PARALLEL MULTI-OBJECTIVE EVOLUTIONARY ALGORITHMS

El-Ghazali Talbi
University of Lille 1, Villeneuve d'Ascq, France
el-ghazali.talbi@univ-lille1.fr

Abstract This paper describes a general overview of parallel multi-objective evolutionary algorithms (MOEA) from the design and the implementation point of views. A unified taxonomy using three hierarchical parallel models is proposed. Different parallel architectures are considered. The performance evaluation issue of parallel MOEA is also discussed.

Keywords: Multi-objective optimization, Parallel evolutionary algorithms.

1. Motivation

On one hand, multi-objective optimization problems (MOPs), such as in engineering design and life science, are more and more complex and their resource requirements to solve them are ever increasing. Real-life MOPs are often NP-hard, and CPU time and/or memory consuming. Although the use of multi-objective evolutionary algorithms (MOEAs) allows to significantly reduce the computational complexity of the solving algorithms, the latter remains time-consuming for many MOPs in diverse domains of application, where the objective function and the constraints associated to the problem are resource (e.g., CPU, memory) intensive and the size of the search space is huge. Moreover, more and more complex and resource intensive MOEAs are developed to obtain a good approximation of the Pareto front in a reasonable time.

On the other hand, the rapid development of technology in designing processors (e.g., multi-core processors, dedicated architectures), networks (local networks (LAN) such as Myrinet and Infiniband or wide area networks (WAN) such as optical networks), and data storage make the use of parallel computing more and more popular. Such architectures represent an effective opportunity for the design and implementation of parallel multi-objective optimization algorithms. Indeed, sequential architectures are reaching physical limitations (speed of light, thermodynamics). Nowadays, even laptops and workstations are equipped with multi-core processors, which represent one class of parallel architecture. Moreover, the ratio cost/performance is constantly decreasing. The proliferation of powerful processors and fast communication networks have shown the emergence of dedicated architectures (e.g GPUs), clusters of processors (COWs), networks of workstations (NOWs), large-scale networks of machines (Grids) and Clouds as platforms for high performance computing.

Parallel computing can be used in the design and implementation of MOEAs for the following reasons:

- **Speedup the search to approximate the Pareto front**: The goal here is to reduce the search time. This helps designing interactive optimization methods which is an important issue for multi-criteria decision making. This is a also an important aspect for some class of problems where there are hard requirements on search time such as in dynamic MOPs and time-critical operational MOPs such as "real-time" planning and control.

- **Improve the quality of the obtained Pareto solutions**: some parallel models for MOEAs allow to improve the quality of Pareto solutions. Indeed, exchanging information between algorithms will alter their behavior in terms of searching in the landscape associated to the MOP. The main goal in a cooperation between algorithms is to improve the quality of Pareto solutions. Both convergence to better Pareto solutions and reduced search time may happen. Let us notice that a parallel model for MOEAs may be more effective than a sequential algorithm even on a single processor.

- **Improve the robustness**: a parallel MOEA may be more robust in terms of solving in an effective manner different MOPs and different instances of a given problem. Robustness may be measured in terms of the sensitivity of the algorithm to its parameters and the target MOPs.

- **Solve large scale MOPs**: parallel MOEAs allow to solve large scale instances of complex MOPs. A challenge here is to solve very large instances that cannot be solved on a sequential machine. Another similar challenge is to solve more accurate mathematical models associated to different MOPs. Improving the accuracy of mathematical models increases in general the size of the associated problems to be solved. Moreover, some optimization problems need the manipulation of huge databases such as data mining problems.

In this paper, a clear difference is made between the parallel design aspect and the parallel implementation aspect of MOEAs. A unifying view of parallel models for MOEAs is presented. The implementation point of view deals with the efficiency of parallel MOEAs on a target parallel architecture using a given parallel language, programming environment or middleware. Different architectural criteria, which affect the efficiency of the implementation, will be considered: shared memory versus distributed memory, homogeneous versus heterogeneous, shared versus non shared by multiple users, local network versus large network. Indeed, those criteria have a strong impact on the deployment technique employed such as load balancing and fault-tolerance. Depending on the type of parallel architecture used, different parallel and distributed languages, programming environments and middlewares may be used such as message passing (e.g., MPI), shared memory (e.g., multi-threading, OpenMP, CUDA), remote procedural call (e.g., Java RMI, RPC), high-throughput computing (e.g., Condor), and grid computing (e.g., Globus).

This paper is organized as follows. In Section 2, the main parallel models for designing MOEAs are presented. Section 3 deals with the implementation issues of parallel MOEAs. In this section, the main concepts of parallel architectures and parallel programming paradigms, which interfere with the design and implementation of parallel MOEAs are outlined. The main performance indicators that can be used to evaluate a parallel multi-objective search algorithms in terms of efficiency are detailed.

2. Parallel Design of Multi-Objective Metaheuristics

In terms of designing parallel MOEAs, three major parallel models are identified. They follow the three hierarchical levels (Table 1):

- **Algorithmic-level:** in this model, independent or cooperating self-contained MOEAs are used. It is a problem-independent inter-algorithm parallelization. If the different MOEAs are independent, the search will be equivalent to the sequential execution of the algorithms in terms of the quality of Pareto solutions. However, the cooperative model will alter the behavior of the MOEAs and enable the improvement in terms of the quality of Pareto solutions.

- **Iteration-level:** in this model, each iteration of a MOEA is parallelized. It is a problem-independent intra-algorithm parallelization. The behavior of the MOEA is not altered. The main objective is to speedup the algorithm by reducing the search time. Indeed, the iteration cycle of MOEAs on large populations, especially for real-world MOPs, requires a large amount of computational resources.

- **Solution-level:** in this model, the parallelization process handles a single solution of the search space. It is a problem-dependent intra-algorithm parallelization. In general, evaluating the objective functions or constraints for a generated solution is frequently the most costly operation in MOEAs. In this model, the behavior of the search algorithm is not altered. The objective is mainly the speedup of the search.

Table 1: Parallel models of MOEAs.

Parallel model	Problem dependency	Behavior	Granularity	Goal
Algorithmic-level	Independent	Altered	MOP algorithm	Effectiveness
Iteration-level	Independent	Non altered	Iteration	Efficiency
Solution-level	Dependent	Non altered	Solution	Efficiency

2.1 Algorithmic-Level Parallel Model

In this model, many MOEAs are launched in parallel. They may cooperate or not to solve the target MOPs.

2.1.1 Independent algorithmic-level parallel model. In the independent-level parallel model, the different MOEAs are executed without any cooperation. The different MOEAs may be initialized with different populations. Different parameter settings may be used for the MOEAs such as the mutation and crossover probabilities. Moreover, each search component of an MOEA may be designed differently: encoding, search operators (e.g., variation operators), objective functions, constraints, fitness assignment, diversity preserving, elitism. This parallel model is straightforward to design and implement. The master/worker paradigm is well suited to this model. A worker implements an MOEA. The master defines the different parameters to use by the workers and determines the best found Pareto solutions from those obtained by the different workers. In addition to speeding up the MOEA, this parallel model enables to improve its robustness [29].

This model raises particularly the following question: is it equivalent to execute k MOEAs during a time t and to execute a single MOEA during $k * t$? The answer depends on the landscape properties of the problem (e.g., distribution of the Pareto local optima).

2.1.2 Cooperative algorithmic-level parallel model. In the cooperative model for parallel MOEAs, the different MOEAs are exchanging information related to the search with the intent to compute a better and more robust Pareto front [30]. In general, an archive is maintained in parallel to the current population. This archive contains all Pareto optimal solutions generated during the search.

In designing this parallel cooperative model for any MOEA, the same design questions need to be answered:

- **The exchange decision criterion (When?)**: the exchange of information between the MOEAs can be decided either in a *blind* (periodic or probabilistic) way or according to an *"intelligent"* adaptive criterion. Periodic exchange occurs in each algorithm after a fixed number of iterations; this type of communication is synchronous. Probabilistic exchange consists in performing a communication operation after each iteration with a given probability. Conversely, adaptive exchanges are guided by some characteristics of the multi-objective search. For instance, it may depend on the evolution of the quality of the Pareto front. A classical criterion is related to the update of the archive, in which a new Pareto solution is generated.

- **The exchange topology (Where?)**: the communication exchange topology indicates for each MOEA its neighbor(s) regarding the exchange of information, i.e., the source/destination algorithm(s) of the information. The ring, mesh and hypercube regular topologies are the most popular ones.

- **The information exchanged (What?)**: this parameter specifies the information to be exchanged between the MOEAs. In general, the information exchanged is composed of:

  - Pareto solutions: this information deals with any selection strategy of the generated Pareto solutions during the search. In general, it contains solutions from the current population and/or the archive. The number of selected Pareto optimal solutions may be an absolute value or a percentage of the sets.

  - Search memory: this information deals with a search memory of a MOEA excluding the Pareto optimal solutions. This information deals with any element of the search memory that is associated to the involved MOEA.

- **The integration policy (How?)**: analogously to the information exchange policy, the integration policy deals with the usage of the received information. In general, there is a local copy of the received information. The local copies of the information received are generally updated using the received ones. The Pareto solutions received will serve to update the local Pareto archive. For the current population, any replacement strategy can be used (e.g., random, elitist). For instance, the best Pareto set is simply updated by the best between the local best Pareto set and the neighboring best Pareto set. Any replacement strategy may be applied on the local population by the set of received solutions.

Few of such parallel search models have been especially designed for multi-objective optimization [29].

The other well known parallel model for MOEAs, the cellular model. may be seen as a special case of the island model where an island is composed of a single individual. Traditionally, an individual is assigned to a cell of a grid. The selection occurs in the neighborhood of the individual. Hence, the selection pressure is less important than in sequential MOEAs. The overlapped small neighborhood in cellular MOEAs helps exploring the search space because a slow diffusion of Pareto solutions through the population provides a kind of exploration, while exploitation takes place inside each neighborhood. Cellular models applied to complex problems can have a higher convergence probability to better solutions than panmictic MOEAs [17].

The different MOEAs involved in the cooperation may evaluate different subsets of objective functions (Fig. 1). For instance, each MOEA may handle a single objective. Another approach consists in using a different aggregation weights in each MOEA, or different constraints [22].

Each MOEA may also represent a different partition of the decision space or the objective space [15, 27]. By this way, each MOEA is destined to find a particular portion of the Pareto-optimal front.

Another main issue in the development of parallel MOPs is how the Pareto set is built during the optimization process. Two different approaches may be considered (Fig. 1):

- *Centralized Pareto Front*: the front is a centralized data structure of the algorithm that it is built by the MOEAs during the whole

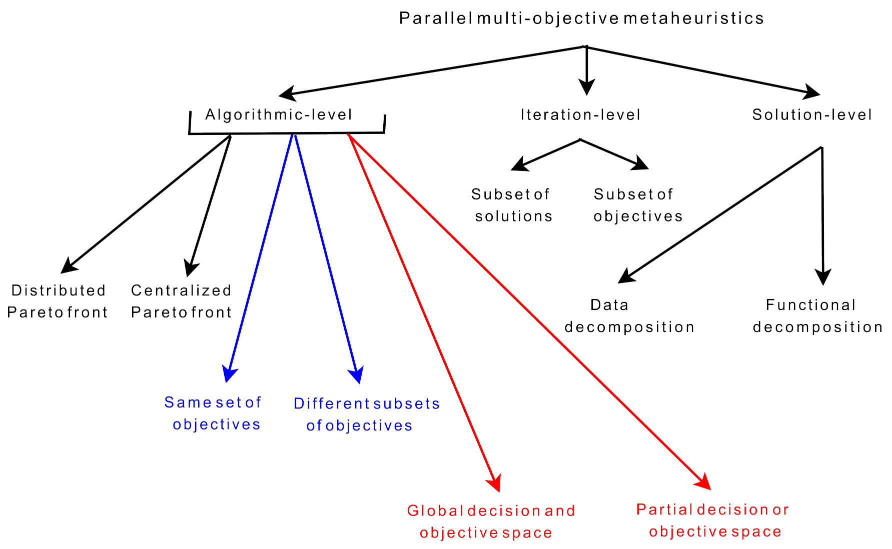

Figure 1: Classification of parallel MOEAs for multi-objective optimization.

computation. This way, the new non-dominated solutions in the Pareto optimal set are global Pareto optima [1, 5, 28].

- *Distributed Pareto Front*: the Pareto front is distributed among the MOEAs so that the algorithm works with local non-dominated solutions that must be somehow combined at the end of their work [8, 18, 19]. No pure centralized approach has been found clearly motivated by efficiency issues [16]. All the found centralized approaches are combined with distributed phases where local non-dominated solutions are considered. After each distributed phase, a single optimal Pareto front is built by using these local Pareto optima. Then, the new Pareto front is again distributed for local computation, and so on.

2.2 Iteration-Level Parallel Model

In this parallel model, a focus is made on the parallelization of each iteration of MOEAs. The iteration-level parallel model is generally based on the distribution of the handled solutions. Indeed, the most resource-consuming part in an MOEA is the evaluation of the generated solutions. Our concerns in this model are only search mechanisms that are problem-independent operations such as the generation of successive populations. Any *search operator* of an MOEA which is not specific to the tackled optimization problem is involved in the iteration-level parallel model.

This model keeps the sequentiality of the original algorithm, and, hence, the behavior of the MOEA is not altered.

It is the easiest and the most widely used parallel model in MOPs. Indeed, many MOPs are complex in terms of the objective functions. For instance, some engineering design applications integrate solvers dealing with different surrogate models: computational fluid dynamics (CFD), computational electromagnetics (CEM), or finite element methods (FEM). Other real-life applications deals with complex simulators. A particularly efficient execution is often obtained when the ratio between communication and computation is high. Otherwise, most of the time can be wasted in communications, leading to a poor parallel algorithm.

The population of individuals can be decomposed and handled in parallel. In *master-worker* a master performs the selection operations and the replacement. The selection and replacement are generally sequential procedures, as they require a global management of the population. The associated workers perform the recombination, mutation and the evaluation of the objective function. The master sends the partitions (subpopulations) to the workers. The workers return back newly evaluated solutions to the master [19] (Fig. 2).

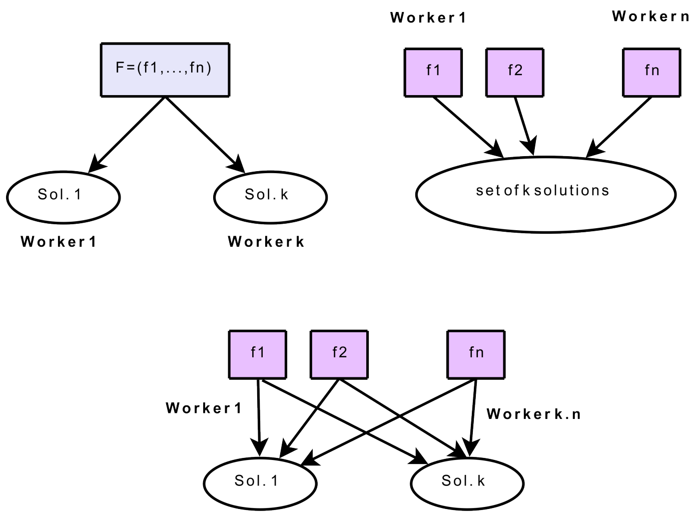

Figure 2: The iteration-level parallel model in parallel MOEAs.

According to the order in which the evaluation phase is performed in comparison with the other parts of the MOEA, two modes can be distinguished:

- **Synchronous:** in the synchronous mode, the worker manages the evolution process and performs in a serial way the different steps of selection and replacement. At each iteration, the master distributes the set of new generated solutions among the workers and waits for the results to be returned back. After the results are collected, the evolution process is re-started. The model does not change the behavior of the MOEA compared to a sequential model.

- **Asynchronous:** in the asynchronous mode, the worker does not wait for the return of all evaluations to perform the selection, reproduction and replacement steps. The *steady-state* MOEA is a good example illustrating the asynchronous model and its advantages. In the asynchronous model applied to a steady-state MOEA, the recombination and the evaluation steps may be done concurrently. The master manages the evolution engine and two queues of individuals of a given fixed size: individuals to be evaluated, and solutions being evaluated. The individuals of the first queue wait for a free evaluating node. When the queue is full the process blocks. The individuals of the second queue are assimilated into the population as soon as possible. The reproduced individuals are stored in a FIFO data structure, which represents the individuals to be evaluated. The MOEA continues its execution in an asynchronous manner, without waiting for the results of the evaluation phase. The selection and reproduction phase are carried out until the queue of non-evaluated individuals is full. Each evaluator agent picks an individual from the data structure, evaluates it, and stores the results into another data structure storing the evaluated individuals. The order of evaluation defined by the selection phase may not be the same as in the replacement phase. The replacement phase consists in receiving, in a synchronous manner, the results of the evaluated individuals, and applying a given replacement strategy of the current population.

In some MOEAs (e.g., blackboard-based ones) some information must be shared. For instance, in ant colony optimization (ACO), the pheromone matrix must be shared by all ants. The master has to broadcast the pheromone trails to each worker. Each worker handles an ant process. It receives the pheromone trails, constructs a complete solution, and evaluates it. Finally, each worker sends back to the master the constructed and evaluated solution. When the master receives all the constructed solutions, it updates the pheromone trails [14].

Ranking methods are used to assign a fitness to each solution of a population. Those ranking methods are computation-intensive and may be also parallelized. Updating the archives at each iteration is also a time consuming task.

2.3 Solution-Level Parallel Model

The main objective of the solution-level parallel model for MOP is to speedup the search by parallelizing the treatments dealing with single solutions (e.g., objectives evaluation, constraint satisfaction). Indeed, the evaluation of multiple objective functions in MOPs is the most time-consuming part into a MOEA. Therefore, several algorithms try to reduce this time by means of parallelizing the calculation of the fitness evaluation [21, 22, 23]. The classical approaches must be adapted to multi-objective optimization (Fig. 1):

- **Functional decomposition:** this approach consists in distributing the different objective functions among the workers, and each of them computes the value of its assigned function on each solution. The master will then aggregate the partial results for all the solutions. Such approach allows a degree of concurrency and the scalability is limited to the number of objective functions, meaning often 2 or 3. Moreover, each objective function may be decomposed into several sub-functions. Then, the degree of concurrency will be equal to the number of sub-functions.
- **Data decomposition:** for each data partition of the problem (database, geographical area, structure, ...), all the objectives of the problem are evaluated and returned to the master. The master will aggregate the different results.

In the multi-objective context, the scalability of this model is limited by the number of objectives and the number of sub-functions per objective. The scalability could be improved again if the different objective functions are simultaneously parallelized.

2.4 Hierarchical Combination of the Parallel Models

The three presented models for parallel MOEAs may be used in conjunction within a hierarchical structure [26]. The parallelism degree associated with this hybrid model is very important. Indeed, this hybrid model is very scalable; the degree of concurrency is $k * m * n$, where k is the number of MOEAs used, m is the size of the population, and n is the number of partitions or tasks associated to the evaluation of a single solution.

3. Parallel Implementation of MOEAs

Parallel implementation of MOEAs deals with the efficient mapping of a parallel model of MOEAs on a given parallel architecture.

3.1 Parallel Architectures

Parallel architectures are evolving quickly. The main criteria of parallel architectures, which will have an impact on the implementation of parallel MOEAs, are: memory sharing, homogeneity of resources, resource sharing by multiple users, scalability, and volatility. Those criteria will be used to analyze the different parallel models and their efficient implementation. A guideline is given for the efficient implementation of each parallel model of MOEAs according to each class of parallel architectures.

Shared memory/Distributed memory architectures: in shared memory parallel architectures, the processors are connected by a shared memory. There are different interconnection schemes for the network (e.g., bus, crossbar, multistage crossbar). This architecture is easy to program. Conventional operating systems and programming paradigms of sequential programming can be used. There is only one address space for data exchange but the programmer must take care of synchronization in memory access, such as the mutual exclusion in critical sections. This type of architecture has a poor scalability (from 2 to 128 processors in current technologies) and a higher cost. An example of such shared memory architectures are SMPs (Symmetric Multiprocessors) machines and multi-core processors.

In distributed memory architectures, each processor has its own memory. The processors are connected by a given interconnection network using different topologies (e.g., hypercube, 2D or 3D torus, fat-tree, multistage crossbars). This architecture is harder to program; data and/or tasks have to be explicitly distributed to processors. Exchanging information is also explicitly handled using message passing between nodes (synchronous or asynchronous communications). The cost of communication is not negligible and must be minimized to design an efficient parallel MOEA. However, this architecture has a good scalability in terms of the number of processors. In recent years, clusters of processors (COWs) became one of the most popular parallel distributed memory architectures. A good ratio between cost and performance is obtained with this class of architectures.

Homogeneous/Heterogenous parallel architectures: parallel architectures may be characterized by the homogeneity of the used processors, communication networks, operating systems, etc. For instance, COWs are in general homogeneous parallel architectures. The proliferation of powerful workstations and fast communication networks have shown the emergence of heterogeneous networks of workstations (NOWs) as platforms for high performance computing. This type of architecture is present in any laboratory, company, campus, institution, etc. These parallel platforms are generally composed of an important number of owned heterogeneous workstations shared by many users.

Shared/Non shared parallel architectures: most massively parallel machines (MPP) and clusters of workstations (COW) are generally non shared by the applications. Indeed, at a given time, the processors composing those architectures are dedicated to the execution of a single application. NOWs constitute a low-cost hardware alternative to run parallel algorithms but are in general shared by multiple users and applications.

Local network (LAN)/Wide-area network (WAN): massively parallel machines, clusters and local networks of workstations may be considered as tightly coupled architectures. Large networks of workstations and grid computing platforms are loosely coupled and are affected by a higher cost of communication. During the last decade, *grid computing* systems have been largely deployed to provide high performance computing platforms. A computational grid is a scalable pool of heterogeneous and dynamic resources geographically distributed across multiple administrative domains and owned by different organizations [9]. Two types of Grids may be distinguished:

- **High-Performance Computing Grid** (HPC Grid): this grid interconnect supercomputers or clusters via a dedicated high-speed network. In general, this type of grid is non-shared by multiple users (at the level of processors).
- **Desktop Grid**: this class of grids is composed of numerous owned workstations connected via non dedicated network such as the internet. This grid is volatile and shared by multiple users and applications.

Volatile/Non volatile parallel architectures: desktop grids constitute an example of volatile parallel architectures. In a volatile parallel architecture, there is a dynamic temporal and spatial availability of resources. In a desktop grid or a large network of shared workstations, volatility is not an exception but a rule. Due to the large scale nature of the grid, the probability of resource failure is high. For instance, desktop grids have a faulty nature (e.g., reboot, shutdown, failure).

The following table 2 recapitulates the characteristics of the main parallel architectures according to the presented criteria. Those criteria will be used to analyze the efficient implementation of the different parallel models of MOEAs.

Table 2: Characteristics of the main parallel architectures.

Criteria	Memory	Homogeneity	Sharing	Network	Volatility
SMP Multi-core	Shared	Hom	Yes or No	Local	No
COW	Distributed	Hom or Het	No	Local	No
NOW	Distributed	Het	Yes	Local	Yes
HPC Grid	Distributed	Het	No	Large	No
Desktop Grid	Distributed	Het	Yes	Large	Yes

Hom: Homogeneous, Het: Heterogeneous.

3.2 Dedicated Architectures

Dedicated hardware represents programmable hardware or specific architectures that can be designed or re-used to execute a parallel MOEA. The best known dedicated hardware is represented by Field Programmable Gate Arrays (FPGAs) and Graphical Processing Unit (GPU).

FPGAs are hardware devices that can be used to implement digital circuits by means of a programming process [31]. The use of the Xilinx's FPGAs to implement different MOEAs is more and more popular. The design and the prototyping of a FPGA-based hardware board to execute parallel MOEAs may restrict the design of some search components. However, for some specific challenging optimization problems with a high use rate such as in bioinformatics, dedicated hardware may be a good alternative.

GPU is a dedicated graphics rendering device for a workstation, personal computer, or game console. Recent GPUs are very efficient at manipulating computer graphics, and their parallel SIMD structure makes them more efficient than general-purpose CPUs for a range of complex algorithms [2]. The main companies producing GPUs are AMD and NVIDIA. The use of GPUs for an efficient implementation of MOEAs is a challenging issue [12, 13].

3.3 Parallel Programming Environments and Middlewares

The architecture of the target parallel machine strongly influences the choice of the parallel programming model to use. There are two main parallel programming paradigms: shared memory and message passing.

Two main alternatives exist to program shared memory architectures:

- **Multi-threading:** a thread may be viewed as a lightweight process. Different threads of the same process share some resources and the same address space. The main advantages of multi-threading are the fast context switch, the low resource usage, and the possible recovery between communication and computation. Each thread can be executed on a different processor or core. Multi-threaded programming may be used within libraries such as the standard Pthreads library [3] or programming languages such as Java threads [10].

- **Compiler directives:** one of the standard shared memory paradigms is OpenMP (Open Multi-Processing, `www.openmp.org`) and CUDA. It represents a set of compiler directives interfaced with the languages Fortran, C and C++ [4]. Those directives are integrated in a program to specify which sections of the program to be parallelized by the compiler.

Distributed memory parallel programming environments are based mainly on the three following paradigms:

- **Message passing:** message passing is probably the most widely used paradigm to program parallel architectures. Processes of a given parallel program communicate by exchanging messages in a synchronous or asynchronous way. The well known programming environments based on message passing are sockets and MPI (Message Passing Interface).

- **Remote Procedure Call:** Remote procedure call (RPC) represents a traditional way of programming parallel and distributed architectures. It allows a program to cause a procedure to execute on another processor.

- **Object oriented models:** as in sequential programming, parallel object oriented programming is a natural evolution of RPC. A classical example of such a model is Java RMI (Remote Method Invocation).

In the last decade, great work has been carried out on the development of grid middlewares. The Globus toolkit (`www.globus.org`) represents the *de facto* standard grid middleware. It supports the development of distributed service-oriented computing applications [20].

It is not easy to propose a guideline on which environment to use in programming a parallel MOEA. It will depend on the target architecture, the parallel model of MOEAs, and the user preferences. Some languages are more system oriented such as C and C++. More portability is obtained with Java but the price is less efficiency. This tradeoff represents the classical efficiency/portability compromise. A Fortran programmer will be more comfortable with OpenMP. RPC models are more adapted to implement services. Condor represents an efficient and easy way to implement parallel programs on shared and volatile distributed architectures such as large networks of heterogeneous workstations and desktop grids, where fault tolerance is ensured by a checkpoint/recovery mechanism. The use of MPI within Globus is more or less adapted to high performance computing (HPC) grids. However, the user has to deal with complex mechanisms such as dynamic load balancing and fault-tolerance. Table 3 presents a guideline depending on the target parallel architecture.

Table 3: Parallel programming environments for different parallel architectures.

Architecture	**Examples of suitable programming environment**
SMP Multi-core	Multi-threading library within an operating system (e.g., Pthreads) Multi-threading within languages: Java OpenMP interfaced with C, C++ or Fortran
COW	Message passing library: MPI interfaced with C, C++, Fortran
Hybrid ccNUMA	MPI or Hybrid models: MPI/OpenMP, MPI/Multi-threading
NOW	Message passing library: MPI interfaced with C, C++, Fortran Condor or object models (JavaRMI)
HPC Grid	MPICH-G (Globus) or GridRPC models (Netsolve, Diet)
Desktop Grid	Condor-G or object models (Proactive)

3.4 Performance Evaluation

For sequential algorithms, the main performance measure is the execution time as a function of the input size. In parallel algorithms, this measure depends also on the number of processors and the characteristics of the parallel architecture. Hence, some classical performance indicators such as speedup and efficiency have been introduced to evaluate the scalability of parallel algorithms [11]. The scalability of a parallel algorithm measures its ability to achieve performance proportional to the number of processors.

The speed-up S_N is defined as the time T_1 it takes to complete a program with one processor divided by the time T_N it takes to complete the same program with N processors

$$S_N = \frac{T_1}{T_N}.$$

One can use *wall-clock time* instead of *CPU time*. The CPU time is the time a processor spends in the execution of the program, and the wall-clock time is the time of the whole program including the input and output. Conceptually the speed-up is defined as the gain achieved by parallelizing a program. If $S_N > N$ (resp. $S_N = N$), a super-linear (resp. linear) speedup is obtained [25]. Mostly, a sub-linear speedup $S_N < N$ is obtained. This is due to the overhead of communication and synchronization costs. The case $S_N < 1$ means that the sequential time is smaller than the parallel time which is the worst case. This will be possible if the communication cost is much higher than the execution cost.

The efficiency E_N using N processors is defined as the speed-up S_N divided by the number of processors N.

$$E_N = \frac{S_N}{N}.$$

Conceptually the efficiency can be defined as how well N processors are used when the program is computed in parallel. An efficiency of 100% means that all of the processors are fully used all the time. For some large real-life applications, it is impossible to have the sequential time as the sequential execution of the algorithm cannot be performed. Then, the incremental efficiency E_{NM} may be used to evaluate the efficiency extending the number of processors from N to M processors.

$$E_{NM} = \frac{N \times E_N}{M \times E_M}.$$

Different definitions of speedup may be used depending on the definition of the sequential time reference T_1. Asking what is the best measure is useless; there is no global dominance between the different measures. The choice of a given definition depends on the objective of the performance evaluation analysis. Then, it is important to specify clearly the choice and the objective of the analysis.

The *absolute speedup* is used when the sequential time T_1 corresponds to the best known sequential time to solve the problem. Unlike other scientific domains such as numerical algebra where for some operations the best sequential algorithm is known, in MOEA search, it is difficult to identify the best sequential algorithm. So, the absolute speedup is rarely used. The *relative speedup* is used when the sequential time T_1 corresponds to the parallel program executed on a single processor.

Moreover, different stopping conditions may be used:

- **Fixed number of iterations:** this condition is the most used to evaluate the efficiency of a parallel MOEA. Using this definition, a superlinear speedup is possible $S_N > N$ [7]. This is due to the characteristics of the parallel architecture where there is more resources (e.g., size of main memory and cache) than in a single processor. For instance, the search memory of an MOEA executed on a single processor may be larger than the main memory of a single processor and then some swapping will be carried out, which represents an overhead in the sequential time. When using a parallel architecture, the whole memory of the MOEA may fit in the main memory of its processors, and then the memory swapping overhead will not occur.
- **Convergence to a set of solutions with a given quality:** this measure is interesting to evaluate the effectiveness of a parallel MOEA. It is only valid for parallel models of MOEAs based on the algorithmic-level, which alter the behavior of the sequential MOEA. A super-linear speedup is possible and is due to the characteristics of the parallel search. Indeed, the order of searching different regions of the search space may be different from sequential search. The sequences of visited solutions in parallel and sequential search are different. This is similar to the super-linear speedups obtained in exact search algorithms such as branch and bound [24].

Most of evolutionary algorithms are stochastic algorithms. When the stopping condition is based on the quality of the solution, one cannot use the speedup metric as defined previously. The original definition may be extended to the average speedup:

$$S_N = \frac{E(T_1)}{E(T_N)}.$$

The same *seed* for the generation of random numbers must be used for a more fair experimental performance evaluation. The speedup metrics have to be reformulated for heterogeneous architectures. The efficiency metric may be used for this class of architectures. Moreover, it can be used for shared parallel machines with multiple users.

3.5 Main Properties of Parallel MOEAs

The performance of a parallel MOEA on a given parallel architecture depends mainly on its *granularity*. The granularity of a parallel program is the amount of computation performed between two communications. It computes the ratio between the computation time and the communication time. The three parallel models (algorithmic-level, iteration-level, solution-level) have a decreasing granularity from coarse-grained to fine-grained. The granularity indicator has an important impact on the speedup. The larger is the granularity the better is the obtained speedup.

The *degree of concurrency* of a parallel MOEA is represented by the maximum number of parallel processes at any time. This measure is independent from the target parallel architecture. It is an indication of the number of processors that can employed usefully by the parallel MOEA. Asynchronous communications and the recovery between computation and communication is also an important issue for a parallel efficient implementation. Indeed, most of the actual processors integrate different parallel elements such as ALU, FPU, GPU, DMA, etc. Most of the computing part takes part in cache. Hence, the RAM bus is often free and can be used by other elements such as the DMA. Hence, input/output operations can be recovered by computation tasks.

Scheduling the different tasks composing a parallel MOEA is another classical issue to deal with for their efficient implementation. Different scheduling strategies may be used depending on whether the number and the location of works (tasks, data) depend or not on the load state of the target machine:

- **Static scheduling**: this class represents parallel MOEAs in which both the number of tasks of the application and the location of work (tasks, data) are generated at compile time. Static scheduling is useful for homogeneous, and non shared and non volatile heterogeneous parallel architectures. Indeed, when there are noticeable load or power differences between processors, the search time of an iteration is derived by the maximum execution time over all processors, presumably on the most highly loaded processor or the least powerful processor. A significant number of tasks are often idle waiting for other tasks to complete their work.

- **Dynamic scheduling**: this class represents parallel MOEAs for which the number of tasks is fixed at compile time, but the location of work is determined and/or changed at run-time. The tasks are dynamically scheduled on the different processors of the parallel architecture. Dynamic load balancing is important for shared (multi-user) architectures, where the load of a given processor cannot be determined at compile time. Dynamic scheduling is also important for *irregular* parallel MOEAs in which the execution time cannot be predicted at compile time and varies during the search. For instance, this happens when the evaluation cost of the objective functions depends on the solution.

- **Adaptive scheduling**: parallel adaptive algorithms are parallel computations with a dynamically changing set of tasks. Tasks may be created or killed as a function of the load state of the parallel machine. A task is created automatically when a node becomes idle. When a node becomes busy, the task is killed. Adaptive load balancing is important for volatile architectures such as desktop grids.

For some parallel and distributed architectures such as shared networks of workstations and grids, fault tolerance is an important issue. Indeed, in volatile shared architectures and large-scale parallel architectures, the fault probability is relatively important. Checkpointing and recovery techniques constitute one answer to this problem. Application-level checkpointing is much more efficient than system-level checkpointing. Indeed, in system-level checkpointing, a checkpoint of the global state of a distributed application composed of a set of processes is carried out. In application-level checkpointing, only minimal information will be checkpointed (e.g., population of individuals, generation number). Compared to system-level checkpointing, a reduced cost is then obtained in terms of memory and time. Finally, security issues may be important for large-scale distributed architectures such as grids and Clouds (multi-domain administration, firewall, etc) and some specific applications such as medical and bioinformatics research applications of industrial concern [30].

3.6 Algorithmic-Level Parallel Model

Granularity: the algorithmic-level parallel model has the largest granularity. Indeed, the time for exchanging the information is in general much less than the computation time of a MOEA. There are relatively low communication requirements for this model. The more important is the frequency of exchange and the size of exchanged information, the smaller is the granularity. This parallel model is the most suited to large-scale distributed architectures over internet such as grids. Moreover, the trivial model with independent algorithms is convenient for low-speed networks of workstations over intranet. As there is no essential dependency and communication between the algorithms, the speedup is generally linear for this parallel model. The size of the data exchanged (for instance the number of Pareto solutions) will influence the granularity of the model. If the number of Pareto solutions is high the communication cost will be exorbitant particularly on a large-scale parallel architectures such as grids.

For an efficient implementation, the frequency of exchange (resp. the size of the exchanged data) must be correlated to the latency (resp. bandwidth) of the communication network of the parallel architecture. To optimize the communication between processors, the exchange topology can be specified according to the interconnection network of the parallel architecture. The specification of the different parameters associated with the blind or intelligent migration decision criterion (migration frequency/probability and improvement threshold) is particularly crucial on a computational grid. Indeed, due to the heterogeneous nature of computational grids these parameters must be specified for each MOEA in accordance with the machine it is hosted on.

Scalability: the degree of concurrency of the algorithmic-level parallel model is limited by the number of MOEAs involved in solving the problem. In theory, there is no limit. However, in practice, it is limited by the owned resources of the target parallel architectures, and also by the effectiveness aspect of using a large number of MOEAs.

Synchronous versus asynchronous communications: the implementation of the algorithmic-level model is either *asynchronous* or *synchronous*. The asynchronous mode associates with each MOEA an exchange decision criterion, which is evaluated at each iteration of the MOEA from the state of its memory. If the criterion is satisfied, the MOEA communicates with its neighbours. The exchange requests are managed by the destination MOEAs within an undetermined delay. The reception and integration of the received information is thus performed during the next iterations. However, in a computational grid context, due to the material and/or software heterogeneity issue, the MOEAs could be at different evolution stages leading to the *non-effect* and/or *super-solution* problem. For instance, the arrival of poor solutions at a very advanced stage will not bring any contribution as these solutions will likely not be integrated. In the opposite situation, the cooperation will lead to premature convergence.

From another point of view, as it is non-blocking, the model is more efficient and fault tolerant to such a degree a threshold of wasted exchanges is not exceeded. In the synchronous mode, the MOEAs perform a synchronization operation at a predefined iteration by exchanging some data. Such operation guarantees that the MOEAs are at the same evolution stage, and so prevents the non-effect and super-solution problem quoted before. However, in heterogeneous parallel architectures, the synchronous mode is less efficient in term of consumed CPU time. Indeed, the evolution process is often hanging on powerful machines waiting the less powerful ones to complete their computation. The synchronous model is also not fault tolerant as a fault of a single MOEA implies the blocking of the whole model in a volatile environment. Then, the synchronous mode is globally less efficient on a computational grid.

Asynchronous communication is more efficient than synchronous communication for shared architectures such as NOWs and desktop grids (e.g., multiple users, multiple applications). Indeed, as the load of networks and processors is not homogeneous, the use of synchronous communication will degrade the performances of the whole system. The least powerful machine will determine the performance.

On a volatile computational grid, it is difficult to efficiently maintain topologies such as rings and torus. Indeed, the disappearance of a given node (i.e., MOEA) requires a dynamic reconfiguration of the topology. Such reconfiguration is costly and makes the migration process inefficient. Designing a cooperation between a set of MOEAs without any topology may be considered. For instance, a communication scheme in which the target MOEA is selected randomly is more efficient for volatile architecture such as desktop grids. Many experimental results show that such topology allows a significant improvement of the robustness and quality of solutions. The random topology is therefore thinkable and even commendable in a computational grid context.

Scheduling: concerning the scheduling aspect, in the algorithmic-level parallel model the tasks correspond to MOEAs. Hence, the different scheduling strategies will differ as follows:

- Static scheduling: the number of MOEAs is constant and correlated to the number of processors of the parallel machine. A static mapping between the MOEAs and the processors is realized. The localization of MOEAs will not change during the search.
- Dynamic scheduling: MOEAs are dynamically scheduled on the different processors of the parallel architecture. Hence, the migration of MOEAs during the search between different machines may happen.
- Adaptive scheduling: the number of MOEAs involved into the search will vary dynamically. For example, when a machine becomes idle, a new MOEA is launched to perform a new search. When a machine becomes busy or faulty, the associated MOEA is stopped.

Fault-tolerance: the memory state of the algorithmic-level parallel model required for the checkpointing mechanism is composed of the memory of each MOEA and the information being migrated (i.e., population, archive, generation number).

3.7 Iteration-Level Parallel Model

Granularity: a medium granularity is associated to the iteration-level parallel model. The ratio between the evaluation of a partition and the communication cost of a partition determines the granularity. This parallel model is then efficient if the evaluation of a solution is time-consuming and/or there are a large number of candidate solutions to evaluate. The granularity will depend on the number of solutions in each sub-population.

Scalability: the degree of concurrency of this model is limited by the size of the population. The use of large populations will increase the scalability of this parallel model.

Synchronous versus asynchronous communications: introducing asynchronism in the iteration-level parallel model will increase the efficiency of parallel MOEAs. In the iteration-level parallel model, asynchronous communications are related to the asynchronous evaluation of partitions and construction of solutions. Unfortunately, this model is more or less synchronous. Asynchronous evaluation is more efficient for heterogeneous or shared or volatile parallel architectures. Moreover, asynchronism is necessary for optimization problems where the computation cost of the objective function (and constraints) depends on the solution and different solutions may have different evaluation cost.

Asynchronism may be introduced by relaxing the synchronization constraints. For instance, steady-state algorithms may be used in the reproduction phase [6].

The two main advantages of the asynchronous model over the synchronous model are fault tolerance and robustness if the fitness computation takes very different computations time. 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. Therefore, the model is blocking and, thus, less efficient on a heterogeneous computational grid. Moreover, as the model is not fault tolerant, the disappearance of an evaluating agent requires the redistribution of its individuals to other agents. As a consequence, it is essential to store all the solutions not yet evaluated. The scalability of the model is limited to the size of the population.

Scheduling: in the iteration-level parallel model, tasks correspond to the construction/evaluation of a set of solutions. Hence, the different scheduling strategies will differ as follows:

- Static scheduling: here, a static partitioning of the population is applied. For instance, the population is decomposed into equal size partitions depending on the number of processors of the parallel homogeneous non-shared machine. A static mapping between the partitions and the processors is realized. For a heterogeneous non-shared machine, the size of each partition must be initialized according to the performance of the processors. The static scheduling strategy is not efficient for variable computational costs of equal partitions. This happens for optimization problems where different costs are associated to the evaluation of solutions. For instance, in genetic programming individuals may widely vary in size and complexity. This makes a static scheduling of the parallel evaluation of the individuals not efficient.

- Dynamic scheduling: a static partitioning is applied but a dynamic migration of tasks can be carried out depending on the varying load of processors. The number of tasks generated may be equal to the size of the population. Many tasks may be mapped on the same processor. Hence, more flexibility is obtained for the scheduling algorithm. For instance, the approach based on the master-workers cycle stealing may be applied. To each worker is first allocated a small number of solutions. Once it has performed its iterations the worker requests from the master additional solutions. All the workers are stopped once the final result is returned. Faster and less loaded processors handle more solutions than the others. This approach allows to reduce the execution time compared to the static one.

- Adaptive scheduling: the objective in this model is to adapt the number of partitions generated to the load of the target architecture. More efficient scheduling strategies are obtained for shared, volatile and heterogeneous parallel architectures such as desktop grids.

Fault-tolerance: the memory of the iteration-level parallel model required for the checkpointing mechanism is composed of different partitions. The partitions are composed of a set of (partial) solutions and their associated objective values.

3.8 Solution-Level Parallel Model

Granularity: this parallel model has a fine granularity. There is a relatively high communication requirements for this model. In the functional decomposition parallel model, the granularity will depend on the ratio between the evaluation cost of the sub-functions and the communication cost of a solution. In the data decomposition parallel model, it depends on the ratio between the evaluation of a data partition and its communication cost.

The fine granularity of this model makes it less suitable for large-scale distributed architectures where the communication cost (in terms of latency and/or bandwidth) is relatively important, such as in grid computing systems. Indeed, its implementation is often restricted to clusters or network of workstations or shared memory machines.

Scalability: the degree of concurrency of this parallel model is limited by the number of sub-functions or data partitions. Although its scalability is limited, the use of the solution-level parallel model in conjunction with the two other parallel models enables to extend the scalability of a parallel MOEA.

Synchronous versus asynchronous communications: the implementation of the solution-level parallel model is always synchronous following a master-workers paradigm. Indeed, the master must wait for all partial results to compute the global value of the objective functions. The execution time T will be bounded by the maximum time T_i of the different tasks. An exception occurs for hard-constrained optimization problems, where feasibility of the solution is first tested. The master terminates the computations as soon as a given task detects that the solution does not satisfy a given hard constraint. Due to its heavy synchronization steps, this parallel model is worth applying to problems in which the calculations required at each iteration are time consuming. The relative speedup may be approximated as follows:

$$S_n = \frac{T}{\alpha + T/n},$$

where α is the communication cost.

Scheduling: in the solution-level parallel model, tasks correspond to sub-functions in the functional decomposition and to data partitions in the data decomposition model. Hence, the different scheduling strategies will differ as follows:

- Static scheduling: usually, the sub-functions or data are decomposed into equal size partitions depending on the number of processors of the parallel machine. A static mapping between the sub-functions (or data partitions) and the processors is applied. As for the other parallel models, this static scheme is efficient for parallel homogeneous non-shared machines. For a heterogeneous non-shared machine, the size of each partition in terms of sub-functions or data must be initialized according to the performance of the processors.

- Dynamic scheduling: dynamic load balancing will be necessary for shared parallel architectures or variable costs for the associated sub-functions or data partitions. Dynamic load balancing may be easily achieved by evenly distributing at run-time the sub-functions or the data among the processors. In optimization problems, where the computing cost of the sub-functions is unpredictable, dynamic load balancing is necessary. Indeed, a static scheduling cannot be efficient because there is no appropriate estimation of the task costs (i.e., unpredictable cost).

- Adaptive scheduling: in adaptive scheduling, the number of sub-functions or data partitions generated is adapted to the load of the target architecture. More efficient scheduling strategies are obtained for shared, volatile and heterogeneous parallel architectures such as desktop grids.

Fault-tolerance: the memory of the solution-level parallel model required for the checkpointing mechanism is straightforward. It is composed of the solution(s) and their partial objective value calculations.

Depending on the target parallel architecture, table 4 presents a general guideline for the efficient implementation of the different parallel models of MOEAs. For each parallel model (algorithmic-level, iteration-level, solution-level), the table shows its characteristics according to the outlined criteria (granularity, scalability, asynchronism, scheduling and fault-tolerance).

4. Conclusions and Perspectives

Parallel and distributed computing can be used in the design and implementation of MOEAs to speedup the search, to improve the quality of the obtained solutions, to improve the robustness, and to solve large scale problems. The clear separation between parallel design and parallel implementation aspects of MOEAs is important to analyze parallel MOEAs. The most important lessons of this paper can be summarized as follows:

- In terms of parallel design, the different parallel models for MOEAs have been unified. Three hierarchical parallel models have been

Table 4: Efficient implementation of parallel MOEAs according to some performance metrics and used strategies.

Property	**Algorithmic-level**	**Iteration-level**	**Solution-level**
Granularity	Coarse (Frequency of exchange, size of information)	Medium (Nb. of solutions per partition)	Fine (Eval. sub-functions, eval. data partitions)
Scalability	Number of MOEAs	Neighborhood size, populations size	Nb. of sub-functions, nb. data partitions
Asynchronism	High (Information exchange)	Moderate (Eval. of solutions)	Exceptional (Feasibility test)
Scheduling and Fault-tolerance	MOEA	Solution(s)	Partial solution(s)

extracted: algorithmic-level, iteration-level and solution-level parallel models.

- In terms of parallel implementation, the question of an efficient mapping of a parallel model of MOEAs on a given parallel architecture and programming environment (i.e., language, library, middleware) is handled. The focus was made on the key criteria of parallel architectures that influence the efficiency of an implementation of parallel MOEAs.

One of the perspectives in the coming years is to achieve Exascale performance. The emergence of heterogeneous platforms composed of multicore chips and many-core chips technologies will speedup the achievement of this goal. In terms of programming models, cloud computing will become an important alternative to traditional high performance computing for the development of large-scale MOEAs that harness massive computational resources. This is a great challenge as nowadays cloud frameworks for parallel MOEAs are just emerging.

In the future design of high-performance computers, the ratio between power and performance will be increasingly important. The power represents the electrical power consumption of the computer. An excess in power consumption uses unnecessary energy, generates waste heat and decreases reliability. Very few vendors of high-performance architecture publicize the power consumption data compared to the performance data.

In terms of target optimization problems, parallel MOEAs constitute unavoidable approaches to solve large scale real-life challenging problems (e.g., engineering design, data mining). They are also an important alternative to solve dynamic and uncertain optimization MOPs, in which the complexities in terms of time and quality are more difficult to handle by traditional sequential approaches. Moreover, parallel models for MOPs with uncertainty have to be deeply investigated.

Acknowledgement: This work is part of a project that has received funding from the *European Unions Horizon 2020 research and innovation program* under grant agreement No 692286.

References

[1] M. Basseur, F. Seynhaeve, and E.-G. Talbi. Adaptive mechanisms for multi-objective evolutionary algorithms. *Proceedings of the Congress on Engineering in System Application (CESA)*, pages 72–86, 2003.

[2] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. Graphics Processing Unit (GPU) Programming Strategies and Trends in GPU Computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[3] D. R. Butenhof. *Programming with POSIX threads.* Addison-Wesley, 1997.

[4] B. Chapman, G. Jost, R. van der Pas, and D. J. Kuck. *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT Press, 2007.

[5] C. A. Coello and M. Reyes. A study of the parallelization of a coevolutionary multi-objective evolutionary algorithm. *Lecture Notes in Artificial Intelligence*, 2972:688–697, 2004.

[6] M. Depolli, R. Trobec, and B. Filipič. Asynchronous master-slave parallelization of differential evolution for multi-objective optimization. *Evolutionary Computation*, 21(2):261–291, 2013.

[7] B. Dorronsoro, G. Danoy, A. J. Nebro, and P. Bouvry. Achieving super-linear performance in parallel multi-objective evolutionary algorithms by means of cooperative coevolution. *Computers & Operations Research*, 40(6):1552–1563, 2013.

[8] S. Duarte and B. Barán. Multiobjective network design optimisation using parallel evolutionary algorithms. *Proceedings of the XXVII Conferencia Latinoamericana de Informática (CLEI)*, 2001.

[9] I. Foster and C. Kesselman (Eds.) *The grid: Blueprint for a new computing infrastructure.* Morgan Kaufmann, San Fransisco, 1999.

[10] P. Hyde. *Java thread programming.* Sams, 1999.

[11] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: Design and analysis of algorithms.* Addison Wesley, 1994.

[12] Z. Li, Y. Bian, R. Zhao, and J. Cheng. A Fine-Grained Parallel Multi-objective Test Case Prioritization on GPU. *Lecture Notes in Computer Science*, 8084:111–125, 2013.

[13] T. V. Luong, E. Taillard, N. Melab, and E-G. Talbi. Parallelization Strategies for Hybrid Metaheuristics Using a Single GPU and Multi-core Resources. *Lecture Notes in Computer Science*, 7492:368–377, 2012.

[14] A. M. Mora, P. García-Sánchez, J. J. Merelo Guervós, and P. A. Castillo. Pareto-based multi-colony multi-objective ant colony optimization algorithms: an island model proposal. *Soft Computing*, 17(7):1175–1207, 2013.

[15] S. Mostaghim, J. Branke, and H. Schmeck. Multi-objective particle swarm optimization on computer grids. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 869–875, 2007.

[16] A. J. Nebro, F. Luna, E.-G. Talbi, and E. Alba. Parallel multiobjective optimization. In E. Alba (Ed.) *Parallel metaheuristics*, Wiley, 2005, pages 371–394.

[17] S. Nesmachnow. Parallel multiobjective evolutionary algorithms for batch scheduling in heterogeneous computing and grid systems. *Computational Optimization and Applications*, 55(2):515–544, 2013.

[18] K. E. Parsopoulos, D. K. Tasoulis, N. G. Pavlidis, V. P. Plagianakos, and M. N. Vrahatis. Vector evaluated differential evolution for multiobjective optimization. *Proceedings of the IEEE 2004 Congress on Evolutionary Computation (CEC'04)*, 2004.

[19] J. Rowe, K. Vinsen, and N. Marvin. Parallel GAs for multiobjective functions. *Proceedings of the 2nd Nordic Workshop on Genetic Algorithms and Their Applications (2NWGA)*, pages 61–70, 1996.

[20] B. Sotomayor and L. Childers. *Globus toolkit 4: Programming Java services.* Morgan Kaufmann, 2005.

[21] N. Srinivas and K. Deb. Multiobjective optimization using non-dominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1995.

[22] T. J. Stanley and T. Mudge. A parallel genetic algorithm for multi-objetive microprocessor design. *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 597–604, 1995.

[23] R. Szmit and A. Barak. Evolution Strategies for a Parallel Multi-Objective Genetic Algorithm. P*roceedings of the Genetic and Evolutionary Computation Conference*, pages 227–234, 2000.

[24] E.-G. Talbi. *Parallel combinatorial optimization.* Wiley, 2006.

[25] E.-G. Talbi and P. Bessière. Superlinear speedup of a parallel genetic algorithm on the SuperNode. *SIAM News*, 24(4):12–27, 1991.

[26] E.-G. Talbi, S. Cahon, and N. Melab. Designing cellular networks using a parallel hybrid metaheuristic on the computational grid. *Computer Communications*, 30(4):698–713, 2007.

[27] E.-G. Talbi, S. Mostaghim, H. Ishibushi, T. Okabe, G. Rudolph, and C.C. Coello. Parallel approaches for multi-objective optimization. *Lecture Notes in Computer Science*, 5252:349–372, 2008.

[28] F. de Toro, J. Ortega, E. Ros, S. Mota, B. Paechter, and J.M. Martín. PSFGA: Parallel processing and evolutionary computation for multiobjective optimisation. *Parallel Computing*, 30(5-6):721–739, 2004.

[29] D. A. van Veldhuizen, J. B. Zydallis, and G. B. Lamont. Considerations in engineering parallel multi-objective evolutionary algorithms. *IEEE Transasctions on Evolutionary Computation*, 7(2):144–173, 2003.

[30] G. Yao, Y. Ding, Y. Jin, and K. Hao. Endocrine-based coevolutionary multi-swarm for multi-objective workflow scheduling in a cloud system. *Soft Computing*, 1–14, 2016.

[31] R. Zeidman. *Designing with FPGAs and CPLDs.* Elsevier, 2002.