure 12. Indeed, the implementation of the multi-start model is trivial, and is similar to the island model without migration.

In the parallel/distributed model for exploration of neighboring candidates, the control of the local search and its application to a solution are managed on the farmer processor. A new component is introduced: the *eoMoveExpl* class. It is simply intended to enable the transformation of a given solution by the application of one movement. In its sequential form, it embeds the necessary operators associated to a specific type of movement (the

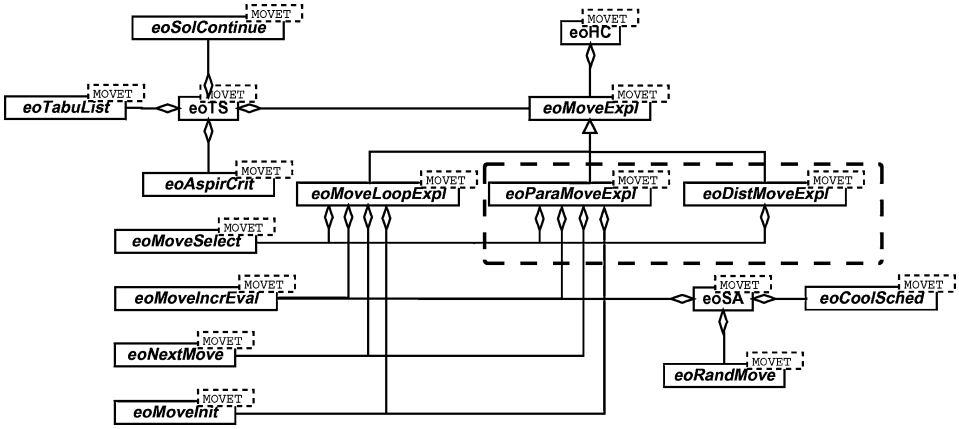

Figure 11. Main classes of a PDLS in ParadisEO.

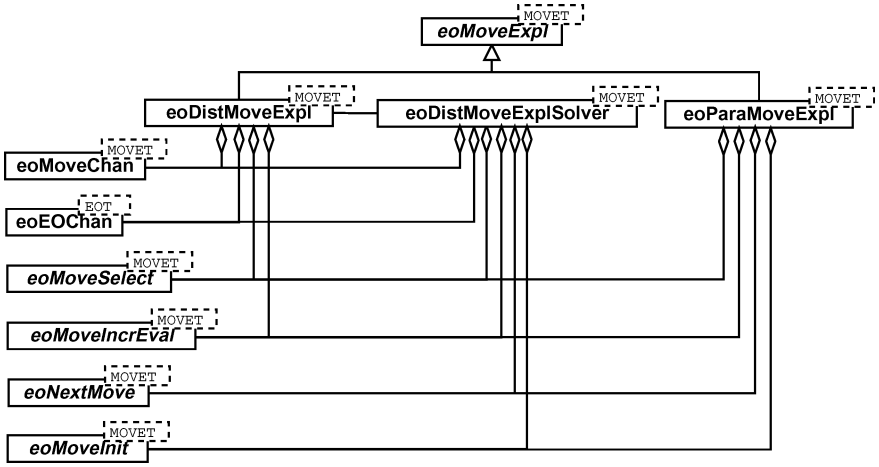

Figure 12. Core classes of the parallel/distributed exploration of neighboring candidates.

incremental evaluation function, the move initializer and updater, the selection strategy, etc).

The parallel version (*eoParaMoveExpl*) of the model is easy. It embeds the helpers allowing to (re)initialize a movement, to update a move to the next one and to compute the cost of its application. The number of processes that can be simultaneously run must be fixed. In addition, a neighborhood partitioning strategy should be defined in the *eoMoveInit* and *eoNextMove* classes.

The distributed model follows the Farmer/Worker paradigm. The implementation includes two classes: eoDistMoveExpl (the farmer) controls the synchronization, and gets results obtained from diverse partial explorations ; and, *eoDistMoveExplSolver* (the worker)

manages the selection of a move for a specific neighborhood. The farmer sends the current solution to partial explorers via the *eoEOChan* channel. Then, it waits for the return back of a selected candidate move via the *eoMoveChan* channel. A second selection is performed by the farmer to choose which of the returned moves should be retained and hence applied.

4.5. Transparent deployment

The deployment of the presented parallel/distributed models is transparent for the user. Indeed, the user does not need to manage the communications and threads-based concurrency. In ParadisEO, as illustrated in figure 13 the communication schema is composed of two layers: the *channels* layer and the *communicators* layer.

A communicator is a low-level interface used for data exchange between distributed applications. It is implemented as an abstract class called `eoComm` that embodies a set of sufficient communication services. This class is an abstraction of the two standard communication libraries PVM and MPI. The user can freely instantiate a communicator to use transparently either the PVM or MPI primitives.

A channel is a higher-level communication layer upon the communication interface. It facilitates the exchange of some data between distributed optimization components. According to the kind of exchanged data, different channels can be instantiated. For instance, the

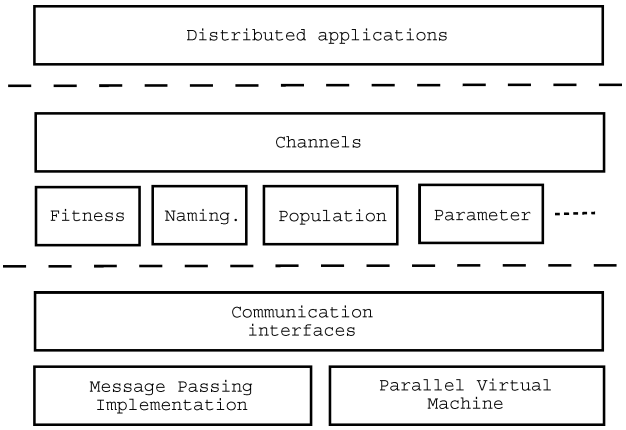

Figure 13. ParadisEO communication layers.

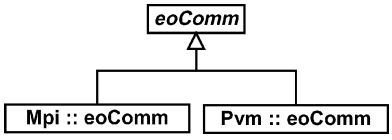

Figure 14. The communication interfaces.

channels *eoPopChan* and *eoFitChan* quoted above are devoted respectively to the exchange of populations of evolving objects, and to the sending back of fitness values. Channels embody some storage space and a set of services that make use of the communicator instantiated by the user, to perform communications.

From user exploitation point of view, the programmer has to instantiate channels and a communicator. The way the channels utilize the communicator is completely hidden for him/her.

5. Experimentation

ParadisEO has been experimented on different academic and industrial problems. In this section, we focus on a real-world application (radio network design). One of the major problems in the engineering of mobile telecommunication networks is the design of the network. It consists in positioning base stations on potential sites in order to fulfill some objectives and constraints (Meunier, Talbi, and Reininger, 2000). More exactly, the multi-objective problem is to find a set of sites for antennas from a set of pre-defined candidate locations, to determine the type and the number of antennas, and their configuration parameters (tilt, azimuth, power, ...). Exactly three objectives are considered: (1) Maximizing the communication traffic; (2) Minimizing the total number of sites; and, (3) Minimizing the communications interferences.

It is a hard practical problem with many decision variables. Tackling it in a reasonable time requires to use parallel heuristics. A parallel distributed multi-objective GA is investigated to solve the problem. The three parallel/distributed models have been exploited.

The islands in the model 1 evolve with heterogeneous variation operators. Migrations are performed according to a ring topology. Emigrants are randomly selected with a fixed percentage of 5% of the Pareto archive. The experimentation of the model on a cluster of 40 Pentium III PCs (733MHz CPU, 256Mo RAM). The island model has been deployed without individuals migration (version 1) and with individuals migration (version 2).

Four GAs are deployed during 10000 iterations with a migration occurring periodically every 200 generations. The quality of the obtained Pareto fronts are expressed by two special metrics for multi-objective optimization: contribution and entropy. The first computes the ratio of non-dominated solutions from one approximative front in comparison with another one. The entropy estimates the diversity of solutions in the criterion space.

The results presented in figures 15 and 16 show that the solutions of the Pareto front returned by the version 2 are much better and more diversified than those of the Pareto front obtained with the version 1. This proves the efficiency and the robustness of the implemented island model.

In model 2, an *eoDistPopLoopEval* farmer object and a set of *eoDistPopLoopSolver* worker objects are deployed on the cluster described above. Each PC is assigned only one worker. The grain size is fixed to one individual at a time. This means that each worker runs a cycle: it asks for one individual i.e. a radio network to be evaluated, it performs its evaluation, and returns back the result to the farmer. The full curve in figure 17 designates the obtained speed-up. The model is efficient and scales in a near-linear way. This can be

Migration	Contribution	Entropy
Activated	0.85	0.84
Deactivated	0.15	0.49

Figure 15. Numerical results for cooperative island E.As.

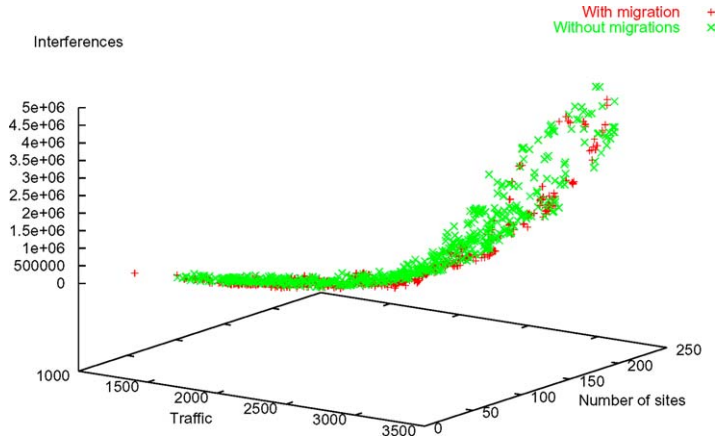

Figure 16. Two approximative Pareto fronts obtained by cooperative island E.As, according to the (des)activation of migrations.

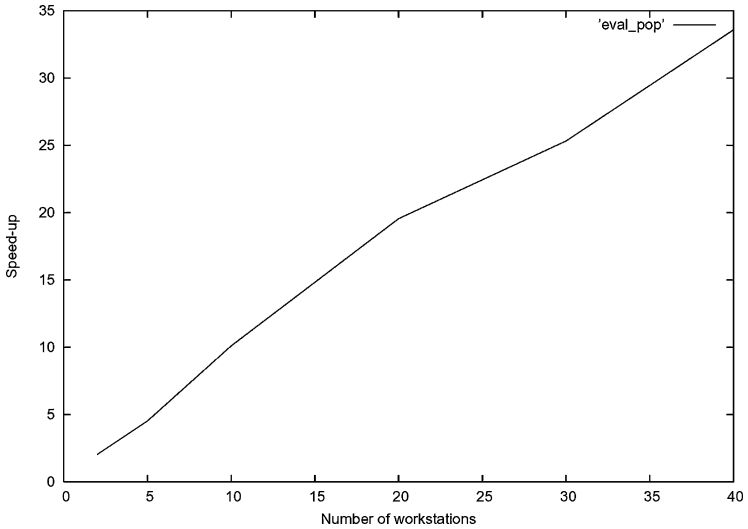

Figure 17. Speed-up measurements for model 2.

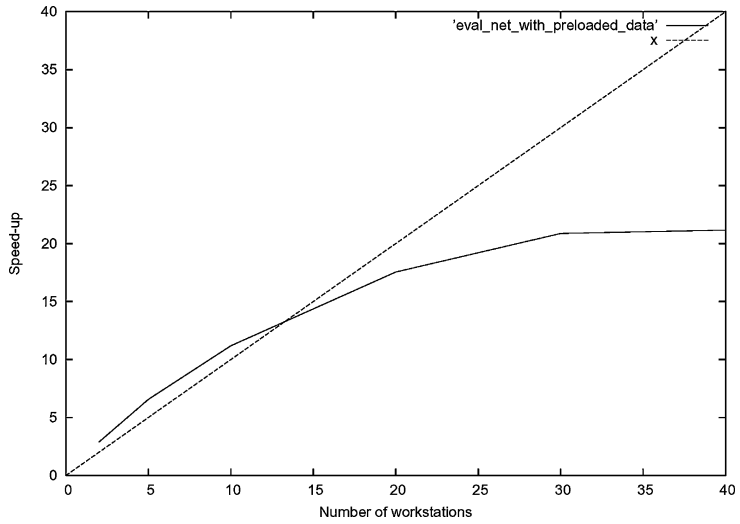

Figure 18. Speed-up measurements for model 3.

explained by the relatively high ratio between the cost of objective function evaluation and the individuals communication cost.

Furthermore, the efficiency of the fitness evaluation of a radio network can be greatly improved if network data have been previously loaded. Nevertheless, the amount of data is very large (i.e. several hundred of megabytes) and can not likely be totally loaded on a single workstation, unless the capacity of its memory is sufficient enough. Yet, it is possible to deploy model 3 by splitting these computational data between some partial evaluating nodes, each one being assigned a subpart of the geographical area (data). Figure 18 shows the speed-up obtained with the experimentation of model 3 on the previous cluster. The model scales super-linearly until 10 workstations, and beyond it follows a logarithmic behavior. The reason is that the workload (partial function computed on a smaller part of the radio network) assigned to each machine decreases when the total number of machines increases. Therefore, model 3 could be deployed efficiently only if the number of workstations contributing simultaneously to the individual evaluation if less than 10 workstations.

The conclusion is that model 2 is recommended in any case as it is efficient and scales in a near-linear way. Model 3 is application dependent, and requires a precaution from the user. It could be used only if the objective function evaluation is costly. In this case, the number (threshold) of workstations dedicated to each evaluation must be fixed by the user. For instance, as it is quoted above this threshold is 10 for the design network application. In the future version of ParadisEO, this parameter will be determined automatically by the framework.

6. Related work

Three major approaches allow the development of PDM: *from scratch or no reuse*, *only code reuse* and *both design and code reuse*. Reusability may be defined as the ability of software

components to build many different applications (Fink, Vo, and Woodruff, 1999). The basic idea behind the **from scratch**-oriented approach is the apparent simplicity of meta-heuristics code. Programmers are tempted to develop themselves their code. Therefore, they are face with several problems: the development requires time and energy, it is error-prone and difficult to maintain, etc. **Only code reuse** consists of reusing third-party code available on the Web either as free individual programs, or as libraries. Indeed, an old third-party code has usually application-dependent sections that must be extracted before the new application-dependent code can be inserted. Changing these sections is often time-consuming and error-prone. The code reuse through libraries (Wall, 1999) is obviously better because these are often well tried, tested and documented, thus more reliable. Nowadays, it is recognized that the OO paradigm is well-suited to develop reusable libraries. However, libraries allow to code reuse but they do not permit the reuse of complete invariant part of algorithms. Therefore, the coding effort using libraries remains important.

The objective of the **both code and design reuse** approach is to overcome this problem i.e. to redo as little code as possible each time a new optimization problem is dealt with. The basic idea is to capture into special components the recurring (or invariant) part of solution methods to standard problems belonging to a specific domain. Unlike libraries frameworks are characterized by the inverse control mechanism for the interaction with the application code. In a framework, the provided code call the user-defined one according to the Hollywood property: "do not call us, we call you". Therefore, frameworks provide the full control structure of the invariant part of the algorithms, and the user has only to supply the problem-specific details. In order to meet this property the design of a framework must be based on a clear conceptual separation between the solution methods and the problems they tackle.

This separation requires a solid understanding of the application domain. The domain analysis results in a model of the domain to be covered by reusable classes with some constant and variable aspects. The constant part is encapsulated in generic/abstract classes or skeletons (Alba and the MALLBA Group, 2002) that are implemented in the framework. The variable part is problem-specific, it is fixed in the framework but implemented by the user. This part is a set of holes or hot spots (Pree et al., 1995) that serve to fill the skeletons provided by the framework when building specific applications. In Roberts and Johnson (2003), it is recommended to use OO composition rather than inheritance to perform this separation. The reason is that classes are easier to reuse than individual methods. Another and completely different way this separation is performed is adopted in Bleuler et al. (2003). Indeed, the framework provides a ready-to-use module for each part, and the two modules communicate through text files. This allows less flexibility than the object-oriented approach. Moreover, it induces an additional overhead, even if this latter is small. Nevertheless, the Pisa approach is multi-language allowing more code reuse.

According to the openness criterion, two types of frameworks can be distinguished: white or glass-box frameworks and black-box frameworks. In black-box frameworks one can reuse components by plugging them together through static parameterization and composition, and not worrying about how they accomplish their individual tasks (Johnson and Foote, 1988). In contrast, white-box frameworks require an understanding of how the classes work so that correct subclasses (inheritance-based) can be developed. Therefore, they allow

more extendibility. Frameworks often start as white-box frameworks, these are primarily customized and reused through classes specialization. When the variable part has stabilized or been realized, it is often appropriate to evolve to black-box frameworks (Fink, Vo, and Woodruff, 1999).

Several white-box frameworks for the reusable design of PDM have been proposed and are available on the Web. Some of them are restricted to only parallel and distributed EA (PDEA). The most important of them are the following: DREAM[3] (Arenas et al., 2002), ECJ[4] (Luke et al., 2002), JDEAL[5] (Costa, Lopes, and Silva, 1999) and Distributed BEAGLE[6] (Gagné, Parizeau, and Dubreuil, 2003). These frameworks are reusable as they are based on a clear object-oriented conceptual separation. They are also portable as they are developed in Java except the last system, which is programmed in C++. However, they are limited regarding the parallel distributed models. Indeed, in DREAM and ECJ only the island model is implemented using Java threads and TCP/IP sockets. DREAM is particularly deployable on peer to peer platforms. Furthermore, JDEAL and Distributed BEAGLE provide only the Master-Slave model (model 2) using TCP/IP sockets. The latter implements also the synchronous migration-based island model, but it is deployable on only one processor.

In the local search domain, most of existing frameworks (Di Gaspero and Schaerf, 2001; Michel and Van Hentenryck, 2001) do not allow parallel distributed implementations. Those enabling parallelism/distribution are often dedicated to only one solution method. For instance, (Blesa, Hernandez, and Xhafa, 2001) provides parallel skeletons for the TS method. Two skeletons are provided and implemented in C++/MPI: independent runs (multi-start) model with search strategies, and a Master-Slave model with neighborhood partition. The two models can be exploited by the user in a transparent way.

Few frameworks available on the Web are devoted to both PDEA and PDLS, and their hybridization. MALLBA[7] (Alba and the MALLBA Group, 2002), MAFRA[8] (Krasnogor and Smith, 2000) and ParadisEO are good examples of such frameworks. MAFRA is developed in Java using design patterns (Gamma et al., 1994). It is strongly hybridization-oriented, but it is very limited regarding parallelism and distribution and ParadisEO have numerous similarities. They are C++/MPI open source frameworks. They provide all the previously presented distributed models, and different hybridization mechanisms. However, they are quite different as we believe that ParadisEO is more flexible because the granularity of its classes is finer. Moreover, ParadisEO provides also the PVM-based communication layer and Pthreads-based multi-threading. On the other hand, MALLBA is deployable on wide area networks (Alba and the MALLBA Group, 2002). Communications are based on *NetStream*, an *ad-hoc* flexible and OOP message passing service upon MPI. Furthermore, MALLBA allows the hybridization of metaheuristics with exact methods.

7. Conclusion and perspectives

In this paper, we have presented the ParadisEO framework dedicated to the reusable design of PDM. ParadisEO provides a broad range of features including EA, LS, parallel and distributed models, different hybridization mechanisms, etc. This high content and

utility increases its usefulness. Nowadays, many European users of ParadisEO are already identified.

ParadisEO is a white-box OO framework based on a clear conceptual separation of the metaheuristics from the problems they are intended to solve. This separation and the large variety of implemented optimization features allow a maximum code and design reuse. The separation is expressed at implementation level by splitting the classes in two categories: provided classes and required classes. The provided classes constitute a hierarchy of classes implementing the invariant part of the code. Expert users can extend the framework by inheritance/specialization. The required classes coding the problem-specific part are abstract classes that have to be specialized and implemented by the user.

The classes of the framework are fine-grained, and instantiated as evolving objects embodying each one only one method. This is a particular design choice adopted in ParadisEO. The heavy use of these small-size classes allows more independence and thus a higher flexibility compared to other frameworks. Changing existing components and adding new ones can be easily done without impacting the rest of the application.

ParadisEO is one of the rare frameworks that provide the most common parallel and distributed models. These models concern the island-based running of metaheuristics, the evaluation of a population, and the evaluation of a single solution. All these are provided in ParadisEO. They are portable on distributed-memory machines and shared-memory multi-processors as they are implemented using standard libraries such as MPI, PVM and PThreads. The models can be exploited in a transparent way, one has just to instantiate their associated ParadisEO components. The user has the possibility to choose by a simple instantiation MPI or PVM for the communication layer. The models have been validated on academic and industrial problems. The experimental results demonstrate their efficiency. The experimentation demonstrate also the high reuse capabilities as the results show that the user redo little code. Furthermore, the framework provides the most common hybridization mechanisms. They can be exploited in a natural way to make cooperating metaheuristics belonging either the same family or to different families.

In the future, the focus will be on three major extensions: (1) allowing, as in MALLBA the design of exact methods, and their hybridization with the metaheuristics ; (2) providing all the required features for multi-objective optimization ; and (3) extending the parallel distributed models to allow their deployment on grid and peer to peer platforms. This is a great challenge as nowadays there is no effective grid-enabled framework for metaheuristics to the best of our knowledge.

Notes

1. Downloadable at http://eodev.sourceforge.net.
2. Downloadable at http://www.lifl.fr/∼cahon/paradisEO/.
3. Distributed Resource Evolutionary Algorithm Machine: http://www.world-wide-dream.org.
4. Java Evolutionary Computation: http://www.cs.umd.edu/projects/plus/ec/ecj/.
5. Java Distributed Evolutionary Algorithms Library: http://laseeb.isr.ist.utl.pt/sw/jdeal/.
6. Distributed Beagle Engine Advanced Genetic Learning Environment: http://www.gel.ulaval.ca/~ beagle.
7. MAlaga+La Laguna+BArcelona: http://neo.lcc.uma.es/mallba/mallba.html.
8. Java Mimetic Algorithms Framework: http://www.cs.nott.ac.uk/∼nxk/MAFRA/MAFRA.html.

References

E. Alba and the MALLBA Group. (2002). "MALLBA: A Library of Skeletons for Combinatorial Optimization." In R.F.B. Monien (ed.), *Proceedings of the Euro-Par*, vol. 2400 of LNCS. Paderborn: Springer-Verlag, pp. 927–932.

Arenas, M.G., P. Collet, A.E. Eiben, M. Jelasity, J.J. Merelo, B. Paechter, M. Preußand, and M. Schoenauer. (2002). A Framework for Distributed Evolutionary Algorithms. In *Proceedings of PPSN VII*.

Blesa, M.J., Ll. Hernandez, and F. Xhafa. (2001). "Parallel Skeletons for Tabu Search Method." In *8th International Conference on Parallel and Distributed Systems*. Kyongju City, Korea: IEEE Computer Society Press, pp. 23–28.

Bleuler, S., M. Laumanns, L. Thiele, and E. Zitzler. (2003). PISA—A Platform and Programming Language Independent Interface for Search Algorithms." In *Evolutionary Multi-Criterion Optimization (EMO'2003)*. Springer Verlag, pp. 494–508.

Booch, G., J. Rumbaugh, and I. Jacobson. (1999). *The Unified Modeling Language User Guide*. Addison Wesley Professional.

Cahon, S., E.-G. Talbi, and N. Melab. (2003). "ParadisEO: A Framework for Parallel and Distributed Biologically Inspired Heuristics." In *IEEE NIDISC'03 Nature Inspired Distributed Computing Workshop, (in conjunction with IEEE IPDPS2002)*. Nice, France.

Costa, J., N. Lopes, and P. Silva. (1999). "JDEAL: The Java Distributed Evolutionary Algorithms Library." http://laseeb.isr.ist.utl.pt/sw/jdeal/home.html.

Fink, A., S. Vo, and D. Woodruff. (1999). "Building Reusable Software Components for Heuristc Search." In P. Kall and H.-J. Luthi (eds.), *Operations Research Proc., 1998*. Berlin: Springer, pp. 210–219.

Gagné, C., M. Parizeau, and M. Dubreuil. (2003). "Distributed BEAGLE: An Environment for Parallel and Distributed Evolutionary Computations." In *Proc. of the 17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS) 2003*.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. (1994). *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Di Gaspero, L. and A. Schaerf. (2001). "Easylocal++: An Object-Oriented Framework for the Design of Local Search Algorithms and Metaheuristics." In *MIC'2001 4th Metaheuristics International Conference*, Porto, Portugal, pp. 287–292.

Glover, F. (1989). "Tabu Search, Part I." *ORSA, Journal of Computing* 1, 190–206.

Goodman, E. (1994). "An Introduction to GALOPPS—The Genetic Algorithm Optimized for Portability and Parallelism System." Technical report, Intelligent Systems Laboratory and Case Center for Computer-Aided Engineering and Manufacturing, Michigan State University.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI, USA: The University of Michigan Press.

Johnson, R. and B. Foote. (1988). "Designing Reusable Classes." *Journal of Object-Oriented Programming* 1(2), 22–35.

Keijzer, M., J.J. Morelo, G. Romero, and M. Schoenauer. (2001). "Evolving Objects: A General Purpose Evolutionary Computation Library." In *Proc. of the 5th Intl. Conf. on Artificial Evolution (EA'01), Le Creusot, France*.

Kirkpatrick, S., C.D. Gelatt, and M.P. Vecchi. (1983). "Optimization by Simulated Annealing." *Science* 220(4598), 671–680.

Krasnogor, N. and J. Smith. (2000). "MAFRA: A Java Memetic Algorithms Framework." In Alex A. Freitas, William Hart, Natalio Krasnogor, and Jim Smith (eds.), *Data Mining with Evolutionary Algorithms*, Las Vegas, Nevada, USA, vol. 8, pp. 125–131.

Luke, S., L. Panait, J. Bassett, R. Hubley, C. Balan, and A. Chircop. (2002). ECJ: A Java-based Evolutionary Computation and Genetic Programming Research System. http://www.cs.umd.edu/projects/plus/ec/ecj/.

Meunier, H., E.-G. Talbi, and P. Reininger. (2000). "A Multiobjective Genetic Algorithm for Radio Network Optimization." In *Congress on Evolutionary Computation*. vol. 1. IEEE Service Center, pp. 317–324.

Michel, L. and P. Van Hentenryck. (2001). "Localizer++: An Open Library for Local Search." Technical Report CS-01-02, Brown University, Computer Science.

Papadimitriou, C.H. (1976). "The Complexity of Combinatorial Optimization Problems." Master's thesis, Princeton University.

Pree, W., G. Pomberger, A. Schappert, and P. Sommerlad. (1995). "Active Guidance of Framework Development." *Software—Concepts and Tools* 16(3), 136.

Roberts, D. and R. Johnson. "Evolving Frameworks. A Pattern Language for Developping Object-Oriented Frameworks." To be published in Pattern Languages of Program Design 3 (PLoPD3).

Talbi, E.-G. (2002). "A Taxonomy of Hybrid Metaheuristics." *Journal of Heuristics, Kluwer Academic Publishers*, 8, 541–564.

Wall, M. "GAlib: A C++ Library of Genetic Algorithm Components." http://lancet.mit.edu/ga/.