PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

GOSSIPING OPTIMIZATION FRAMEWORK (GOOF)
A DECENTRALIZED P2P ARCHITECTURE FOR FUNCTION OPTIMIZATION

Marco Biazzini

Advisor:

Prof. Alberto Montresor

Università degli Studi di Trento

March 2010

Abstract

This thesis discusses the implementation of function optimization algorithms through distributed and decentralized processing in a peer-to-peer fashion.
Our research is focused on a fully decentralized, general purpose P2P environment, with no special or ad-hoc facility for executing optimization tasks. Relevant information is exchanged among nodes by means of epidemic protocols, exploiting the overlay network topology formed by peers. A key issue in such a context is the relationship between the solution quality and the amount/kind of exchanged information among the various running instances. We propose and detail novel heuristics and hyper-heuristics. Experimental results obtained both in simulated and real P2P environments are presented and discussed as well.
*Distributed optimization has a long and rich history, but little has been done to make it exploit the (potentially) large computing facilities a reliable P2P network can provide. We propose a novel framework that aims at easing the burden of performing function optimization tasks in a decentralized P2P network of solvers. Our 'GOssiping Optimization Framework' (*GOOF*) bridges the gap between P2P services that can provide large reliable networks of interconnected nodes and the needs of optimization practitioners who are often not able to find a reasonably simple way to run their algorithms in such a distributed environment.*

Keywords

P2P protocols, optimization search heuristics,
distributed computing, function optimization framework

Contents

List of Tables

List of Figures

Acknowledgements

I'd like to thank in these few lines some people who made it possible to me to walk the road of research and finally defend my work as presented in this Thesis.

Starting with Prof. Roberto Battiti who introduced me to this path and gave me the first precious advises. Then Prof. Mauro Brunato, with whom I collaborated in the early stages of my work. Special thanks go to my advisor, Prof. Alberto Montresor for his kind and expert guidance throughout these years. I'm really grateful to Dr. Márk Jelásity for his kind and valuable support and for his collaboration, which I took from much more than I could give.

Not only fine scientists helped me through these years, but also the friendship and the affection of many people. Too many for a list: I write here a "thank you" to them all. They are anyway enlisted in my heart.

Finally — *dulcis in fundo* — my love and my deepest thanks to the most special persons in my life: to my dear Nata, who simply kept (and still keeps) me alive with her love and her restless caring in good and bad times; and to my parents, who have always supported everything that I have been and everything that I have done.

Chapter 1

Introduction

The job of scientists working on function optimization often involves two different aspects [17]: on the one hand, they are required to benchmark newly developed optimization algorithms against a large collection of existing problems; on the other hand, when confronted with a specific optimization problem, they may be required to attack it with a large arsenal of optimization techniques, to obtain the best possible results.

When the distributed dimension is added, it makes this already complicated scenario even worse. Can existing algorithms be plugged in a distributed environment? Can multiple instances of the same algorithm cooperate? Can multiple algorithms share their work-in-progress results? How multiple instances are organized, and how they communicate? Is there a class of optimization problems that are more appropriate for a distributed environment?

Distributed optimization has a long research history [131]. Most of the previous work assumes the availability of either a dedicated parallel computing facility, or, in the worst case, specialized clusters of networked machines that are coordinated in a centralized fashion (master-slave, coordinator-cohort, etc.). While these approaches simplify management, they normally show severe limitations with respect to scalability and robustness.

The goal of our work is to investigate an alternative approach to distributed

function optimization. The idea is to adopt recent results in the domain of large-scale decentralized systems and peer-to-peer (P2P) systems, where a large collection of loosely-coupled machines cooperate to achieve a common goal. Instead of requiring a specialized infrastructure or a central server, such systems self-organize themselves in a completely decentralized way, avoiding single points of failure and performance bottlenecks. The advantages of such approach are thus extreme robustness and scalability, and the capability of exploiting existing (unused or underused) resources.

The applicative scenario we have in mind is a potentially large organization that owns, or at least controls, several hundreds or even thousands of personal workstations, and wants to exploit their idle periods to perform optimization tasks. In such systems, high level of *churn* may be expected: nodes may join and leave the system at will, for example when users start or stop to work at their workstations.

Such a scenario is not unlike a Grid system [84]; a reasonable approach could thus be to collect a pool of independent optimization tasks to be performed, and assign each of them to one of the available nodes, taking care of balancing the load. This can be done either using a centralized scheduler, or using a decentralized approach [73].

An interesting question is whether it is possible to come up with an alternative approach, where a distributed algorithm spreads the load of a *single* optimization task among a group of nodes, in a robust, decentralized and scalable way. We can rephrase the question as follows: can we make a better use of our distributed resources by making them cooperate on a single optimization process? Two possible motivations for such approach come to mind: we want to obtain a more accurate result by a specific deadline (focus on quality), or we are allowed to perform a predefined amount of computation over a function and we want to obtain a quick answer (focus on speed).

To be able to do this as smoothly as possible, it would definitely be helpful

to have a distributed framework, hosted on every participant node, that can ease the burden of deploying optimization algorithms in such a fully decentralized fashion. A uniform environment, which the optimization task relies on, to carry on the quest for the optimal value and coordinate all the resources that are spread on different machines.

When it comes to making optimization algorithms collaborate and share information, several aspects equally matter. Due to the huge variety of existing and possible optimization algorithms, the framework has to be generic enough to suit most (if not any) of them. It must be designed in such a way that it is not too difficult to plug a new algorithm in the system, meanwhile making the execution of the algorithm not unusual to the user, with respect to the procedure that is well known when the same algorithm is used in a different environment.

Another requirement is interoperability. Different optimization algorithms, having very little in common, should be able to cooperate to solve the same problem with little or no user intervention. On the other way round, the same optimization algorithm should be able to operate on a large number of different problems with no adjustment or change in either of the two entities being required (except for problem-specific tuning, of course).

No striving should be asked to the user in order to handle and set up the P2P network of solvers. A basic parametrization of the amount of communications to be enabled should be all that the user needs to know about the way the solvers spread information throughout the network. Moreover, the framework should provide a clear and standard way of handling the event of receiving relevant information and making a proper use of it locally.

To help scientists in exploring and exploiting a large number of novel experimental configurations, we have designed and developed GOOF, which stands for 'GOssiping Optimization Framework'. It can handle in a consistent way the process of information sharing between different solvers, according to the user's decisions. Moreover, it transparently handles the differences among the

optimization processes which run in different network nodes, giving a powerful contribution to solve the aforementioned issues.

Structure of the Thesis

Chapter 2 presents a brief contextualization of the problem. We focus the scope of our contribution describing recent interesting achievements in the fields of distributed computing and function optimization. Chapter 3 describes the general architecture of the framework we propose, detailing on the goals and the addressed issues of the design. In Chapter 4 we detail the optimization core of our framework, describing its features and presenting the various algorithms we developed and integrated in framework. Chapter 5 focuses on how the framework provides effective communication between the solvers, pointing out relevant characteristics and versatility. Chapter 6 displays and discuss several experimental results we obtained by using our algorithms, also showing how our framework can successfully cope with different requirements. Conclusive remarks are drawn in Chapter 7.

Chapter 2

State of the Art

Since the very beginning, our research effort has lead us to be involved in different fields. Deeply belonging to the function optimization field, strongly connected with the distributed computing and the peer-to-peer paradigm as well. Aware as we are that we cannot cover in details the recent history of all of them, we present in this chapter a synthetical survey of those topics that are mostly relevant to our work, discussing some of the latest results that relate to our contribution.

In the following we thus examine the field of distributed computing and peer-to-peer networks (Section 2.1), then we present some recent results in the field of optimization, concentrating on those obtained by using search heuristics and evolutionary techniques (Section 2.2).

2.1 Parallel and distributed computing

Parallel and distributed computing has been used for years in order to tackle the complexity of many tasks. The existing different scenarios vary with respect of many characteristics, but you can observe some recurring trends in the way they are designed. This fact allows you to look at them through the following, rough classification:

- **Independent tasks within a tightly coupled system**
  Here we can list all the parallelization flavors of the various algorithms, that are designed and implemented requiring a parallel multiprocessor architecture or an equivalent dedicated infrastructure. The dynamism of the system is very poor and there are strong limitations with respect to scalability and robustness.

- **Related tasks handled by a central coordinator within a loosely coupled system**
  The main example of this kind of distributed computing architecture is the so-called "collaborative search". This expression indicates the fact that different entities give their own contribution in order to reach the general goal of performing a searching task (See the BOINC Project[1] as an example). Although working in a totally distributed environment, projects like these are mostly designed in a centralized fashion. This means that there is a unique coordinator that assigns tasks, collects results and combines the outcomes to find an acceptable solution. Such an architectural design is desirable when the search space – more precisely, the interesting or most promising part of it – is already known. But this is not always the case...

2.1.1 P2P applications networks

A third paradigm has actually been devised, that is highly popular and studied in many fields — both belonging to scientific research and not — but not yet fully exploited for optimization purposes. We are talking about peer-to-peer (P2P) applications networks. During the past decade various large scale networks have emerged as computing platforms such as the Internet, the web, in-house clusters of cheap computers, and, more recently, networks of mobile devices. The exploitation of these networks for computing and for other purposes such

[1] The BOINC Project, `http://boinc.berkeley.edu/`.

as file sharing and content distribution has followed a different path. Whereas computing is normally performed using GRID technologies, other applications, due to legal and efficiency reasons, favored fully decentralized self-organizing approaches, that became known as P2P computing.

A P2P computer network is based on the concept of pooling computing power and bandwidth of the participants in the network, eliminating the distinction between servers and clients. In contrast to the strain imposed on servers by the transfer of large files among multiple users, this system has significative advantages. As the storage space, bandwidth, and computing power of the network's computers are pooled, the capacity of the network increases as more members join.

There are many possible classifications of P2P networks [125]. We cite here the two of them that are most widely used. The first is according to the network's degree of *centralization*. We can have:

- **Pure P2P** — peers act as equals, merging the roles of clients and server. There is no central server managing the network and no central router
- **Hybrid P2P** — a central server keeps information on peers and responds to requests for that information. Peers are responsible for hosting available resources (as the central server typically does not have them), for letting the central server know what resources they want to share, and for making their shareable resources available to peers that request them.

The second is based on how the nodes in the *overlay network* are linked to each other. We can classify the P2P networks as:

- **unstructured** — They are formed when the overlay links are established arbitrarily. Such networks can be easily constructed. If a peer wants to find a desired piece of data in the network, the query has to be spread through the network in such a way it can find as many peers as possible that share the data. If it is true that the queries may not always be resolved, since there

is no correlation between a peer and its managed content, it is also worth saying that various communication paradigms has been devised, that can efficiently tackle the "quest for the content" task (see [144] as an example).

- **structured** — They overcome the above limitations by maintaining a Distributed Hash Table (DHT) and by allowing each peer to be responsible for a specific part of the content in the network. These networks use hash functions and assign values to every piece of content and every peer in the network and then follow a global protocol in determining which peer is responsible for which content. The drawbacks are related to the overload due to caching, keyboard management and prioritization difficulties for the most popular contents.

The P2P paradigm for distributed computing can make networked applications scaling in previously unseen extents, saving efficiency and robustness as well. Though mostly known in the content distribution area (file sharing), it has been proved to be highly efficient for media streaming (see the growing phenomenon of VOIP and the worldwide thriving of applications like Skype [14], that successfully uses a fully decentralized P2P streaming protocol) and it is also exploited for scientific purposes, related to solving massive computation problems in the absence of a dedicated infrastructure (again, BOINC is a worthy example).

P2P systems presents a high dynamism, characterized by the well known phenomenon called *churn*. The term means the fact that nodes usually join and leave the system seamlessly and, most of the times, they leave in an unexpected manner, without carrying out any alerting "leaving procedure". If you add churn to the often huge scale of the existing P2P networks, it is clear that it cannot exist a node who is able to claim a global, up-to-date knowledge of the entire system. The best you can achieve in this contest is having a consistent, trustworthy distributed information shared or replicated among the nodes.

2.1.2 Epidemic Protocols

Epidemic and rumor mongering protocols are a successful example of how to deal with the high levels of unpredictability associated with P2P systems. The research on the application of epidemic protocols in distributed systems started with the seminal work of Alan Demers [43], who proposed several models for the decentralized dissemination of database updates. The models are characterized by two major features:

Anti-Entropy — Each node periodically selects a random peer and performs an *information exchange* with it. There are three way a node can do this:

- *push* — the originator of the exchange sends its own updates to the peer;
- *pull* — the originator asks for updates from the peer;
- *push-pull* — both the operations are performed.

Gossip — Whenever a node receive an update, it selects a small number k of peers and sends the update to them with probability p. Values k and p have to be tuned in order to make the update reaching all the peers, minimizing the communication overhead given by redundant messages.

One of the key issue of an epidemic protocols is therefore the selection of a random peer among all nodes. Given that a P2P network is usually wide and very dynamic, maintaining a consistent list of suitable peers is not a trivial task. To solve this issue, the concept of *peer sampling service* [71] has been devised as an epidemic protocol as well. It provides each node with a uniform random sample of the entire population of the P2P network. The framework we propose uses epidemic protocols both to create and maintain the overlay topology and to spread fresh information among solvers. We will explain in detail how the paradigm has been devised and how we implemented it in Section 3.3.

On top of the peer sampling service, several epidemic protocol can be devised to address several different problems. Besides their original goal to implement efficient and robust information dissemination, they are now used to solve membership management [71], aggregation [73], topology management [70], resource sharing, etc. While tackling all these tasks, epidemic (gossiping) protocols always keep some peculiar traits [106] that tell them apart from other message passing techniques:

- At each iteration, a peer to communicate with is selected at random;
- all nodes keep and spread only local information;
- communication is periodic;
- nodes do not assume a large transmission and processing capacity to be locally available;
- all nodes run an identical protocol.

Although presenting such a simple and not demanding set of characteristics, gossiping protocol prove to be highly robust to failures and able to empower systems with remarkable (typically logarithmic) scalability and convergence time.

2.2 Optimization algorithms and techniques

Many important optimization and decision problems are computationally intractable and are sometimes even hard to approximate. However, in many cases some of these hard problems are crucial and require a solution. Given that an efficient algorithm for these problems is (probably) out of the question, practical research is devoted to the discovery of *heuristic methods* which would optimistically be efficient over a large variety of problem instances. Furthermore, they would allow to meet the resolution delays often imposed in the industrial field.

Often these algorithms make a clever use of the objective function. Some may gather information during an iteration and then use it in the next one, others may use only local information from the current point. Regardless of these specifics, all optimization algorithms ought to be:

robust — they should perform well for all reasonable choices of the variables' initial values;

efficient — they should not require excessive computing power and storage;

accurate — they should output a solution with an acceptable precision, without being too sensitive to data errors, rounding errors, etc.

Of course these goal are always conflicting. A good design must take into account the '*best*' tradeoff among them. But even the best design turns out to be inadequate to tackle very hard problems, if not associated with a clever usage strategy and implementation.
That is why great attention has been paid for years to a certain number of execution policies, in order to increase the efficiency and the performance of the algorithms.

In most of the use cases, the optimization algorithms are stochastic by nature; in particular, the earliest stages of the search require some random decision, in order to choose good values for the first evaluation without any prior information. By changing the random seeds, different results can be obtained from distinct runs of the same algorithm. Given this, it makes sense to explore the tail of the outcome distribution by running multiple concurrent executions of an algorithm, so to reach more quickly the lower values.

Optimization heuristics have been classified in many different ways and the literature offers a large number of taxonomies. Any of them can be useful to characterize these algorithms, most of them are quite similar but never completely overlapping. Even when they are quite different, none is inconsistent

and none can claim to be clearly "better". As [17] wisely notices, smart techniques often arise from simple basic ideas whose application is both effective and easy to adapt to many different contexts. Thus different researchers may "discover" the same algorithmic ideas in different places, describing them using a (slightly) different terminology and applying them at first to (slightly) different problems. Then, trying to merge all the existent in a single taxonomy implies the choice of some denominations over others and some criteria instead of others. This process is obviously subjective, practically arbitrary and in the end there is no universal criteria to evaluate the adequacy of such theoretical classifications.

Despite of all the differences in making distinctions, everyone supports the idea of telling apart simple heuristics from more complex techniques that make use of simple heuristics in a clever way, in order to overcome some known limits or drawbacks. Here we choose to call them *meta-techniques*, just because the prefix *meta-* is by far the one with the most aged and assessed tradition when you have to talk about some "higher level" mechanism that logically or practically incorporates, makes use or somehow handles the basic ones. The problem of "where" to draw the line that tells apart heuristics and meta-techniques is an unsolved problem in literature, that gave birth to various fuzzy rules of thumb to discriminate heuristics, meta-heuristics, hyper-heuristics, racing techniques, algorithms portfolios, etc. So everyone agrees that a difference exists, but there are different ideas about what belongs to which group and if and why these groups are really distinct or not.

We are not giving here "yet another sub-optimal taxonomy", nor we are choosing one over the others. In the following of this section, we present several kinds of techniques and cite various useful publications that present their own view on this broad topic. For obvious practical reason, we have to use a given denomination to identify the algorithms. Both in apprehending the state of the art and in designing our distributed optimization framework, we tried to stick

to the famous Occam's razor principle, thus considering the minimal amount of distinctions that are necessary to correctly understand the algorithms and to provide a general and versatile implementation of most (possibly all) of them. Indeed we found the job of optimizing the optimization literature something that duly overcomes all known approximation strategies...

2.2.1 Optimization heuristics

In recent years, a large catalogue of heuristic techniques has emerged. These heuristics are said to have been mostly inspired by nature, society, physics, etc. to produce theoretical models which match the circumstances of a given problem. Heuristics have made it possible to solve cases which were impossible with conventional techniques, although in most cases the solutions achieved are only "almost optimal", having been obtained by considering characteristics which have been subjectively established by the user.

Generally speaking, optimization heuristics begin with an initial guess of the optimal values of the variables and then iterate by using different strategies, generating improved estimates until a given stopping criteria is met. Most commonly they either reach a solution that is considered acceptable or execute a stated number of iterations. One of their major advantages consists in the fact that their application does not rely on a set of strong assumptions about the optimization problem. Most of the times, to implement a good heuristic it is sufficient to be able to evaluate the objective function for a given element of the search space. It is not necessary to assume some global property of the objective function, nor is it necessary to be able to calculate derivatives.

On the other side, heuristics do not produce high-quality (or even exact) solutions with certainty, but rather stochastic approximations. However, when exact methods fail or are practically intractable, heuristics may provide satisfying approximations to the global optimum, at least when the amount of computational resources spent on a single run of the algorithm or on repeated runs is increased.

A well designed heuristic should also be robust to changes in problem characteristics, not fitting only a single problem instance, but the whole class.

Among heuristics, *local search* methods are prominent and widely used to build intelligent algorithms that search the space of the solutions avoiding traps that might be hidden in the landscapes. Jacobson and Yücesan (2004) [69] and Hoos and Stützle (2005) [66] are excellent guides while exploring these lands.

Local search methods are based on the systematic exploration of neighborhoods of successive solutions. Starting at some initial feasible solution, they are based on the idea that a solution may be improved through a sequence of one step changes, until no further improvements are possible. Accordingly, neighbor solutions are obtained from the current one by the application of an adjacency relation, i.e. a function, defined over the set S of solutions, that maps a solution into a subset of S. This concept of *neighborhood* is the central issue to all local search algorithms.

The simplest local search algorithm is *Iterative Improvement*. The neighborhood of the current solution is searched for a lower cost solution. If such a solution is found, then it will become the new current solution; otherwise, a local optimum has been found. The *Steepest Descent* algorithm is a variant of this scheme. Here a neighbor solution is chosen, that has the smallest cost within the neighborhood. *Variable Neighborhood Search* is another extension of these approaches. Here the current neighborhood is enlarged whenever a local optimum is found within it. Usually the search starts with small neighborhoods to be quick in the beginning, while the gradual neighborhood enlargement allows finding solutions that satisfy some desired termination condition.

The three methods we have just recalled are the fundamental conceptual models of most of the recent algorithms. It is useful to divide local search methods into two subsets: trajectory methods and population-based methods. Just to cite the two "halls of fame", in the first class we find threshold methods — like Simulated Annealing (SA) and Threshold Accepting (TA) — and Tabu Search

(TS), whereas prominent in the second class are Genetic Algorithms (GA), Differential Evolution (DE), Particle Swarm Optimization (PSO) and Ant Colonies Optimization (ACO).

Trajectory methods start with a tentative solution and then move in the search space, following a trajectory that leads to better function values. They are also called *single agent* methods, because only one solution per iteration is processed. They often define neighborhoods to mark temporary or permanent perimeters of the search. Finding efficient neighborhood functions that lead to high quality local optima can be challenging. All these local search methods have their own rules for the choice of a neighbor and the acceptance of a solution. They may or may not accept to momentarily worsen the current solution, in order to escape local minima. We recall here some basics of the most used methods and give some details about the Reactive Affine Shaker heuristic (RASH), a trajectory method we use in our current research (see Chapter 6).

Simulated Annealing

SA is a generic probabilistic heuristic introduced by Kirkpatrick et al. in 1983 [85], widely used to locate a good approximation to the global minimum of a given function in a large search space. The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material, that causes the size of its crystals to increase and reduces their defects. The heat makes the atoms move from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; slowly cooling down the material gives these atoms more chances to find some configurations that has lower internal energy than the initial one.

By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random neighbor solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter T (called the temperature), that is gradually decreased

during the process. Initially, T is set to a high value (or infinity), and it is decreased at each step according to some user specified annealing schedule that must end with $T = 0$ towards the end of the available time. This way, in the beginning the system wanders towards a broad region of the search space containing good solutions, ignoring small features of the energy function; then drifts towards narrower and narrower low-energy regions; finally move downhill according to the steepest descent heuristic.

It has been proved [60] that for any given finite problem, the probability that SA terminates with the global optimal solution approaches 1 as the available time is extended. However, this is often not helpful, since in many cases the time required to ensure a significant probability of success will exceed the time required for a complete search of the solution space. Among the many variants and enhancements of SA that have been proposed, we recall the Quantum Annealing algorithm [40], the thermodynamic annealing scheme devised in [41] and an interesting study by Rudolph [120] about parallel SA and its relation with population-based algorithms.

Threshold Accepting

TA was presented by Dueck and Scheuer in 1990 [45]. It is a trajectory methods that can be seen as a "deterministic variant" of SA. TA starts with a (random) feasible solution. Given a sequence of pre-defined thresholds S_t, TA always accepts a solution that improves the objective function, but deteriorations are only accepted if they are not worse than the threshold that has to be chosen in the present iteration. As for the temperature parameter in SA, the thresholds are typically ranked in a decreasing order. Over time, the threshold goes to zero, thus TA turns into a classical local search. This heuristic has been shown to have a better convergence time than SA, provided that the sequence of thresholds can be choose in an optimal way with respect to the given problem. Several studies have been dedicated to tune this algorithm against specific problems and to

achieve better general performances by combining it with a Tabu scheme [54].

Tabu Search

TS was presented by Glover in 1987 [55]. It is particularly designed for the exploration of discrete search spaces where the set of neighbor solutions is finite. The method implements the selection of the neighborhood solution in a way to avoid cycling, i.e, visiting the same solution more than once. More technically, the neighborhood of the current solution is dynamically modified by the heuristic, in such a way that the search will not end up in an already visited local optimum, at least in the short term. To achieve this, TS enhances the local search by using memory structures, among which the tabu list: each entry of this list is either a solution or a set of attributes a solution can be mapped into. Any solution that ca be matched against an entry of the tabu list cannot be visited again. Of course the list is not permanent and elder tabu moves are periodically purged (we could say they are forgotten by the algorithm).

Generally TS allows more flexible approaches to a given problem, cause the definition of tabu moves usually do not enforce a strict monotonicity of the fitness function. Moreover, the tabu list can be populated in such a way that also takes into account the current position, the recent history of the searcher, or some feature of the problem, etc. TS is largely adopted in many different fields and has a growing number of variants. We recall here he Reactive Tabu Search [16] and the TS-based reinforcement learning technique of Burke et al. [31].

Reactive Affine Shaker

The RASH heuristic has been proposed by Brunato and Battiti in 2006 [27]. It is an aggressive local optimizer devised as building block for a general local search algorithm. An adaptive random search is performed, whose scheme does not require any assumptions about the function to be optimized. The search re-

gion is adapted at each step by an affine transformation, taking into account the local knowledge derived from trial points generated with a uniform probability in the search region. This way it is possible to quickly approach local minima in the attraction basin where the initial point falls, by adapting the step size and direction to maintain heuristically the largest possible movement per function evaluation.

begin
  $\mathcal{R} \leftarrow$ small isotropic set around x ; `// x is the initial point; R is the search region`
  while *local termination condition is not met* **do**
    Pick $\Delta \in \mathbb{R}^d$ such that $x + \Delta, x - \Delta \in \mathcal{R}$; `// Δ is the current displacement`
    if $f(x + \Delta) < f(x)$ **then** `// f is the function to optimize`
      $x \leftarrow x + \Delta$;
      Extend $\mathcal{R}$ along Δ;
      Center $\mathcal{R}$ on x;
    else if $f(x - \Delta) < f(x)$ **then**
      $x \leftarrow x - \Delta$;
      Extend $\mathcal{R}$ along Δ;
      Center $\mathcal{R}$ on x;
    else
      Reduce $\mathcal{R}$ along Δ;
  return x;

Algorithm 1: The Reactive Affine Shaker algorithm

Algorithm 1 shows the pseudocode of the RASH heuristic. At each iteration, the search region is re-shaped according to what has turned out to be the most promising direction in the previous iteration. Equation 2.1 shows the affine transformation. Here d is the number of dimensions in the search space, while$(b_1 \dots b_d)$ are independent vectors which define the search region $\mathcal{R}$. The dilation parameter ρ can be heuristically set at 1.2.

$$\forall b_j \in (b_1 \dots b_d) \quad b_j \leftarrow \left(\mathbf{I} + (\rho - 1) \cdot \frac{\Delta\Delta^T}{\|\Delta\|^2} \right) \cdot b_j \tag{2.1}$$

The initial point in the trajectory is a critical choice: local optima work as

trajectory attractors during the search, thus the RASH searcher will not be able to escape from a sub-optimal solution by itself, if this solution is the best available in the current region. This is the reason why RASH is meant to be used as a part of a more complex heuristic, providing a powerful exploitation mechanism that needs to be complemented by some technique that enables an extensive exploration of the search space. Such an idea has been implemented and proposed in [29], where a RASH optimizer works within a memory-based iterated scheme that can estimate good starting point for it to be improved. As we will see in Chapter 4, we have combined the RASH heuristic with other algorithms to create novel distributed heuristics that show promising results in our preliminary experiments. □

While trajectory methods refine a single solution at each iteration, *Population-based* methods work updating a whole set of solutions (called population) at a time. They are also called *discontinuous* methods because, at least in their original version, they are allowed to "make jumps" in the whole search space. Therefore, these methods might be more efficient with regard to exploring the whole function domain, at the cost of a higher computational load and more complex structures. This is the realm of the biology-inspired techniques, as our brief list may confirm. We recall here some basic concepts about the prominent population-based heuristics, giving more details about PSO and DE, which have been used in our published works and in ongoing research, as we will see in Chapter 4.

Genetic Algorithms

GA have been introduced by Holland in 1975 [64]. New candidates for the solution are generated with a mechanism called *crossover*, which combines part of the genetic patrimony of each parent and then applies a random *mutation*. If the new individual, called child, inherits good characteristics from his parents it will have a higher probability to survive.

GA are among the most popular heuristic techniques applied in many fields. The variants and the hybridizations that have been proposed are countless. In spite of the fact that several studies have addressed the issue of analyzing the inner mechanism of this successful techniques, no exhaustive analytical explanation has been given so far about the reasons of their effectiveness. Among the open issues, the high sensitivity to diversity within the set of candidates is prominent and has lead research to devise different solutions whose effectiveness is hard to generalize [56, 116, 145].

Ant Colony Optimization

ACO has been first introduced by Colorni et al. in 1992 [38]. This heuristic imitates the way ants search for food and find their way back to their nest. First an ant explores its neighborhood randomly. As soon as a source of food is found, the ant starts to transport food to the nest leaving traces of pheromone on the ground which will guide other ants to the source. The intensity of the pheromone traces depends on the quantity and quality of the food available at the source as well as on the distance between source and nest, as for a short distance more ants will travel on the same trail in a given time interval. As the ants preferably travel along important trails their behavior is able to optimize their work. Pheromone trails evaporate and, once a source of food is exhausted, the trails will disappear and the ants will start to search for other sources.

For the heuristic, the search area of the ant corresponds to a discrete set from which the elements forming the solutions are selected, the amount of food is associated with an objective function and the pheromone trail is modeled with an adaptive memory.

Particle Swarm Optimization

PSO is a population-based heuristic optimization technique for continuous functions which has been introduced by Eberhart and Kennedy in 1995 [79]. This

nature-inspired method performs its search by iteratively updating a small number N (usually in the tens) of random "particles" (solutions), whose status information includes the current position vector $\boldsymbol{x}_i$, the current speed vector $\boldsymbol{v}_i$, the optimum point $\boldsymbol{p}_i$ and the *fitness* value $f(\boldsymbol{p}_i)$, which is the "best" solution the particle has achieved so far. Another "best" value that is tracked by the particle swarm optimizer is the global best position $\boldsymbol{g}$, in which the swarm has achieved the best fitness value obtained so far by any particle in the population.

After finding the two best values, every particle updates its velocity and positions as described by the following equations:

$$\boldsymbol{v}_i = \boldsymbol{v}_i + c_1 * rand() * (\boldsymbol{p}_i - \boldsymbol{x}_i) + c_2 * rand() * (\boldsymbol{g} - \boldsymbol{x}_i) \tag{2.2}$$

$$\boldsymbol{x}_i = \boldsymbol{x}_i + \boldsymbol{v}_i \tag{2.3}$$

In these equations, $rand()$ is a random number in the range $[0, 1]$, while c_1 and c_2 are learning factors. Usually $c_1 = c_2 = 2$. The pseudo code of the procedure is as follows:

```
begin
    foreach particle i do
        Initialize i;
    while maximum iterations or
            minimum error criteria is not attained do
        foreach particle i do
            Calculate current fitness value f(x_i);
            if f(x_i) is better than f(p_i) then
                p_i ← x_i;
        g ← bestOf(p_i), i = 1 to N;
        foreach particle i do
            Calculate velocity v_i according to equation 2.2;
            Update position x_i according to equation 2.3;
```

Algorithm 2: Basic PSO algorithm

Particle speeds on each dimension are bounded to a maximum velocity v_i^{max}, specified by the user. If the sum of "accelerations" causes the velocity on that

dimension to exceed it, then the velocity on that dimension will be limited to v_i^{max}.

The version of PSO described in Algorithm 2 assumes that all particles agree on the global best point $\boldsymbol{g}$ found so far, and is often referred to as the "classical" or "full-information" version. Effects of incomplete topologies on the performance of PSO have been studied for sociology-inspired small-world graphs [80] as well as other types of random graphs [82]. Such studies were motivated by the observation that incomplete topologies may prevent the system from concentrating too much on early-found local optima, therefore improving solution quality. Dynamic topologies based on particle clustering have also been proposed [81] to induce a differentiated behavior among groups of particles having similar local best positions. While full information has generally been shown to outperform partial topologies [104], our work focuses on a case where incomplete information is a consequence of network topology, and global data maintenance is not practical.

About PSO a large corpus of publications has grown mostly in the past decade. The performances and the behavior of the algorithm itself has been widely studied [1], also as part of more complex heuristic algorithms [13, 81, 102]. As for all the heuristics, the parameter tuning issues have been analyzed with respect of different problems and different contexts [78, 98, 142], including distributed environments [20]. What is mostly attractive in PSO heuristic is the relative simplicity of the algorithm, the possibility to obtain quite different behaviors according to different parameters' settings and the intrinsically easy way of orchestrate an extensive domain search, by a clever exploiting of the particles diffusion and convergence speed.

Differential Evolution

DE is a well-known and broadly studied method for the optimization of multi-dimensional functions that particularly suits multimodal (i.e. having more than

one minimum) functions, introduced by Storn and Price in 1997 [127].

DE is basically a scheme for generating trial parameter vectors, commonly denoted as *individuals*. We can see an individual as a vector of coordinates in the objective function domain, thus denoting a point in which the function can be evaluated. A group of individuals is called a *population*. DE generates new individuals by linearly combining two or more existing population individuals. The various different procedures of vector recombination are usually encoded in different *operators*.

Algorithm 3 outlines the basic DE when an operator using a single difference vector is chosen, that applies the recombination with a probability that is independent for each dimension. Here D is the number of dimensions (parameters) of the objective function f; $rand$ is a real number generated uniformly at random in the interval $[0 \dots 1[$; $next(j)$ gives the element that comes next to j in a vector like it was a circular list. By calling $applyOperator$ a new parameter (dimension value) is generated by linearly combining the d-th parameters of A, B and C. This happens for each dimension with a probability that is less than CR, a user given real value in $[0 \dots 1]$, or always for the latest dimension that is considered. This way, we are sure that the new individual T differs from I at least for one parameter. Finally, the variable $Best$ records the individual having the best fitness, i. e. the vector of coordinates in the function domain whose value is optimal.

Different operators realize different DE variants. Table 2.1 summarizes the most common operators. They are usually classified according to the notation proposed in [127], that is $DE/x/y/z$ where:

x specifies the vector to be mutated;

y is the number of difference vectors used;

z denotes the way the probability of performing the linear combinations is drawn.

Notation	Linear combination	Probability to be applied
DE/rand/1/bin	$T = E + Fact \cdot (A - B)$	Independent binomial experiments for each dimension
DE/rand/2/bin	$T = E + Fact \cdot (A + B - C - D)$	
DE/best/1/bin	$T = Best + Fact \cdot (A - B)$	
DE/best/2/bin	$T = Best + Fact \cdot (A + B - C - D)$	
DE/rand/1/exp	$T = E + Fact \cdot (A - B)$	Conditional probability for each dimension w.r.t. the precedent one
DE/rand/2/exp	$T = E + Fact \cdot (A + B - C - D)$	
DE/best/1/exp	$T = Best + Fact \cdot (A - B)$	
DE/best/2/exp	$T = Best + Fact \cdot (A + B - C - D)$	

Table 2.1: The most common DE operators. T is the new individual generated by the combination of other individuals. A, B, C, D, E are individuals chosen at random within the current population, while $Best$ is the individual representing the current optimal solution. $Fact \in [0 \dots 2]$ is a user defined real constant factor that controls the amplification of the differential variation.

DE algorithms try and find the optimal value of an objective function by repeatedly applying one or more operators to a given population. No global probability distribution is required to generate candidate solutions (individuals) while moving toward the optimum (being it a maximum or a minimum), thus the algorithm is completely local and self-organizing. It is easy to see that, provided a way to properly distribute the overall population, DE is a suitable candidate to distributed optimization tasks.

Distributed computing partially isolates populations from each other, allowing them to explore different portions of the search space. It has been frequently proved that distributed evolutionary algorithms can improve the results of the sequential version at the same computational effort measured in number of function evaluations. That is why there is an ever growing research about the ways the local populations can be made interact with each other in order to improve the final result. A way has to be found of designing DE algorithms so that diversity is preserved, if not enhanced, to avoid a premature convergence of the local populations to sub-optimal solutions [35].

It is due to be really careful when it comes to distribute a population among distinct nodes. A correct management of the population is crucial for DE to

begin
  Initialize a population $\mathcal{P}$;
  foreach *individual* $I \in \mathcal{P}$ **do**
    Evaluate $f(I)$;
  $Best \leftarrow J \in \mathcal{P}$ such that $f(J)$ is optimal;
  while *specific termination criteria is not met* **do**
    foreach *individual* $I \in \mathcal{P}$ **do**
      $A =$ Random pick from $\mathcal{P} \setminus \{I\}$;
      $B =$ Random pick from $\mathcal{P} \setminus \{I, A\}$;
      $C =$ Random pick from $\mathcal{P} \setminus \{I, A, B\}$;
      $d =$ Random pick from $[1 \dots D]$; `// start from a dimension chosen at random`
      for $j \leftarrow 1$ *to* D **do**
        if $rand < CR \parallel j = D$ **then**
          $T_d \leftarrow$ applyOperator(d, A, B, C); `// T is the new individual`
        else
          $T_d \leftarrow I_d$;
        $d \leftarrow next(d)$ in $[1 \dots D]$; `// round-robin scan the parameter vector`
      if $f(T)$ *is better than* $f(I)$ **then** `// f is the objective function to optimize`
        Put T in the new population $\mathcal{P}_{new}$;
        if $f(T)$ *is better than* $f(Best)$ **then**
          $Best \leftarrow T$;
      else
        Put I in the new population $\mathcal{P}_{new}$;
    $\mathcal{P} \leftarrow \mathcal{P}_{new}$;

Algorithm 3: Basic DE algorithm

succeed in approximating the optimum. The size, of course, but also the speed at which the population changes need to be carefully investigated for each specific problem, in order to achieve good results. Thus a number of dynamic or adaptive way to pro-actively tune these parameters have been proposed. This is an issue DE shares with all the evolutionary algorithms, thus we refer here to a broader literature without loss of specificity [97].

A brief but neatly written survey of various distributed techniques that have

been recently devised to serve this purpose can be found in [9]. Using population-based algorithms, computation can be parted in many different ways. The main three we may recall are:

farming evaluations of members of the local population are farmed out to other computing nodes [126];

island model in which every node run its own evolutionary algorithm, regularly exchanging information with the others [139];

diffusion the population is endowed with a spatial structure that restricts the application of the various operators [4, 133].

It is quite interesting to note the island and the diffusion models can be used from the point of view of a pure algorithmic design, even in absence of physical parallelization.

Dynamic population size adaptivity has been devised in [93] and shown to have a good impact in reducing the number of function evaluation needed for a given solution quality. Hierarchically organized approaches have obtained good results in preventing populations from prematurely converging to suboptimal solutions [11]. The main issue is that they have to be tune against a specific situation as well. Recently some adaptive mechanisms to tailor the step length of the evolution of the individuals have been devised [12]. But these ways to optimize optimization heuristics actually belong to the next section.

2.2.2 Meta-techniques

As we explained above, the allocation of the content in the various sections of this chapter is necessarily arbitrary. We could write about DE in this section instead of the previous one, considering each DE operator as a simple heuristic and the whole DE frame as a meta-technique. A quite similar consideration could be done about ACO. As it comes to GA, one could be even more

creative [117]. Considering a general framework, optimization heuristics are sometimes called *meta-heuristics* because they are considered to be a general skeleton of an algorithm applicable to a wide range of problems. Following this approach, a meta-heuristic may evolve to a particular heuristic when it is specialized to solve a particular problem.

Taillard et al. (2000) [130] and Birattari et al. (2001) [25] offer some more speculations about this. Part of the literature seems more keen to associate the so-called *unguided search* paradigm, in which memoryless methods are used to find a better solution value, with what we called simple heuristics. Thus meta-heuristics are mainly considered those algorithms who implements the *guided search* paradigm, i.e. using some memory of the past to incorporate rules and hints on where to search. For instance, in GA and DE the population represents the memory of the recent search experience; in ACO the pheromone matrix represents an adaptive memory of previously visited solutions; in TS the tabu list provides a short term working memory.

Following the most common terminology, meta-heuristics are made up by different components and if components from different meta-heuristics are assembled we obtain a hybrid meta-heuristic. From this point on, it is easy to imagine how huge is the number of novel algorithms that have already been proposed and how many are just yet to come. The construction of hybrid meta-heuristics is motivated by the need to achieve a good tradeoff between the capabilities of a heuristic to explore the search space and the possibility to exploit the experience accumulated during the search. We just mention here that the studies about restart policies [51, 99] can be also considered as examples of meta-techniques applied to basic heuristics.

The question of combining multiple heuristics has been addressed many times by various disciplines interested in solving hard optimization problems. In general, finding the optimal combination has been shown to be NP-hard even to approximate [122]. To better explain why this topic is so important, we

briefly re-formulate the problem from a different perspective.

Many computational problems that arise in the world are NP-hard, and thus likely to be intractable from a worst-case point of view. However, the particular instances of these problems that are actually encountered can often be solved effectively using heuristics that do not have good worst-case guarantees. As we have seen, there are a number of heuristics available for solving any particular NP-hard problem. Unfortunately, there is no heuristic that performs best on all problem instances [141]. Thus, when solving a particular instance of an NP-hard problem, it is not clear a priori how to best make use of the available CPU time.

Algorithm portfolios

In literature is common to refer to the general problem of determining how to solve a problem instance in this setting as *algorithm portfolio* design [67]. An algorithm portfolio may consist of a single stochastic algorithm that is periodically restarted (possibly with a different parametrization), following an opportune pre-computed strategy. Or it may be the composition of different algorithms that alternate according to a given schedule. As Streeter et al. [128] point out well, the problem has both a machine learning aspect and a scheduling aspect: one has to predict which heuristic will solve the instance first and also to determine how long it is worth to run a heuristic before trying a different one.

One of the earliest works on the problem of computing an optimal schedule is [77]. Here the goodness of a schedule is evaluated in terms of its competitive ratio, i.e. the time required to solve a given problem instance using the schedule, divided by the time required by the optimal schedule for that instance. More recently, Gomes et al. [59] showed how to improve state-of-the-art heuristics for Boolean satisfiability and constraint satisfaction by randomizing the heuristic's decision-making heuristics and running the randomized heuristic with an appropriate restart.

Another approach to algorithm portfolio design is to use features of instances to predict which algorithm will be the fastest on another instance, and then simply allocate all the CPU to that algorithm. Moreover, existing heuristics can be successfully combined into a new and faster heuristic by collecting some tens of training instances, that are used to compute a schedule for the interleaving execution of the existing heuristics. Exploiting instance-specific features it is possible to generate an optimal schedule for a particular problem instance.

This view, often called "off line portfolio composition" does not address all possible aspects of the algorithm portfolio design problem. It is possible to make scheduling decisions dynamically based on the observed behavior of the heuristics. Basically this means that one attempts to predict a heuristic's remaining running time based on its current state and adapt the schedule accordingly. For example, Gagliolo and Schmidhuber [50] presented an approach for allocating CPU time among heuristics in an online setting, based on statistical models of the behavior of the heuristics, although their approach has no rigorous performance guarantees. Also, in [17] the authors describe advanced reactive strategies to tune some of the portfolio parameters based on fresh online information.

As soon as we start the topic of choosing on the fly the best algorithm among a pool of running instances, we meet other broadly used categories of meta-techniques: racing strategies and hyper-heuristics. As usual, it is practically impossible to draw a clear line between the two. It is also arguable that several portfolio schemes could be called hyper-heuristics and viceversa.

Racing

A *racing* strategy is a class of methods by which one evaluates running algorithms like they were competing against each other, in order to assign most of the resources to the one who shows the most winning attitude toward the current problem [100]. Instead of learning offline which algorithm is potentially

the best in solving the optimization task, an online evaluation based on user defined criteria is performed while the optimization task is running. Depending on how each competing algorithm is scoring, the allocation of resources is dynamically modified in such a way to be sure the best performing algorithm will be given, for instance, the larger amount of CPU time to perform the task.

The racing strategy can be also used offline, as a learning tool to find out the best configuration for an algorithm. This way the race between the competitors is not the main optimization task, but is part of a prodromic parameter tuning phase, that lead to discover the most promising heuristic for solving a given problem. In [17] several examples are detailed.

Hyper-heuristics

The notion of *hyper-heuristics* (HH) overlaps the one of racing strategies, including them as a subset. According to a very recent classification [33] they can be defined as search or learning mechanisms whose aim is to generate and/or select heuristics to solve a given problem.

HH are high level problem independent heuristics that work with any set of problem dependent heuristics and adaptively apply and combine them to solve a specific problem [32]. They are similar to meta-heuristics; the difference is that meta-heuristics are not off-the-shelf methods that can be readily applied to any problem; they are schemes that have to be instantiated and tuned to specific problems. As opposed to this, HH do work off-the-shelf using any given set of operators and algorithms. The tradeoff is that HH are "good enough, soon enough, cheap enough" [31] approaches while meta-heuristics can achieve better performance although require significantly more investment.

Although it is a promising and useful idea to design and apply parallel HH, relatively little work has been done in this area, compared to the significant body of work on parallel meta-heuristics [5]. In [115], a master-slave model is proposed, along with a more distributed model where there are many clusters

that implement a master-slave model locally. In [94] a hybrid island model is applied in which a central coordinator decides which algorithm has to be applied by several interconnected islands, according to the feedback they periodically provide. In [111] and agent-based approach is proposed that is nevertheless also conceptually centralized involving a single HH agent. Finally, in [135] a Grid-based solution is proposed with a central HH server and slave nodes performing low-level search.

All these works can be considered as a successful effort towards the mitigation of the drawbacks of both a pure centralized and a pure independent approach, by means of the combination of the two. All of them assign the fundamental task of performing the evaluation of a pool of algorithms to a single component, thus falling, practically or conceptually, in a master-slave approach. We believe that emerging platforms such as cloud computing [61], as well as the more established peer-to-peer [10] and Grid [76] platforms all favor a coarse grained, decentralized approach that has no bottlenecks and that scales well and tolerates failure and dynamism. Our goal is to target such platforms.

2.2.3 P2P optimization algorithms

The field of distributed optimization has been widely investigated for years either as a specific case of distributed computing, or as a specific function optimization technique. However, most of the existing research in this domain assumes a properly arranged architecture, like a parallel computing provision or clusters of networked machines, usually handled by a central coordinator. These kind of systems either have strict synchronization requirements or rely completely on a central server, which coordinates the work of clients and acts as a status repository [131].

Whereas large scale parallel computing is normally performed using GRID technologies, it is an emerging area of research to apply peer-to-peer algorithms in distributed global optimization. P2P algorithms can replace some of the cen-

tralized mechanisms of GRIDs that include monitoring and control functions. For example, network nodes can distribute information via "gossiping" with each other and they can collectively compute aggregates of distributed data (average, variance, count, etc) to be used to guide the search process [83]. This in turn increases robustness and communication efficiency, allows for a more fine-grained control over the parallel optimization process, and makes it possible to utilize large-scale resources without a full GRID control layer and without reliable central servers.

P2P heuristic optimization is quite a newborn branch of distributed optimization. As it happens in all the happy beginnings, researchers are extensively exploring the various issues this new field entails. We give here a brief overview of the recent publications, mostly related to P2P implementations of population-based algorithms.

In [103] the authors presented some preliminary evaluations of a parallel hybrid MO-EA (multi-objective evolutionary algorithm) deployed in a P2P environment. Several results show that is possible to successfully parallelize such an evolutionary algorithm in a P2P fashion, exploiting the resources available in a network of 120 heterogeneous PCs.

In [63] an extensive experimentation was performed, to test the fault tolerance of the island model on GA, when executing them on a distributed system. The results show that this model can be trusted when running experiments on a non-reliable parallel or distributed infrastructure, the quality of the outcomes being at most 2% worse (on average) than when a reliable infrastructure is employed, without adding any special techniques required for dealing with faults.

Battiti et al. [28] presented an implementation of PSO in a distributed peer-to-peer environments. Global information sharing among processes is managed by epidemic protocols, that ensure spreading of relevant data generated during the search. The results show that the message complexity needed to outperform a single-node sequential algorithm can be low and that the proposed approach

is therefore viable.

Almost the same PSO distributed algorithm has been proposed by different authors in [123] to address multi-objective optimization problems, in which the main goal is to find a set of values lying on the Pareto-frontier that can be find for the objectives being tackled. In spite of the different experimental environment (simulated very large networks in the previous case, small real network in this case) results confirm the usefulness of the approach.

Recently, the same authors [124] proposed new models to substitute failing particles of the same kind of multi-objective PSO algorithm, in such a way that the capability of the affected swarm to explore the search space can be enhanced. The initialization of the new particles is performed using a combination of binary search to fill the gaps in the space between the two known Pareto-front extremes and edge extension, to improve the exploration beyond the known Pareto-front.

Van Steen et al. [140] presented a fully decentralized evolutionary algorithm in which the population size is kept stable by locally adapting the surviving rate of the individuals according to global population estimations, performed by means of gossiping protocols. The parent and survivor selection can be done completely autonomously and asynchronously, without central control, yet avoiding the risk of population explosion or implosion.

In [89] the authors used the same algorithm to tackle hard problems whose computational time increase exponentially as they scale. Moreover, they compared their results with a sequential GA algorithm, outperforming this competitor thanks to the gossiping adaptive mechanism used to control the population.

Laredo et al. [90] proposed the Gossiping-based Evolvable Agent model (actually already devised in [88]), where every individual of an evolutionary algorithm's population self-schedules its own action as an agent (evolvable agent) and dynamically self-organizes its neighborhood via newscast (gossip-based). Tests run in a really distributed deployment with multi-threaded configurations

confirm that P2P evolutionary algorithms are competitive with respect to not-distributed ones.

Finally, in [91] the authors presented an extensive evaluation of a generic P2P evolutionary algorithm with respect of different (simulated) network size and churning conditions, while tackling a known hard test function. Results show that the gossiping enabled small-world structure of the network makes the algorithm very robust even when very high churn rates are applied.

As a whole, existing literature about P2P heuristic optimization confirms that not only this approach to distributed optimization is viable, but moreover that can suit hard problems, provided that an adequate number of resources are available. The availability of many cheap and interconnected machines is an easy goal to obtain in many contexts nowadays and this new kind of algorithms show the capability to fruitfully exploit such a computational environment. As we detail in Chapter 6, our research fully confirm this conviction.

2.2.4 Optimization frameworks

To effectively produce a not trivial good, one needs to use appropriate tools that can help handling the small and basic issues, letting the user concentrate on the core task: to design, deploy, test and finally evaluate the product. Optimization tasks are not different. To improve the efficiency and the effectiveness of the scientists' job several tools have been devised, whose general goal is to provide a set of facilities that can ease the optimization algorithm "production" chore. Such tools are commonly named *optimization frameworks*. Actually the term is not really univocal, because it is use in literature to name

- sets of conceptual engineering guidelines to algorithm specification;
- sets of tools for algorithm design and implementations;
- sets of facilities to deploy and run distributed algorithms.

Although clearly oriented towards the second semantics, we are interested in aspects that apply to all of these tools.

Among the existing frameworks, several address the same issues and have the same goals. Some of them are more renowned, most claim to be better than the others with respect of some feature. Talking about "successful" or "unsuccessful" frameworks is actually inappropriate, because it often happens that the most effective framework, according to one's mind, is the framework whose usage one knows best. In the following we briefly describe the most recent and widely known optimization frameworks, characterizing them according to some objective features, without entering any discussion about practical evaluation. The expertise of the user and the correct choice of the most appropriate tool for the peculiarities of the task being tackled are the two key points that make one's personal judgement in the everyday practice.

Most of the frameworks that have been proposed in the recent years are focused on population-based heuristics. There is no general agreement about the fact that designing and tuning these kind of algorithms needs more guidelines and facilities than developing trajectory methods. It can be the case that evolutionary algorithms are simply more appealing at the present time. For sure population-based heuristics have given birth to an amount of variants, hybridizations, cooperative versions etc. that patently outnumbers the counterparts. Thus the need of some means to have a unified and clean understanding of them all, characterizing and analyzing the main common aspects.

One of the key properties of the frameworks is to make a clear conceptual distinction between the solution methods and the problems to be solved, in order to neatly separate the design and the implementation of these two components [110]. They provides sets of methods or functions of common use, in order to make the user do the smallest amount of work from scratch. Usually frameworks value the reusability of the components. On the one hand they try and maximize the possibility to efficiently use the same tool in different con-

texts; on the other hand they often aims at being easily extendible. Thus a user should be able to add features and components with little or no effort. For those that provide distributed deployment facilities, a further key point is how transparently the parallelization of the algorithms is handled and how reliable the frameworks prove to be in this settings. In this case, it is also important that a framework supports different architecture and operative systems.

The first remarkable distributed computing facility we recall is the *BOINC* project [7]. It is a set of distributed abstractions for applications, data and files that implements a client-server paradigm, allowing users to set up and configure their applications by exchanging XML description files with the central server. Computations are performed in a highly distributed way, generating several sets of results. Results are replicated and generally a consensus has to be achieved to determine their validity. Any machine can join the system and make a given set of resources (cpu, memory, bandwidth) available to users who need them. Participation is stimulated by means of a credit mechanism, that accords a larger access to the resources to the more generous contributors. Cheating detection schemes and automated resource probing prevent fraudulent user behavior. This system is not optimization-oriented and we believe it would require a great effort to be effectively used for this purpose. Moreover, although distributing the applications among a large number of peer computational units, it ultimately relies on a client-server design that cannot be considered as a fully decentralized approach. Nevertheless, its extreme configurability and the potentially huge dimension of the distributed facilities are undisputed.

Another quite renowned platform to develop and deploy distributed computational tasks is JADE (Java Agent Development Framework). [2] The goal of JADE is to simplify the development of multi-agent systems through a large set of system services and agents: naming service, yellow-page service, message transport and parsing service and a library of standardized interaction proto-

[2]The Java Agent Development Framework, `http://jade.cselt.it`.

cols ready to be used. All agent communication is performed through message passing, where ACL is the language to represent messages. The agent platform can be distributed on several hosts. The JVM executed on each host contains agents and provides a complete run time environment for their execution, also allowing several agents to concurrently execute on the same host. Agents are implemented as one thread per agent, but whenever they need to execute parallel tasks, JADE supports scheduling of cooperative behaviors, besides the usual multi-threaded execution environment that Java technology provides. A graphical user interface (GUI) for the remote management is provided, that allow users to monitor and control the agents. JADE is actively supported and ever growing. Not optimization oriented, it is a reliable and robust hybrid p2p networking environment to run distributed application that ease the task of deploying cooperative agents.

Coming to something more specific and focused on the optimization algorithms and software production, *EasyLocal++* [53] was proposed as a set of object-oriented tools to simplify the design and the development of local search optimizers. The set of classes and facilities it provides forces the user to think about a local search (meta-)heuristic as a set of interacting objects that serve each other during the computation. It also provides a textual output interface and it can produce simple plots. We think the conceptual model is good, though it lacks flexibility and needs some extension to fit different kind of algorithms. In our opinion, it is anyway not trivial and quite time consuming to get acquainted with both the set of classes to use while developing an algorithm and the script-based configuration module required to run experiments. More recently, a quite similar approach has brought different authors to present the *MOEAT* [121] framework, written in C#.

Moving within the number of frameworks devoted to analysis and design of algorithms, *Magma* [118] offers an interesting example of multi-agent architecture for conceptual design and practical implementation of meta-heuristics. It

proposes a multi-level description of the meta-heuristics as a set of agents that create, locally improve, exploratively search and dynamically interact to find the optimum. This concept translates directly in a set of methods and interfaces that guide the implementation of the algorithms. Besides providing a good set of already implemented heuristics, in our opinion the value of this framework consists in the devised classification of heuristic agents based on their goals, that effectively makes it possible to implement a wide range of meta-heuristics in a layered fashion. It does not provide any facility for the distributed deployment of the algorithms, nor for their mutual communication.

Very few frameworks are natively suited for distributed environments. One prominent example is ParadisEO [34]. It is an object-oriented framework dedicated to the reusable design of parallel and distributed meta-heuristics. It embeds the implementations of widely used evolutionary algorithms and local searches algorithms, along with the most common parallel and distributed models and hybridization mechanisms. Their implementation is portable on distributed-memory machines as well as on shared-memory multiprocessors. The provided software architecture allows a good level of code re-usability, achieved anyway by means of a quite complex hierarchy of classes and templates the user has to master. Although not P2P-oriented, this framework covers a large range of optimization techniques, enabling their active cooperation in a distributed fashion. A similar approach, but limited to evolutionary algorithms and to their master/slave coordinated execution has guided the design of DistributedBeagle [52].

The Java Distributed Evolutionary Algorithms Library (JDEAL) [39] is an example of Java-based optimization framework that has lately evolved towards the distributed dimension. It uses JDS (Java Distribution Service) over Voyager [3] to perform parallel distributed computations. The various flavors of genetic algorithms available in the framework are coordinated through a master-

[3]The Voyager P2P network, `http://www.recursionsw.com/Products/voyager.html`.

slave configuration that enables effective operativity. Instances of algorithms can be dynamically added and removed at runtime, tolerating the most common types of failures. The amount of configuration required to set up a working instance is reasonably low and legacy code is smoothly adapted by extending appropriate framework components. This framework presents many remarkable features. However, its distributed computational model, though solid, is quite different from the one we target.

It is worth citing in this context also ECJ (Java-based Evolutionary Computation Research System) [4], even if its design goals are rather different from the other framework we examine. It makes available a set of evolutionary algorithms and genetic programming techniques mainly designed for centralized execution. A distributed execution environment is also provided, that is based on the island model, whose communications are implemented by means of Java TCP/IP sockets. This architecture ensures limited scalability. The peculiarity of this framework consists in the way it interacts with the users. In a normal execution, either parameter files or a very basic graphic interface should be use to set up the initial configuration of the experiment and make it run.

MALLBA [3, 6] is a software tool for the resolution of combinatorial optimization problems using generic algorithmic skeletons implemented in C++. Skeleton classes implement the generic core functionalities of various optimization methods and devise three different implementations for any of them: sequential, parallel for Local Area Networks and parallel for Wide Area Networks. The framework integrate several optimization techniques together and offers the possibility of both sequential and parallel environments to coexist transparently. The skeleton design is based on the separation between the features of the specific problem to be solved and the general resolution technique to be used. The user must fill in the required classes with an specific problem-dependent implementation, whereas the optimization and parallelization tech-

[4] ECJ, `http://cs.gmu.edu/~eclab/projects/ecj/`.

nique is completely provided by the library. Communications handling is done via a light middleware layer that simplifies their tuning and provides easy access to message passing, broadcasting and coordination facilities. While we think this framework is a fine piece of engineering, we could not find any explicit information about the differences between the WAN parallel execution of the algorithms and the LAN parallelization technique (that differs according to the algorithm, but always demand a faultless stable networking environment).

A remarkable framework that addresses both the issues of assisting the user in the algorithm design and helping the distributed execution is DREAM (Distributed Resource Evolutionary Algorithm Machine) [10]. This peer-to-peer system is based on the island model, implemented through epidemic protocols. DREAM is targeted toward Wide Area Networks (WAN), where communications rely on internet standard protocols. Moreover, various kind of user interfaces are provided to meet the needs of users with different skills, interests or commitment. The range of issues and the target that this framework aims to address are larger and slightly different from ours. DREAM is devoted to evolutionary algorithms, while we propose a framework which can suit any optimization algorithm. Of course this means that the external interface are much less specialized, because they *must not* demand the user algorithms to comply with very specific constraints or design patterns. Then our tool is primarily meant to be used by those scientists who want to easily and quickly deploy and test their algorithms in a P2P distributed fashion. DREAM has been devised as a platform that can run virtually any agent-based distributed application and offers several different user interfaces, to suit the most different kinds of users.

An interesting approach to address the issue of solvers P2P distribution has been recently proposed in [21]. G2DGA is a framework implemented as a hybrid P2P overlay with two types of objects, (i) islands that run a GA process, and (ii) a supervisor that perform monitoring and adaptation. The supervisor creates the island objects and defines a migration policy that is sent to each of

them, specifying the migration interval, rate, and a list of neighboring islands. The islands run the GA and handle migration, which is asynchronous. The migrants are sent directly between the peers (islands), while the supervisor collects feedback data from the islands and is responsible for adaptation. By its hybrid design G2DGA can retain a global overview of the GA which provides opportunities for adaptation that are difficult to achieve in a pure P2P architecture, while avoiding the bottle-neck and single point of failure problems associated with traditional client-server solutions since; (i) the supervisor is implemented as a P2P node. If the computer goes off-line or becomes busy, the P2P load-balancer will transparently move the supervisor object to another computer, and (ii) the supervisor is optional, and only used to improve the performance of the GA. The islands will continue to work without it. G2DGA is based on G2P2P, which is a P2P distributed object framework based on .NET remoting and XML encoded message passing communications.

Finally, a very recently born-again java-based framework that presents interesting characteristics is EvA2 [129]. The project is currently in active development, but already capable of offering implementations of various heuristics, a graphic user interface with plotting facilities and seamless integration of MatLab code. This framework shares several designing goals with ours (for instance the target users and the main usability purposes, that we describe in Chapter 3) and demonstrate a quite more mature development state. Anyway, it does not provide any mechanism for a distributed deployment of the algorithms, nor addresses the problem of communications in a network of solvers.

The framework we propose addresses the issues of the interchangeability of the optimization components and focuses on the specificity of a P2P environment. Making use of the epidemic paradigm to enable solver-to-solver communication, our simple yet versatile architecture can ease the effort of porting and executing optimization tasks in a fully decentralized network of solvers.

Chapter 3

GOOF framework architecture

GOOF aims at bridging the gap between the distributed system and the function optimization fields. This means that GOOF has mainly been conceived to meet these two objectives: (i) to couple a general optimization framework to a P2P middleware that can handle different P2P network topologies and (ii) to make it easy for function optimization practitioners to plug legacy or novel optimization code in such an "unfamiliar" environment.

GOOF is based on the P2P paradigm, where a large collection of computing nodes cooperate in a decentralized way toward a common goal. Our algorithmic approach is based on the *epidemic* paradigm, a very light-weight approach to distributed computing. Gossip protocols have proven to be able to deal with the high levels of unpredictability associated with P2P systems.

GOOF consists of an epidemic protocol with a set of interfaces and services, written in Java, that come in two flavors:

- as a framework ready to be plugged in PEERSIM [74], a widely used P2P network simulator specialized in the simulation of gossip-based protocols, and
- as a framework ready to be plugged in CLOUDWARE [105], that is a real-world implementation of some of the main PEERSIM concepts for real deployments in large-scale networks.

An important advantage of this framework organization is that the user can execute optimization tasks both in a simulated and in a real P2P environment without changing a single line of code in the actual implementation of the algorithms and/or of the functions already deployed.

Exploiting the capabilities of the existing applications it relies on, GOOF provides seamless interaction with the P2P environment and a clean and easy-to-get set of interfaces to quickly plug algorithms to be executed and problems to be solved.

This chapter describes the architecture of our optimization framework. We first present the assumptions over the underlying system model; we then present the generic architecture, briefly introducing the main modules, whose characteristics will be explained in detail in the next chapters.

3.1 System Model

We consider a network consisting of a large collection of *nodes*. The network is highly dynamic; new nodes may join at any time, and existing nodes may leave, either voluntarily or by *crashing*. Since voluntary leaves may be simply managed through "logout" protocols, in the following we consider only node crashes. Byzantine failures, with nodes behaving arbitrarily, are excluded from the present discussion.

We assume nodes are connected through an existing routed network, such as the Internet, where every node can potentially communicate with every other node. To actually communicate with another node, however, a node must know its *identifier*, e.g. a pair ⟨IP address, port⟩.

The nodes known to a node are called its *neighbors*, and as a set are called its *view*. Together, the views of all nodes define the topology of the overlay network. Given the large scale and the dynamism of our envisioned system, views are typically limited to small subsets of the entire network. Views can

change dynamically, as well as the overlay topology.

Our parallelization approach is based on a symmetric island model: we assume that we are given independent nodes, each of which running the same algorithm, periodically communicating with each other. From now on we use the words "node" and "island" interchangeably.

Our target networking environment consists of independent nodes that are connected via an error-free message passing service: each node can pass a message to any target node, provided the address of the target node is known. We assume that node failures are possible. Nodes can leave and new nodes can join the network at any time as well.

3.2 Framework Design and Architecture

Taking into account the manifold issues concerning the creation and maintenance of a P2P network on the one side, the consistency and performance requirements of an optimization task one the other side, we devised a three layer architecture of independent modules:

- *The topology service* is responsible for creating and maintaining an adequate overlay topology to be used by the other layers to communicate information about the search space.

- *The function optimization service* evaluates the target function over a set of points in the search space, opportunely selected based on both local information (provided by this module, based on past history) and remote information (provided by the communication service).

- *The communication service*, as the name implies, enables node to communicate, with the goal of exchanging information about the current best solution and the space that has been explored, coordinating their actions in future runs of the function optimization service.

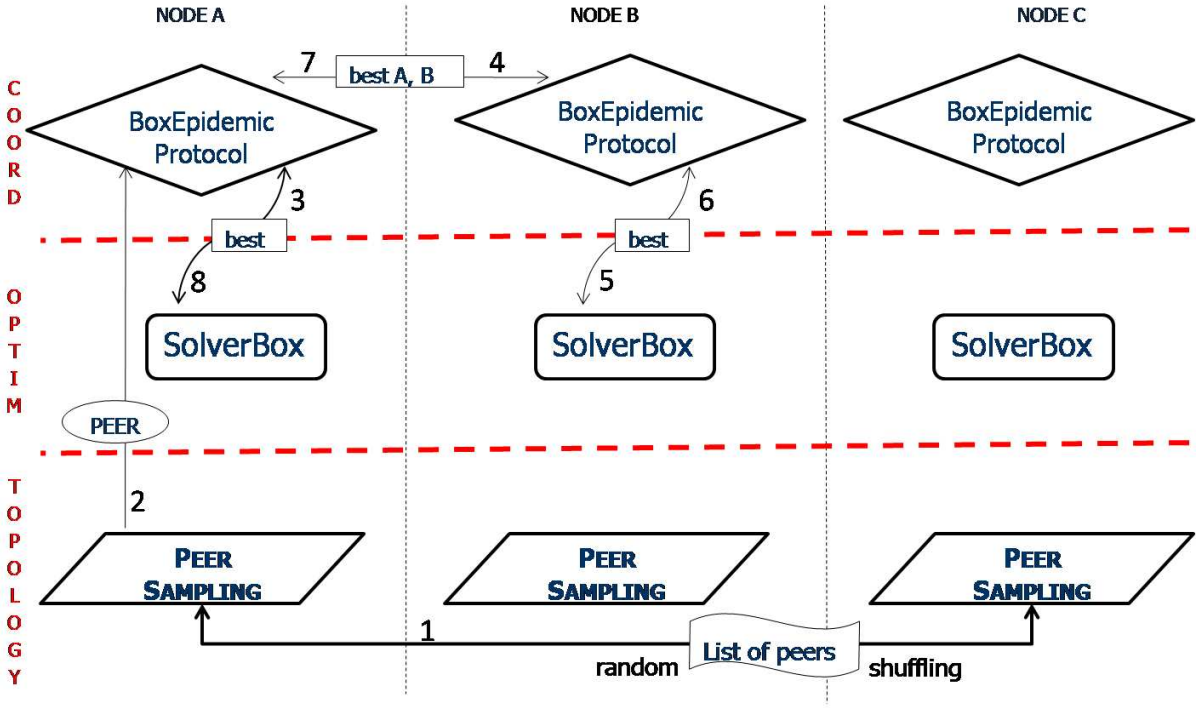


Figure 3.1: High level architectural view and communication steps

The three modules cooperate in order to perform the optimization task and spread useful information among peer nodes. In Chapter 4 we will discuss in details the Function Optimization Service. Figure 3.1 illustrates the three layers architecture and the interaction between the services. The numbers visible in this figure keep track of the steps that the system carry out whenever the optimization service decides to communicate result to another peer layer running in a different node. The whole procedure is quite intuitive and plainly realizes the epidemic paradigm. More details about this are given in Chapter 5.

3.3 The Topology Service: Peer Sampling

Overlay topology maintenance is obviously fundamental to keep the network of solvers connected, thus making it possible for the nodes to share information fruitfully in a reliable way. As examples, consider a random topology used by

a gossip protocol to diffuse information about global optima; a mesh topology connecting nodes responsible for different partitions of the search space; but also a star-shaped topology used in a master-slave approach.

In GOOF, the topology service is provided by the middleware projects that GOOF is based upon, PEERSIM and CLOUDWARE. They provide (either in simulation or in a real deployment) basic functionalities like a selection of robust topologies, bootstrap mechanisms to let node join and leave, and a few basic transport layers. Concerning the overlay network topology, the current implementation of GOOF relies on the NEWSCAST protocol [75], which has proven to be a valuable building block to implement several P2P protocols [73]. We provide here a brief description of the protocol and its characteristics.

Each NEWSCAST node maintains a view containing c node descriptors, each of them composed of a remote node identifier and a logical time-stamp. NEWSCAST is based on the epidemic paradigm: periodically, each node (i) selects a random peer from its partial view; (ii) updates its local descriptor and (iii) performs a *view exchange* with the selected peer, during which the two nodes send each other their views, merge them, and keep the c freshest descriptors.

This exchange mechanism has four effects: views are continuously shuffled, creating a topology that is close to a random graph with out-degree c; views can be considered a random sample of the entire network (hence the name "peer sampling"); the resulting topology is strongly connected (according to experimental results, choosing $c = 20$ is already sufficient for very stable and robust connectivity); and finally, the overlay topology is self-repairing, since crashed nodes cannot inject new descriptors any more, so their information quickly disappears from the system.

No special provisions are taken to deal with failures. Messages initiated by random peers can eventually be lost, with the only effect of slowing down the spreading of information. Nodes may be subject to churn without affect the consistency of the overall computation, thanks to the robustness provided

by the chosen topology service (NEWSCAST). Nodes suddenly leaving cause no harm to the overall computation. Nodes that join the network simply start with the common initial configuration and then get updated as soon as they start receiving gossiping messages.

The neighborhood structure at each node is random. More precisely, we assume that at any point in time all nodes can request a random node address from the local *peer sampling service* that returns a random sample from the entire network. Note that in this framework it would also be possible to use gossip algorithms to generate better neighborhood structures [36, 133]. Based on our tests and acquired experience, we felt no need so far to implement other structured topologies, though obviously it may be an easy additional feature to be added in the future.

While considering the possibility of function optimization on top of structured networks, a key point is the fact that distributed function optimization is quite a different task from content distribution and retrieval. As we briefly mentioned in Section 2.1.1, the various advantages of structured P2P network over unstructured ones are mainly related to the fair distribution and the quick retrieval issues. Of course these features are desirable in any task, thus in distributed optimization as well, but according to us they are in this case less crucial. As already said, P2P optimization moves its steps from the parallel concurrent optimization paradigm and aims at offering a more scalable and robust way of achieving a comparable quality of solution. In a massive parallel computational system it is usually not important to track which computational unit provided a given updated to the shared memory, nor it is relevant to know where a value has come from in order to be able to fetch it from the common knowledge base. The only mandatory requirement is that units should be able to access fresh information as quick as possible, whenever they need it. Analogous situation we have in a fully decentralized system.

Of course in distributed optimization tasks a fairly spread workload and a re-

liable access to the most recent information available are very important design goals. The advantages of a structured P2P overlay can be effectively exploited to improve the performance of an optimization task: for instance, portions of the search domain can be assigned to nodes quite easily, as well as a proportional number of function evaluations. But here we have to pay attention to the possible costs of these benefits. In a small networks of optimizers, where the networking infrastructure is not only very reliable, but also quite easy to manage and control (we can take a LAN as a typical example) this scenario is quite worth trying.

But if we plan — as we do — to tackle large scale networks of (possibly sparse) machines connected by the existing internet infrastructure, some serious issues arise while relying on structured topologies. It is not only about scaling problems or about connectivity troubles in presence of churn, but also about the fact that the more our distributed algorithm design exploits the overlay structure, the more it is exposed to the risks that this structure entails. If for instance we split the search space in distinct regions and assign to each node a different region, connection faults or churn will seriously tamper the quality of our final solution. We briefly mention some effect of churn in such an algorithmic design in Section 4.3. Then if we think to improve this design by providing some redundancy, i. e. by splitting the search domain in overlapping regions, we have to face the fact that typically we don't know the search region well enough to be able to decide a priori a good way of parting the domain, to avoid redundance of uninteresting regions (that translates to wasting computing power and time) while minimizing the risk to loose important portions of the search space.

One more final consideration concerns the peculiarity of distributed optimization. One of the most challenging goals for distributed algorithm designers is to enable cooperation among the various solvers. Enabling networked optimizers collaboration is something different than distributing evenly the tasks among them. While the latter goal may be surely helpful, the main require-

ments to achieve an effective cooperation are more related to the possibility to spread information in a fast and efficient way, while keeping the system robust to failures or defections. With respect to this, we believe that unstructured P2P networks offer an interesting working environment, whose issues can be effectively tackled by a proper use of epidemic protocols.

All these considerations are personal and we do not claim any proof that can definitely support them. Few works that have partially addressed the aforementioned issues [23, 119] seem to confirm our thoughts, but as far as we know the whole topic has not yet been addressed thoroughly enough to draw any clear conclusion.

3.4 Some considerations about P2P optimization algorithms

We started our research investigating when and how is possible to effectively implement a decentralized distributed optimization task. In our published papers we designed and tested several distributed optimization algorithms and hyper-heuristics that fully exploit the epidemic mechanism to cooperate in the same task. We showed that by using this architecture is not only possible, but also effective to achieve results that are often better than those you can achieve by means of a centralized or tightly coupled architectural design. We also found some interesting insights about how the implemented heuristics work in such a distributed environment, that would have been quite hard to obtain in a traditional framework.

Two opposite techniques could be followed to design a distributed optimization algorithm:

- **Without coordination: exploiting stochasticity** — Global optimization algorithms are stochastic by nature; in particular, the first evaluation is not driven by prior information, so the earliest stages of the search require some random decision. Different runs of the same algorithm can evolve

in a very different way, so that *parallel independent execution* of identical algorithms with different random seeds yields a better expected outcome w.r.t. a single execution.

- **With coordination: exploiting communication** — Some optimization algorithms can be modeled as parallel processes sitting in a multi-processor machine supporting shared data structures. Processes can be coordinated in such a way that every single step of each process (i.e., decision on the next point to evaluate) is performed while taking into account information about all processes. In order for such approach to be efficient, the cost of sharing global information should not overcome the advantage of having many function evaluations performed simultaneously.

In between these two extremal cases (no coordination at all or complete information), it is possible to imagine a wide spectrum of algorithms that perform individual searches with some form of loose coordination. P2P overlay networks provide the right environment in which a loose but reliable coordination can be used to orchestrate independent cooperating optimization tasks. If the global optimization task is properly designed, individual successes are beneficial for the whole network, while single failures are mostly harmless.

In order to achieve such a nice hallmark, it is essential that the designer pays attention to some novel and interesting issues.

– *Synchronization*. Depending on the kind of algorithms and on the way their computations are distributed among the various running instances, there may be performance decays due to synchronization needs. On the other hand, not all the algorithms have strict synchronization requirements. For those which have them, it is likewise fundamental to know *when* the synchronization is strictly needful and when it is not. Thus, it is important to achieve a good understanding of the relation between performance drifts and ratio of communication events, in order to find, for each kind of algo-

rithm, the maximal amount of synchronization occurrences that does not penalize performances.

– *Information sharing*. How much information has to be shared among the distributed instances of an algorithm? Of course, it depends on what algorithm is running, but also on how many of its instances are active in the network at the same time. A careful consideration of *what* has to be shared and what can be kept as local is crucial for each kind of algorithm.

– *Convergence and communication rate*. *How often* shared information must be updated? The spreading of the information relies on both the frequency of the message exchanging and the interconnection topology. The behavior of the same algorithms in different kind of P2P topologies may present significant differences. A tradeoff between a good information exchange and a rapid advancement of each individual search process has to be found. Moreover, in a totally decentralized system is due to pay attention to the effects of the delays in propagating the updates.

The framework we provide try and address these issues by giving the user the possibility to decide how much information the nodes share, how often they spread and receive updates and which way these updates are actually used by the local solver. This flexibility has been achieved by simple and effective design choices that do not put any particular workload on the user, while providing a quick and effective way to deploy distributed optimization tasks.

Chapter 4

The Function Optimization Service

The function optimization service contains two core components: a collection of *entity interfaces*, that must be implemented by the actors of an optimization task, and the *SolverBox*, whose goal is to orchestrate the execution of these actors. In the rest of this section, we present these components in detail, starting from the main characters acting in the optimization play.

4.1 The entity interfaces

Recalling the description of a generic optimization process provided in Chapter 2, the two main characters in this play are the function to be optimized and the algorithm that seeks the optimal function value. Of course, the huge variety of optimization algorithms currently available and their remarkable structural complexity force us to consider subtler distinctions in our design. We therefore come to a set of four interfaces: *Solver*, *Algo*, *Meta* and *Function*.

- Interface *Solver* is the core component of the framework, the one that implements the algorithmic procedures to evaluate the objective function, keeps track of the results and moves toward the minimum.
- Interface *Algo* represents a basic algorithm that is meant to be used by a *Solver* to perform a part of its task. This interface allows the implementa-

tion of reusable procedures that can be exchanged with one another.

– The *Meta* interface is useful to plug in the framework meta- or hyper-heuristics, a racing algorithm, a portfolio selector, etc.

– Interface *Function* corresponds to the function to be evaluated in the framework.

The set of methods of each interface has been conceived as the smallest and simpler set of operations that can guarantee the appropriate interaction among the entities and with the other nodes in the network. According to our research experience, we may say that is relatively easy to adapt an existing, previously implemented algorithm to the proper GOOF interface, for it to be plugged in the P2P network of solvers.

4.1.1 The *Solver* interface

The complete source code of this interface is given in Appendix A. We refer to that for the commented list of methods to be implemented, remarking here some important points.

The internal complexity of a solver may vary greatly, depending on the implementer's choices. It may be worthwhile having a 'very basic' solver that repeatedly apply a given *Algo* (in which the real complexity resides) and simply keeps track of the best solutions found so far. Or the user could prefer to concentrate everything in the solver and even make no use of any *Algo* at all. This interface leaves the greatest freedom to the user, which is allowed to shape the solver in the most suitable fashion, as long as its interface's methods are properly implemented.

Being the indispensable component of the framework, the *SolverBox* takes care of the *Solver* initialization calling an `init` method the user must implement. This initialization is guaranteed to happen exactly once and right after

the *Solver* instantiation. The other relevant methods, dealing with the optimization task, the interaction with the *SolverBox* and the information exchange processes, are clearly explained in our API. It is important to underline that, by implementing the simple requirements of these few methods and with no knowledge about communication and internal mechanisms, the user can enable relevant information exchange with remote peer entities and be updated with fresh results coming from any other kind of solver in the network. This "double channel" of communication is made available to the *Solver* by implementing two distinct `update` methods, as the API clearly explains.

An important concept that is worth clarifying is the solver notion of "timing". Depending on the user choices, function evaluations may be performed by a *Solver* or by an *Algo*. Depending on the actual algorithm, at each iteration one ore more function evaluations may occur. Given that it is important for the *SolverBox* to know how many function evaluations have been performed at the end of each iteration, both *Solver* and *Algo* instances must implement a method (`getTime`) that serves precisely to this purpose.

Completely different from this is the notion of `timestamp`, that is the record of the moment in which an event occurred (for instance, the moment the current best result was found). We purposely left this notion somehow fuzzy, because the "counting of the time" is actually strictly related to algorithm's or user's requirements. One might consider wall clock time and refer to that to set a time limit to the computation; or the number of performed function evaluations could be taken into account; or the times some condition occurred during the computation, etc. GOOF do not force any choice and leave the user free to choose the "time representation" that better suits the user. The only requirement is that the `timestamp` can be expressed as a long value and compared against a maximum, to decide whether the computation should terminate or not.

4.1.2 The *Algo* interface

The *Algo* interface is meant to represent a basic algorithm whose design requires less effort and complexity than a *Solver*. Moreover, this layer of abstraction is a way to provide interchangeable building blocks to be used by *Solvers* to perform part of their task.

GOOF does not check for compliance between an *Algo* and a *Solver*, thus it is up to the user to ensure that a *Solver* uses a proper *Algo*. From the framework perspective, any *Algo* may be used by any *Solver*. This also means that a *Solver* may switch to different *Algos* in subsequent iterations. This interface allows the implementation of reusable procedures that can be exchanged with one another (paying attention all of them are compatible with the given *Solver*), e.g. building blocks or heuristics which a *Solver* can choose among or which a *Meta* can select and assign to the *Solver* at each iteration.

An *Algo* is identified by the other entities of the local node by its `index`. Thus the user must ensure that this identifier is unique and does not change throughout the whole computation.

The *Algo* interface is given in Appendix B. It is quite "lightweight", most of all because an *Algo* does not have any direct interaction with the *SolverBox* or with remote nodes.

4.1.3 The *Meta* interface

A *Meta* object is a component whose goal is to choose among different *Algo* instances and assign the chosen one to the *Solver*, for it to be used in the next iteration. No function evaluation is usually required to perform such a task. This interface leaves the implementer free to choose how to handle the pool of available *Algo* instances as well as any functional detail. As said, most of the meta-techniques we discussed in Section 2.2.2 may fit in this interface.

The *Meta* interface is given in Appendix C. Its set of methods provide all the

needed interaction with the *SolverBox* and the means to exchange information with similar remote *Meta* instances. It is worth noticing that only the "current" values are retrieved from the local *SolverBox* at any iteration. It is up to the *Meta* to keep track of any past information it may need. No way of accessing entities other than *Algos* is given. Interaction with the local *Solver* is transparently provided by the *SolverBox*. This way it is possible to design and use *Meta* components in a completely independent way, although the user is anyway free to "tune" a *Meta* against a specific *Solver* at will.

4.1.4 The *Function* interface

This interface (given in Appendix D) declares the few methods that are necessary to this component to interact with the *Solver* and to describe the operational domain of the problem. Its main purpose is to define a standard representation of the objective problem any GOOF-compliant module can understand and use.

4.2 The *SolverBox*

The *SolverBox* takes care of the execution of the local solver and provides it the information shared by other solvers in the network. It may be seen as a proactive 'wrapper' of the other entities, that coordinates their work and transparently handles the provisioning of data coming from remote nodes, as well as internal updates resulting from local *Solver* or *Meta* iterations.

The actual implementations of the *SolverBox* on PEERSIM and CLOUDWARE differ on various aspects, mainly because in the former the *SolverBox* is an epidemic protocol among the others, whereas in the latter it is an independent object running in a thread of its own. Notwithstanding this major dissimilarity, the actions performed are the same and the general contract with the other entities is absolutely identical.

The *SolverBox* keeps its own records of the best results found so far by any node in the network. These records are thus updated every time a new best result is found by the local *Solver* or received from a remote node. This way information can be shared among every node and 'good news' can be spread, while leaving to the specific entities the decision to take advantage of them (when, how often, etc.) or not. They may be made available to the internal entities by means of the public methods defined in the entity interfaces.

The *SolverBox* iterates the same sequence of instructions until a user defined termination criterion is met (currently, a given number of function evaluations to be performed). We may briefly summarize the series of actions performed in a *SolverBox* iteration in the following way:

1. Process any new message from other nodes in the network, updating both the *SolverBox* and the local entities as needed;

2. If a *Meta* object has been defined, make it choose the next *Algo* to be used and assign it to the *Solver*;

3. Perform the next *Solver* step, thus evaluating the *Function* and updating the internal *Solver* state;

4. If a *Meta* object has been defined, update it with the latest *Solver* outcomes;

5. If a new best result has been found in the latest *Solver* step, updates the *SolverBox* records.

At the end of each iteration, a communication event occurs, which we explain in details in the next chapter. The way the *SolverBox* interact with the *Solver* and the *Meta* instances, as well as any management mechanism are completely transparent to the user, who just need to know about and take care of the optimization component(s). The only additional effort we require from the user is to fill in the *SolverBox* textual configuration file with three basic parameters:

`maxTs` The maximum number of function evaluations to be perform by the solver run in this box (`long`).

`push-pull` Whether the epidemic communication policy of the SolverBox is push-pull or not, as detailed in Chapter 5 (`boolean`).

`solverPool` The number of solvers in the configuration among which the actual solver to be instantiated is chosen (`int`).

The first parameter is quite straightforward. About the second, we just remark that this setting concern only the SolverBox, not the peer sampling or any other epidemic protocol belonging to PEERSIM or CLOUDWARE that may serve the framework. The third parameter deserves some detail. Any *Solver* or *Meta* instance can be configured in the way the user prefers (exploiting the configuration parsing capabilities of PEERSIM and CLOUDWARE is of course an easy possibility), but their names must be included in the *SolverBox* configuration file, so that the *SolverBox* can know what has to be instantiated and run. To make it quicker to deploy and run experiment in which different *Solvers* — possibly matched against different *Metas* — run in different nodes and cooperate, we provide a way to automatically choose uniformly at random a *Solver* from a specified pool. The `solverPool` parameter gives the cardinality of this pool. In the current implementation of the *SolverBox*, this selection is made at each node uniformly at random among the *Solvers* listed in the configuration file. Further improvements of this mechanism are easy to provide as needed.

Along with the *SolverBox* and the entity interfaces we described, GOOF features several abstract classes that greatly ease the burden of plugging legacy or novel code in the general architecture. For instance, `AbstractMeta` and `AbstractSolver` give a standard and well documented implementation of most of the methods described in the respective entity interfaces, thus leaving the user free to concentrate on the algorithmic core only. Based on these abstractions, a basic `StepperSolver` is also provided, that wraps all the operations

needed to iteratively use an *Algo* and record the outcomes. By extending this solver, it is possible to plug and use a previously written algorithm in a very short time.

Presently GOOF integrates basic implementations of several optimization heuristics (PSO, RASH, various DE operators) and distributed hyper-heuristics (a distributed tabu algorithm and several racing policies). Most of this optimization techniques have been presented in our published papers. Although designed and experimented before the GOOF framework was fully developed, they have been devised according to the same modular architecture GOOF represents the completion of. And indeed their final integration in the framework has been quite effortless. In the following of this chapter we present in detail these algorithms, while in Chapter 6 we describe several published and unpublished experimental results obtained throughout our research.

4.3 Distributed PSO

In our first paper on this topic [24] we analyzed the implementation of a P2P flavor of the standard Particle Swarm algorithm (see Section 2.2.1). We showed how the 'swarm intelligence' can be implemented in a distributed fashion, so that only the total number of particle becomes relevant to the final quality of the solution. That is, our epidemic communication protocol could scale and spread the essential information among swarms running on different nodes with no remarkable performance decay. We showed that when 8 to 256 particles are working, no matter how they are partitioned among the nodes, the overall system works like a unique, giant swarm.

Our experiments pointed out that a distributed P2P-networking design of the system can actually improve the solution quality. For how unexpected could this sound, there are such cases when a lack of global knowledge is actually the main reason that leads to a more thorough exploration of the search space,

ending up to a better solution quality.

The main contribution of this paper was to present a generic distributed framework that enables experiments into such spectrum, and to discuss a first instantiation of such framework. We experimentally demonstrate that on particular conditions, our algorithm shows better performance than the original (centralized) one.

Our distributed optimization task was performed by a P2P network of nodes (solvers). Each node p, maintained and executed a particle swarm of size k. Each particle $i \in \{1, \ldots, k\}$ was characterized by its *current position* $\boldsymbol{p}_i^p$, its *current velocity* $\boldsymbol{v}_i^p$ and the *local optimum* $\boldsymbol{x}_i^p$. Each swarm of a node p was associated to a *swarm optimum* $\boldsymbol{g}^p$, selected among the particles local optima. Clearly, different nodes might know different swarm optima; we identified the best optimum among all of them with the term *global optimum*, denoted $\boldsymbol{g}$.

PSO iterated over the particles, updating the current position and velocity as described in Section 2.2.1, and selecting, after each evaluation, the best local optimum as the swarm optimum. Then an anti-entropy epidemic algorithm (see Section 2.1.2) spread information about the global optimum among nodes, working as follows: periodically, each node p initiated a communication with a random peer q (selected through a *peer sampling service*), sending the pair $\langle \boldsymbol{g}^p, f(\boldsymbol{g}^p) \rangle$, i.e. its current swarm optimum and its evaluation.

Whenever q received such a message, it compared the swarm optimum of p with its local optimum; if $f(\boldsymbol{g}^p) < f(\boldsymbol{g}^q)$, then q updated its swarm optimum with the received optimum ($\boldsymbol{g}^q = \boldsymbol{g}^p$); otherwise, it replied to p by sending $\langle \boldsymbol{g}^q, f(\boldsymbol{g}^q) \rangle$. The rate r at which messages were sent by the anti-entropy algorithm was a parameter of the algorithm related to the communication overhead: the more frequent the messages, the larger the bandwidth required.

As resulting from the presentation and the discussion of our outcomes, we showed that:

1. distributed nodes interaction through the adopted protocol is effective and

tantamount to the information sharing mechanism of the adopted solver;

2. the overhead due to epidemic communications is negligible. Networks of different sized will achieve similar performance, if they host the same number of solver processors (in this case, PSO particles);

3. a distributed and decentralized architecture cause no detriment to the optimization task and does not affect the quality of the results;

4. we devised and tested an effective way to distribute the load of a PSO computation through different machines while obtaining the same performance we would have on a single, but much more powerful, machine.

Our next work [23] proposed a detailed comparison between this distributed version of PSO and a P2P implementation of a standard Branch-and-Bound algorithm (B&B) based on interval arithmetic. A remarkable difference with respect to the previous work : we introduced network 'churning' to test how robust our architecture could prove to be. We focused on two key properties:

- scaling with the constraint of a fixed amount of available function evaluations;

- scaling with the constraint of having to reach a certain solution quality.

We can briefly summarize our results as follows. B&B is extremely fast on smaller networks, achieving better results in shorter time. Then it also has the good property of refusing to utilize all the available resources, if the function being evaluated is too easy. The embedded mechanism to achieve a good load balancing, typical of this algorithm, works effectively also in our P2P fashion. Anyway, its efficiency does not scale up to very large networks, whereas PSO can scale better, successfully exploiting the exertion of hundreds of nodes. Moreover churn is always harmful for B&B, whereas it can be beneficial for PSO (increasing the explorative drive of the algorithm). The interacting effects of problem difficulty, network size, and failure patterns on optimization

performance and scaling behavior are still poorly understood in P2P global optimization.

4.4 Distributed hyper-heuristics

Our latest work [22] has been to design and test a set of P2P distributed hyper-heuristics (HH). The topic is well known and studied, but, as far as we know, we've been the first proposing HH explicitly designed to fully exploit such a peculiar distributed environment. This time, we tested our work by means of a new ad hoc implementation of Differential Evolution framework (see Section 2.2.1). As the experimental results showed, our distributed HH perform better than the single DE operators in the large majority of experiments. We proved our HH to be more 'stable' in achieving good performances, while the usual operator can be very unfortunate on several cases. Furthermore, one of our HH has achieved a state-of-the-art result in minimizing the Cassini1 space trajectory function (see the European Space Agency web site [1] for details). This is quite remarkable, given that all the HH have not been tuned against any specific function.

Our extensive experimentation made clear that this environment favors conservative methods in general: a solver node should not change its heuristic very often. This could be due to the fact that variants of differential evolution, that we mostly use as basic heuristics due to their competitive performance and simple configuration, strongly depend on the population distribution.

Note that in the usual sequential setting, that is, improving only one population (or solution) iteratively, being conservative is very difficult at best. Consequently, our results offer a new insight that could be useful even for sequential algorithms.

In the remainder of this chapter we describe in details our set of distributed

[1] ESA Global Optimisation Trajectory Problems, `http://www.esa.int/gsp/ACT/inf/op/globopt/evvejs.htm`.

HH. All of them are based on an island model, where islands communicate through various scalable and fault-tolerant gossip protocols.

4.4.1 Pruner

The main motivation of applying HH is arguably their ability to adaptively combine search diversification and intensification in order to produce good solutions. Nevertheless, in our case, since we apply meta-heuristics as a set of basic heuristics, it might also make sense to try and pick the one that fits the problem at hand best, since meta-heuristics themselves could deal with balancing between exploration and exploitation to a certain degree, with different success depending on the problem.

The Pruner HH is designed with this idea in mind. It initially uses the entire collection of available algorithms $\mathcal{A}$, but as search proceeds, it removes more and more algorithms from this set and does not consider them anymore. At any given time, we will call the set of algorithms that are still being considered the *eligible* set.

We decrease the size of the eligible set according to a schedule that is defined by the maximal number of iterations (or cycles) I that is assigned to each island. Recall that in each cycle we evaluate one new solution. The size of the eligible set in cycle r is $|\mathcal{A}|(I - r)/I$.

The main idea is that a node applies the same algorithm until either the number of eligible algorithms decreases, or a new current best solution is received from another node through gossip. When any of these events occur, Pruner sorts the algorithms according to the best results they have produced so far and attempts to choose an algorithm that is better than the current one.

The Pruner HH is shown in Algorithm 4. In this algorithm, $stats$ stores, for each heuristic, the best solution found so far. Array $rank$ is a sorted list of the algorithms (from best to worst) based on the information contained in $stats$. Variable $curr$ holds the current algorithm.

```
for r ← 1 to I do
    if a new val has been gossiped that is better than bestVal then
        newBest ← true;
        bestVal ← val;
        bestAlg ← alg;
        stats[alg] ← val;
    n_e ← ⌈|A|(I − r)/I⌉;
    if n_e has changed or newBest then
        newBest ← false;
        rank ← sort(stats);
        i ← lookup(rank, curr);
        if i > n_e then
            i ← 1;
        else
            i ← max(0, i − 1);
        curr ← rank[i];
    val ← run(curr, bestVal);
    if val is better than bestVal then
        bestVal ← val;
        bestAlg ← alg;
        stats[alg] ← val;
    p ← getRandomPeer() ;                          // peer sampling service
    send ⟨bestVal, bestAlg⟩ to p;
```

Algorithm 4: Pruner HH

In each cycle Pruner first computes the number n_e of *eligible* algorithms. If n_e has changed from the previous iteration, or a recent gossip message has updated the best known solution, the current algorithm $curr$ to be used for subsequent run is updated as follows. First, the position of algorithm $curr$ in the sorted list of algorithms $rank$ is obtained through the lookup call. If the current algorithm is not eligible any more, we switch to the best algorithm available (that is, $rank[1]$). Otherwise, the algorithm one rank better than the current algorithm is chosen.

If none of the events happen, then nothing happens: the current algorithm is

not changed.

It is important to note that — since all nodes manage their own eligible sets that can differ — Pruner can occasionally add a removed algorithm again if a result is received through gossip that ranks the given algorithm high enough. This feature is quite remarkable: it enables a sort of "performance recovery" for those nodes that have purged the "best" available algorithm (i.e. the one that is considered the best by the "wisdom of the crowd") because of to some unfortunate iterations in which it couldn't achieve good results. Moving the current best position of a solver along with the current best algorithm in use is important to achieve a reasonable convergence of results among the nodes in the network.

As a final remark, we note that the pseudocode in 4 presents an "aggressive" version. A more "easy-going" version would choose the last *eligible* algorithm in the new rank (instead of the first) when the latest used algorithm falls out of the group of the eligible in the new rank. This can be useful to "slow down" the convergence of the solver to the same attraction basin, improving the exploration of a larger function domain.

4.4.2 Scanner

Apart from shrinking the eligible set in the same way as Pruner, the key idea of Scanner is giving a chance to all algorithms in order to get a more thorough picture of the performance of a given algorithm, and also to allow for possible synergic effects among the algorithms.

To achieve this, we implement two ideas. First, we define a minimal number of consecutive executions for each heuristic (building on the fact that our heuristics can themselves jump out of local optima). Second, we keep iterating over all the algorithms in the current eligible set and give all of them the minimal number of consecutive executions (scanning).

The Scanner HH is shown in Algorithm 5. Here, $stats[a]$ stores the *latest*

```
begin
    for r ← 1 to I do
        if newBest then
            newBest ← false;
            rank ← sort(stats);
            i ← 1;
            counter ← 0;
            phase ← SCAN;
        val ← run(rank[i], bestVal);
        counter ← UPDATESTATS(val, rank[i]);
        if counter > MaxNonImproving(phase) then
            counter ← 0;
            i ← i + 1;
        if i = ⌈|A| · (I − r)/I⌉ then                    // Eligible group size
            if phase = SCAN then
                rank ← sort(stats);
                phase ← NORMAL;
            i ← 1;
        p ← getRandomPeer() ;                            // peer sampling service
        send ⟨bestVal, bestAlg⟩ to p;

PROCEDURE updateStats(val, alg)
begin
    stats[alg] ← val;
    if val is better than bestVal then
        bestVal ← val;
        bestAlg ← alg;
        return 0;
    else
        return counter + 1;

PROCEDURE onReceive(⟨val, alg⟩)
begin
    if val is better than bestVal then
        newBest ← true;
    UPDATESTATS(val, alg);
```

Algorithm 5: Scanner HH

solution obtained by algorithm a. Additional variables are $rank$, a sorted list based on $stats$; $counter$, the number of non-improving iterations for the current algorithm; and $phase$, a state variable that stores the current phase of the algorithm: SCAN or NORMAL. Function $\mathrm{MaxNonImproving}(phase)$ takes the phase as input and returns the maximum number of consecutive non-improving iterations any algorithm is allowed to take.

This hyper-heuristic is organized in two distinct phases. Phase SCAN is activated whenever a gossip message containing a new best solution is received. At that point, algorithms are sorted based on the latest solutions they found so far (stored in $stats$) and variables are initialized in order to start scanning from the first algorithm. Subsequently, a few iterations for each of the eligible algorithms are executed, with the goal of verifying whether the new solution just received can be further improved by the remaining eligible algorithms.

When all the eligible algorithms have been tested, we switch to phase NORMAL. In this phase we keep scanning the same way as in phase SCAN except that the maximal number of non-improving iterations is larger and depends on time as well. The exact formula we use is

$$\mathrm{MaxNonImproving}(\textsc{normal}) = \lceil I/(c \cdot n_e) \rceil$$
$$\mathrm{MaxNonImproving}(\textsc{scan}) = \min(\mathrm{S}, \mathrm{MaxNonImproving}(\textsc{normal})/2)$$

where n_e is the size of the eligible set and c is the number of iterations since the current algorithm has been kept to be the current algorithm continuously. Note that since n_e can change, this recursive formula cannot be solved exactly independently of time, but nevertheless it is approximately $\sqrt{I/n_e}$. Parameter S is a constant whose value has been heuristically fixed at 15. This setting, as well as all other design decisions, are a result of an extensive experimentation with earlier versions and alternatives.

Scanner shares with Pruner the ability of recovering a previously discarded algorithm that is signalled as "best" by another node via gossiping. Here the

feature becomes even more crucial, given that the same algorithm is sure to iterate several consecutive times, once it is considered eligible. This HH proved to be able to exploit a good attraction basin better than the others, achieving some "top scored" results in several different parametrization settings.

4.5 Novel P2P-RASH implementations

As said, one of the main goal of GOOF is to provide a flexible framework to easily plug legacy algorithm and run them in a P2P distributed environment. We show the versatility of our architecture by briefly describing two novel implementation of the RASH heuristic (see Section 2.2). We devised and implemented two distinct P2P flavors of the algorithm, that make different use of GOOF interfaces and help us show here how we meant the framework to be versatile.

In the first flavor (`Rash_AR`), the heuristic is realized as an *Algo* instance, thus requiring a *Solver* to be run. As we have already seen, the thin *Algo* interface makes it simple to adapt existing code. Moreover, in this way we can immediately exploit the capability of GOOF to transparently provide shared information access. The `Rash_AR` *Algo* instance receives fresh general updates through the caller *Solver* with no major intervention on the code from the user side. In spite of the simplicity of the design, we embedded in the algorithm a simple local restart policy, to improve its performance: after a given number of iterations with no global best improvement, the algorithm is reset and restarted in a domain point that is halfway between its latest position and the global best position. If it was evaluating the very global best without being able to improve it further, it is restarted in a random point.

Embedding a restarting policy — although being it rudimental like the one we chose — inside an *Algo* instance may seem a contradiction with respect of the whole design of our framework. After all, GOOF consists of different

components and interfaces with the very purpose of decoupling different functionalities, increasing the reusability of the modules, etc. While this is entirely true, as we repeatedly said, it is also worth considering that a good practice should never become a constraint that forces you to uselessly increase the complexity of an application. In this case, our restart mechanism was contained in 5 lines of code and tightly coupled to the specific heuristic: did we really have to implement a whole new module for that? It makes sense to do that for reusable code, not for small local features, even if crucial for the specific case. This is also the reason why our interfaces are generic enough to leave the user free to shape the algorithms in the favorite way, giving opportunities for a modular design without forcing it.

Anyway, the good practice of abstracting and decoupling different functionalities is all but a nonsense. So we realized another implementation of the RASH heuristic, for which we made both a *Solver* (`RashSolver`) and an *Algo* (`RashShaker`). They are meant to be used together: while the shaker moves towards the local attractor, the solver keeps track of the search history, can send and receive specific data from other `RashSolvers` as needed (as opposite, `Rash_AR` only receives general *SolverBox* updates) and handles a slightly different local restart policy. Without going in further details, we just notice that while the first implementation realizes parallel independent RASH instances which can only share information about the global best, this second implementation makes it possible for the solvers to exchange specific data among each other, thus enabling a potentially finer coordination and pooling of the resources.

We believe this brief example of usage of our architecture may clarify the versatility and usefulness of GOOF, with respect to the purpose of easing the porting of existing code in a flexible and modular fashion. In the next section we give a second example about another important design objective which GOOF may be worth to be used for.

4.6 Experimental design of a new hybrid meta-heuristic

A further main goal of our modular architecture is to provide a flexible facility to design and test new distributed algorithms. Moreover, we claim the entity interfaces we devised may allow to easily re-use legacy code and enable different algorithms interaction with little coding effort.

To give a concrete example of GOOF helpfulness, we briefly sketch here the design of a new hybrid meta-heuristic. Our final goal will be to design and implement what we may call the RAPSO meta-heuristic, whose name derives from the two algorithms which we want to combine: RASH and PSO. Taking into account their characteristics, it may seem hard to find a way to make them collaborate. On one side, PSO is a population based algorithm that works on several candidate solution in parallel, thus trying and exploring the search region while chasing the global optimum. On the other side, RASH is a trajectory method that repeatedly refines a single solution by moving towards a local optimum.

Nonetheless, we may exploit the fact that both can work with the same representation of the search domain (vectors of real values as trial points which the objective function is evaluated in). Moreover, GOOF makes it easy for us to make them share some relevant information (the current optimum and its position, at least), thanks to its communication service. Thus we can actually devised several ways to build our RAPSO distributed algorithm, whose rationale will be to capitalize both the exploration capability of PSO and the remarkable RASH's ability in scouting around local optima.

We devised three possible designs for the RAPSO algorithm, differently exploiting GOOF's facilities:

1. A parallel concurrent implementation of RASH and PSO *Solver* instances, running in different nodes and sharing relevant information.

2. At each node, a general *Solver* runs PSO particles and RASH as *Algo* in-

stances; a *Meta* evaluates the performance after each iteration and decides which algorithm should be run in the next.

3. At each node, a PSO *Solver* runs a swarm of *Algo* particles; a *Meta* "turns" a given particle in a RASH instance according to some expected performance improvement.

The first design is the simplest and most primitive way of building a distributed algorithm on existing independent ones. It may be useful to try and see if the core ideas are effective and to which extent simple concurrent runs can improve the performances of each algorithm. To keep the things quite simple, if some "meta-mechanism" is needed (for instance a basic local restart policy), it can be implemented inside the *Solver* objects, or by means of ad hoc *Meta* instances that reset the whole *Solvers*.

The second design better exploits GOOF's abstractions and describe a modular algorithmic architecture in which any component is independent and can be switched with an equivalent one as needed. A basic solver (like the `Stepper-Solver` we mentioned in Section 4.1) may iterate either a PSO particle or a RASH shaker, according to evaluations made by a *Meta*. Decisions about the next running *Algo* at each node may be taken looking at the recent local performances, or at the latest information received via gossiping, or based on the improvement expectation given the available knowledge of the search domain, etc. GOOF makes it possible and easy to try different *Meta*, implementing different strategies, without touching the other components; or to switch among *Solver* implementations of the same algorithm to test how they fit the given problem.

Finally, the third design realizes a deeper hybridization of the two original techniques, resulting in a brand new meta-heuristic that integrates both the swarm intelligence of PSO and the aggressive local search of RASH in a single algorithm. This may be a good choice for a "final deployment stage" in which

a new algorithm is made available also as a standalone tool, that may or may not be used in a distributed pool of solvers. For it to be effective, it is required a grounded understanding of its dynamics, as well as a good comprehension of its behavior with respect to different objectives.

It is easy to see that these three designs are not an exhaustive list of possible algorithmic architectures. We presented this brief overview just to give an example of how our framework can be helpful in devising novel solutions, minimizing the necessity to code anew and providing a nice set of tool to enable a profitable cooperation among different components. In Section 6.4 we present some preliminary experimental results obtained both in a simulated and in a real deployment by implementing the first aforementioned design. In spite of the simplicity of the solution, the outcomes show that the core idea is worth investigating and improving.

Chapter 5

The Communication Service

As above mentioned, the other main feature of GOOF is to provide a communication policy between solvers in the network, thus implementing the *communication service* layer of our architecture. The actual implementation is strictly dependent on the P2P application GOOF relies on, being it PEERSIM or CLOUDWARE. However, in both cases the epidemic communication paradigm is adopted and gossip messages are sent according to one of the following *exchange policies*:

- *push*: a node spreads information about its state and processes received messages without ever replying to the sender;
- *push-pull*: a node spreads information about its state and may reply, if needed, to received messages.

Obviously, depending on which of the exchange policies has been selected, the user might observe a direct impact not only on the communication overhead, but also on the performance of the optimization task. The quality of the final solution may or may not vary, making one choice clearly preferable to the other, or suggesting a diversification related to the behavior of the specific solver that is running.

Another factor that can make a difference is the *gossiping rate*, i.e. a measure of how often an epidemic message is sent, measured in relative terms with

respect to how often a function is evaluated by a given solver. Independently of the algorithm run on the island, we always propagate the current best solution to all islands. Whenever islands perform gossiping communications with peers, they send them the best solution they know of, and when they receive it, they update their own current best solution. We assume the period of gossip to be not less than one function evaluation, which presupposes that the function is non-trivial and takes a sufficiently long time (in the order of a second or more) to compute. It can be shown that the time to propagate a new current best solution to every node this way takes $O(\log N)$ periods in expectation where N is the network size [113].

In any case, the gossiping rate can be chosen by the user, enabling a finer control of the general information sharing policy. Despite the various technical details related to the maintenance of the communication strategy, the usage of the communication service requires no deep understanding from the user. There is no further necessity to fiddle with parameters or network configurations, once the user has decided the two basic settings:

– whether the type of epidemic communication is push or push-pull and

– which is the probability that the node A sends a message to the node B at the end of each A's *SolverBox* iteration.

The way a node behaves whenever a communication event occurs deserves a deeper explanation. Both *Solver* and *Meta* are associated with a *gossip probability* parameter, respectively p_{Solver} and p_{Meta}. At the end of each *SolverBox* iteration, a random number r in $[0, 1[$ is generated, and a message is created containing (i) *Solver* information if $r < p_{Solver}$, and (ii) *Meta* information if the *Meta* component is present and $r < p_{Meta}$. Clearly no message is created if neither *Solver* nor *Meta* information has to be sent. Every message contains the best values known by the sender, along with the *Algo* that has found them (if any). Then it can also contain some entity-specific data provided by the *Solver*

	data name	data type
mandatory data	Best Value	double
	Best Position	double[]
	Best Algo	String
	Best Timestamp	long
optional data	Solver Name	String
	Solver Data	Object[]
	Meta Name	String
	Meta Data	Object[]

Table 5.1: GOOF's epidemic message format.

and/or the *Meta*. A receiver sends back a reply to the sender node if and only if the best value in the sender is worse than the current best in the receiver and the chosen epidemic policy is push-pull.

In this way, the user is able to control the information spreading rate among the various distributed components, choosing the best option with respect to their characteristics or depending on other relevant aspects (e.g. the overall number of nodes in the network, or the number of nodes hosting the same *Solver*/*Meta* as opposed to how many nodes host a different one, etc.).

Whenever a message is received, it is appended to an 'inbox' queue. All the messages in this queue are processed at the beginning of the next *SolverBox* iteration, until the queue is emptied. For each of the messages in the queue, the *SolverBox* updates its current best values as needed. Then it makes the entity-specific data in the message available to its own *Solver*/*Meta* if and only if the data have been provided by an instance of the very same class of *Solver*/*Meta*. This way, a peer-entity exclusive communication among instances of the same class, but running in different nodes, is also provided. Table 5.1 summarize the content of a GOOF's epidemic message. It is worth noticing that the user is not required to know about — nor to fiddle with — message composition at sender nodes or message parsing at receiver nodes. All it is required to the user is to implement the proper methods to handle specific *Solver* or *Meta* data within the

implemented algorithms, as we detailed in Section 4.1.

Few more notes about the sending/receiving process. PEERSIM is a single-threaded application that emulates simple transfer protocols. This means that the optimization job and the communication events are actually serialized and can be fully controlled and handled by the user. Quite a different situation we have about CLOUDWARE. This application is inherently multi-threaded and of course the GOOF implementation on CLOUDWARE fully exploits the possibility to concurrently perform optimization tasks and communications by using different threads. However, this makes things slightly less predictable than in a simulated environment, because threads are managed differently by different operating systems and their actual scheduling can vary significantly, depending on many factors (hardware architecture, current machine workload, job scheduling policy, ...), none of which may be predict or controlled by GOOF users.

Nonetheless, our tests have shown that, under fair workload conditions on the various network nodes, the behavior of a GOOF application on CLOUDWARE is remarkably stable. This means that whenever a message is created by the *SolverBox*, it is actually sent by the epidemic communication protocol within few iterations. All the same happens about message reception: once a message is received by a transport protocol, for it to be delivered to the *SolverBox* only few iterations may be required. Of course the situation may be quite worse under pathological machine conditions, or in case different nodes show very different computational performances (due to different hardware or firmware or system architecture).

Another difficult situation happens if the function to be optimized is computationally very easy. This can lead to a dramatically unfair allocation of CPU time that favors the optimization task, making the communication threads — thus, the information sharing process among solvers — have an almost negligible impact on the final solution quality. We believe that such a critical situation may hardly happen at the same time on the majority of the nodes collaborating

in the same network. Therefore the distributed optimization task does not really fail even in such a case. It is anyway obvious that the non-trivial goal of achieving an 'optimal' overall performance requires careful and deep understanding of the actual system dynamics.

In Section 6.3 we show and discuss several experimental results obtained with DE in a simulated environment. Churn has been simulated on networks of solver having different size and the gossiping optimization proved to be able to cope well with it. A clever initialization policy may be beneficial for the optimizer to avoid performance decays, especially in presence of churn, as some work recently shown [124]. But we will see that by tuning the gossiping rate with respect to the network size and the expected churn it is possible to achieve high quality results even in faulty networking environments.

Chapter 6

Experimental results

This chapter presents several experimental results obtained by using the algorithms introduced in Chapter 4. These results have been carried out in a simulated environment and they have already been published in our papers. Furthermore, we show and discuss unpublished experiments in which novel heuristics have been tested against well known test functions, that prove GOOF's extreme versatility and usefulness. Finally, we propose and briefly describe some preliminary results of a novel meta-heuristic whose prototype has been tested in both a simulated and a real deployment.

6.1 Experimental Results with Particle Swarm

In [24] we demonstrated the applicability of our P2P algorithmic architecture by discussing a first example instantiation of PSO.

We focused our attention on six well known testing functions (De Jong's F2, Zakharov, Rosenbrock, Sphere, Schaffer's F6, Griewank), whose evaluations produced a quite interesting set of results. While the first (being a Rosenbrock specialization) is defined in a 2-dimensional domain space, all the others have been evaluated in a 10-dimensional domain space. These functions, widely used in order to benchmark optimization algorithms, have been carefully selected to provide a large spectrum of behaviors with respect to their solvabil-

ity with PSO. Table 6.1 shows their analytical expression. We can say that F2 is 'easy'; Zakharov, Sphere and Rosenbrock present some 'nice' outcomes; whereas Griewank and Schaffer are the most difficult to treat. Although we are aware that this is not an insightful and thorough classification, we believe that this set represents a diversity of behaviors w.r.t. to PSO.

	Function $f(x)$	D	$f(x^*)$	K
F2	$100\,((x_1^2 - x_2^2)^2) + (1 - x_1)^2$	$[-2.048, 2.048]^2$	0	1
Sphere10	$\sum_{i=1}^{10} x_i^2$	$[-5.12, 5.12]^{10}$	0	1
Rosenbrock10	$\sum_{i=1}^{9} 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2$	$[-100, 100]^{10}$	0	1
Zakharov10	$\sum_{i=1}^{10} x_i^2 + (\sum_{i=1}^{10} ix_i/2)^2 + (\sum_{i=1}^{10} ix_i/2)^4$	$[-5, 10]^{10}$	0	1
Griewank10	$\sum_{i=1}^{10} x_i^2/4000 - \prod_{i=1}^{10} \cos\left(x_i/\sqrt{i}\right) + 1$	$[-600, 600]^{10}$	0	$\approx 10^{19}$
Schaffer10	$0.5 + (\sin^2(\sqrt{\sum_{i=1}^{10} x_i^2}) - 0.5)/$ $(1 + (\sum_{i=1}^{10} x_1^2)/1000)^2$	$[-100, 100]^{10}$	0	≈ 63 spheres

Table 6.1: Test functions. D: search space; $f(x^*)$: global minimum value; K : number of local minima.

6.1.1 Experimental Setup

Each experiment simulates n nodes, each of them runs a swarm of k particles, that repeatedly evaluate a function f. Globally, the n swarms perform e total evaluations, evenly distributed among their particles, and each node exchanges the information about the global optimum with a random peer every r local function evaluations (r is the *cycle length* of the epidemic algorithm implemented in the coordination service). Experiments are repeated 50 times, and individual dots for each experiment are shown whenever possible.

In the following, three figures of merit are considered: the *solution quality*, measured as the distance between the best known global optimum and the solution obtained by our mechanism; the *total number of evaluations*, performed globally by all swarms; and the *total time* required to complete a task. Time is measured as the number of evaluations locally performed at each node; we deliberately avoid to evaluate actual time, as it depends on the particular function evaluated and the computing power of nodes.

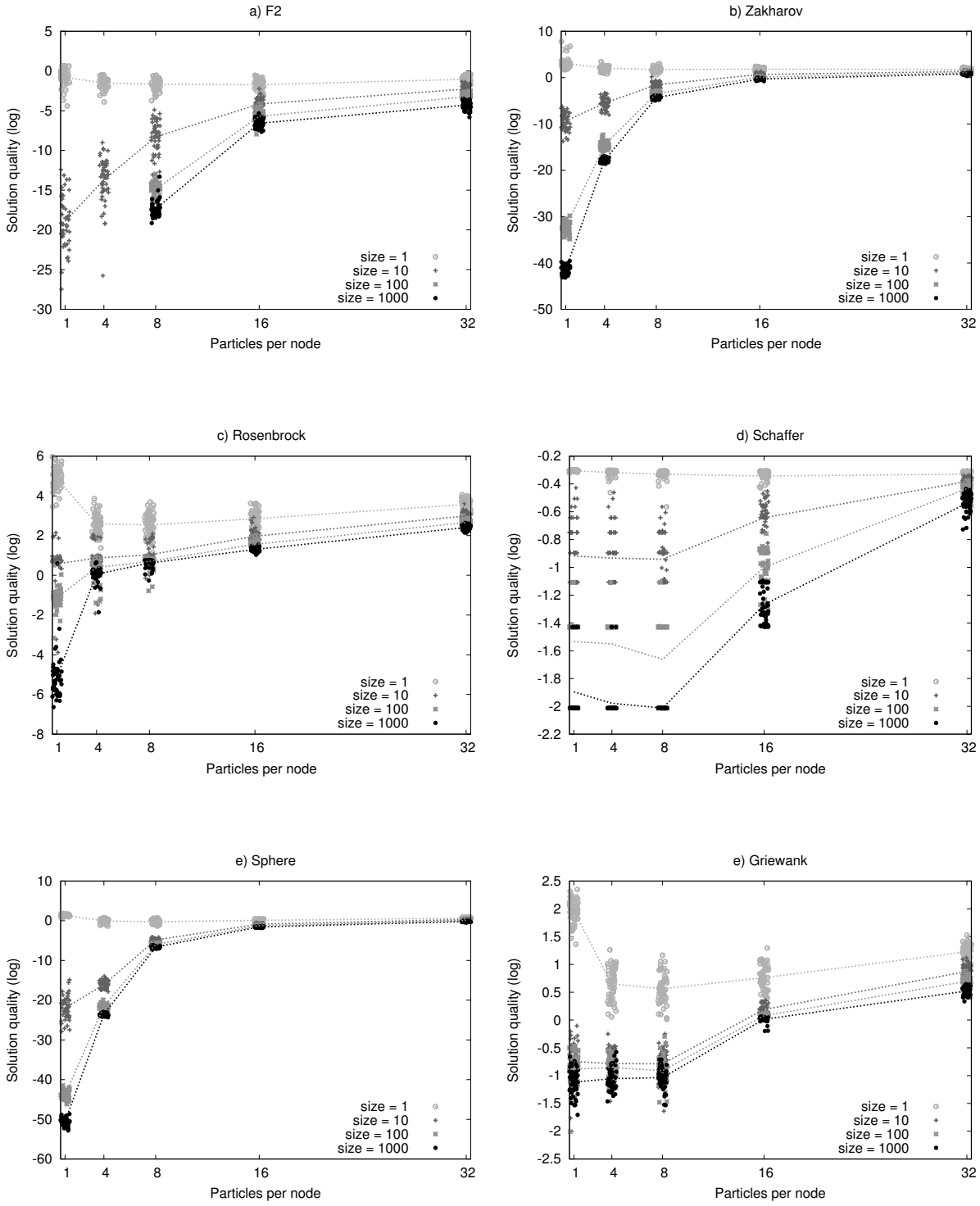


Figure 6.1: First set – Solution quality vs swarm size

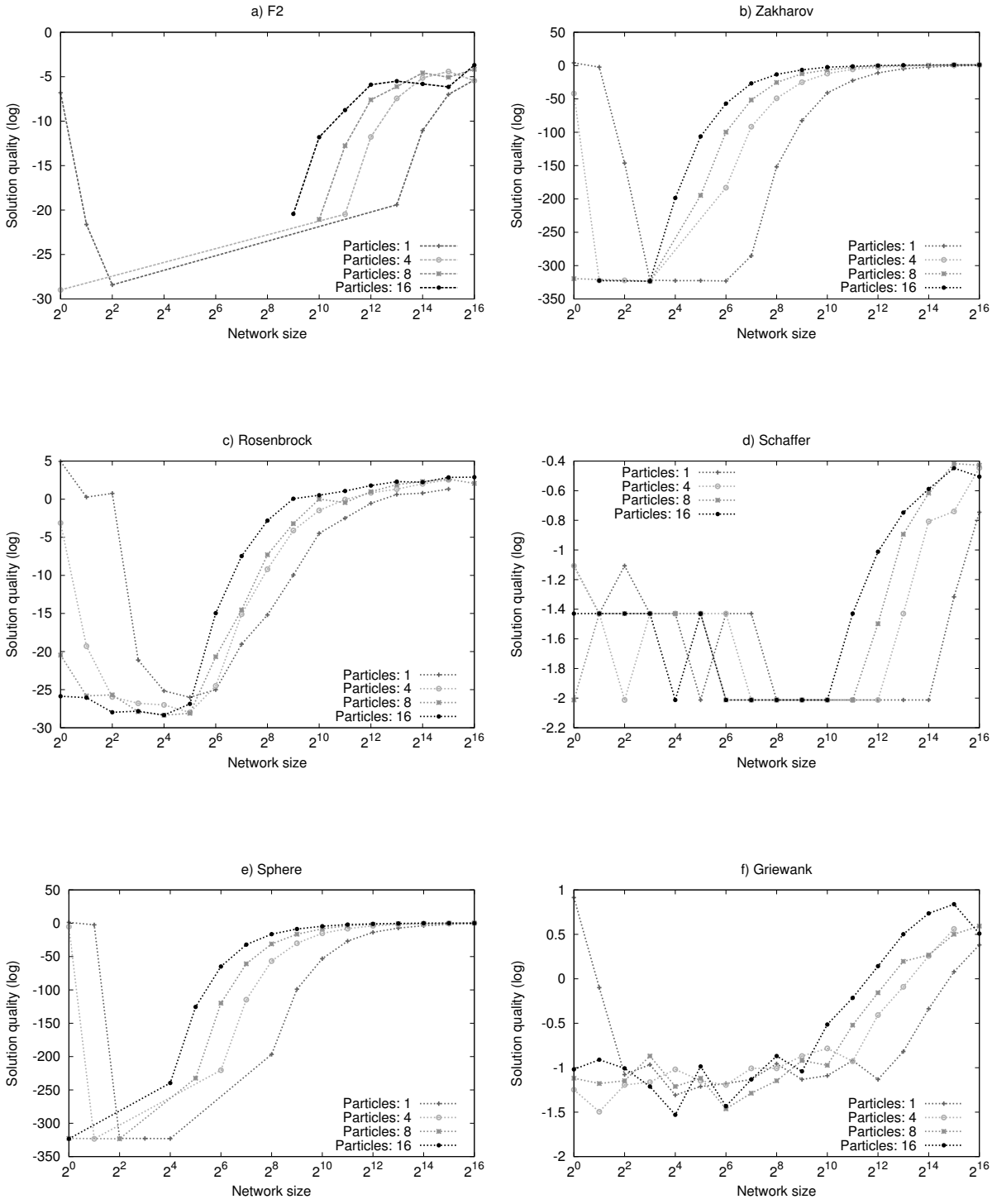


Figure 6.2: Second set - Solution quality vs network size

Other figures of merit are only briefly mentioned here, as they can easily be obtained from the other parameters. *Communication overhead* is produced by both the NEWSCAST layer and the coordination service. Depending on the expected rate of churn, NEWSCAST cycles length could be expected in the range $[10s, 60s]$; during a cycle two messages of few hundred bytes are exchanged per node, inducing an overhead of few bytes per second. Similar considerations can be done for the coordination service, but the overhead depends on the specific gossip rate adopted. The system present a high *robustness to churn*, i.e. the capability of dealing with nodes continuously joining and leaving. In fact, the reliability of the computation does not depend on any single point of failure; even if a large portion of the network fails, the computation will end successfully, slowing down proportionally to the number of failed nodes.

Param. values			avg	min	max	Var
		F2	0.0	0.0	0.0	0.0
n	$1, 10, 100, 1000$	Zakharov	0.52043	0.23106	0.95915	0.02700
k	$1, 4, 8, 16, 32$	Rosenbrock	0.07979	3.27435E-7	3.98660	0.31785
e	1000	Sphere	2.49767E-51	1.56467E-53	2.20189E-49	0.80330
r	k	Schaffer	0.00972	0.00972	0.00972	0.00000
		Griewank	0.09849	0.01232	0.28627	0.00187

Table 6.2: Adopted values for the configuration parameters and best results of the first set of experiments – Solution quality and swarm size.

Param. values			min
		F2	0.0
n	$2^i,\ i = 0\ to\ 16$	Zakharov	0.0
k	$1, 4, 8, 16, 32$	Rosenbrock	2.8890E-29
e	2^{20}	Sphere	0.0
r	k	Schaffer	0.0097
		Griewank	0.0221

Table 6.3: Adopted values for the configuration parameters and best results of the second set of experiments – Solution quality and network size

Param. values			avg	min	max	Var
		F2	1.0120E-8	1.3080E-10	1.5402E-7	0.4798
n	$10, 100, 1000$	Zakharov	0.0285	0.0076	0.0570	9.4755E-5
k	16	Rosenbrock	11.6670	5.4879	170.8049	530.4765
e	1000	Sphere	0.0027	7.4966E-4	5.8552E-3	0.0259
r	$2, 4, 6, \ldots, 64$	Schaffer	0.0105	0.0097	0.0372	0.0165
		Griewank	0.7133	0.4053	0.8868	0.0094

Table 6.4: Adopted values for the configuration parameters and best results of the third set of experiments – Solution quality and gossip cycle length.

6.1.2 Evaluating Quality

The first set of experiments is aimed at finding out as the solution quality changes with respect to the number of computing particles per node, and with respect to the number of involved nodes. In each experiment, we report the solution quality of the best global optimum found after 1000 evaluations of the function per node. Gossip rate is equal to k, meaning that a gossip exchange is performed after all the particles within the same swarm had been evaluated once. The idea here is the following: how the quality of solution change, if we are willing to dedicate a fixed quantity of time (i.e., number of evaluations per node) but a variable number of nodes to the computation? What is the influence of the main configuration parameter (swarm size) on the results?

Figure 6.1 shows the outcomes. As we can see, there is a profitable relation

Param. values			avg	min	max	Var
		F2	235940.0	186000.0	271000.0	2.48384E8
n	$2^i,\ i = 0\ to\ 10$	Zakharov	511800.0	478000.0	540000.0	1.42082E8
k	$1, 4, 8, 16$	Rosenbrock	1927000.0	1740000.0	2033000.0	9.75250E9
e	2^{20}	Sphere	36740.0	16000.0	66000.0	1.46687E8
r	k	Schaffer	192020.0	151000.0	224000.0	2.29734E8
		Griewank	–	–	–	–

Table 6.5: Adopted values for the configuration parameters and best results of the fourth set of experiments – Total time and network size

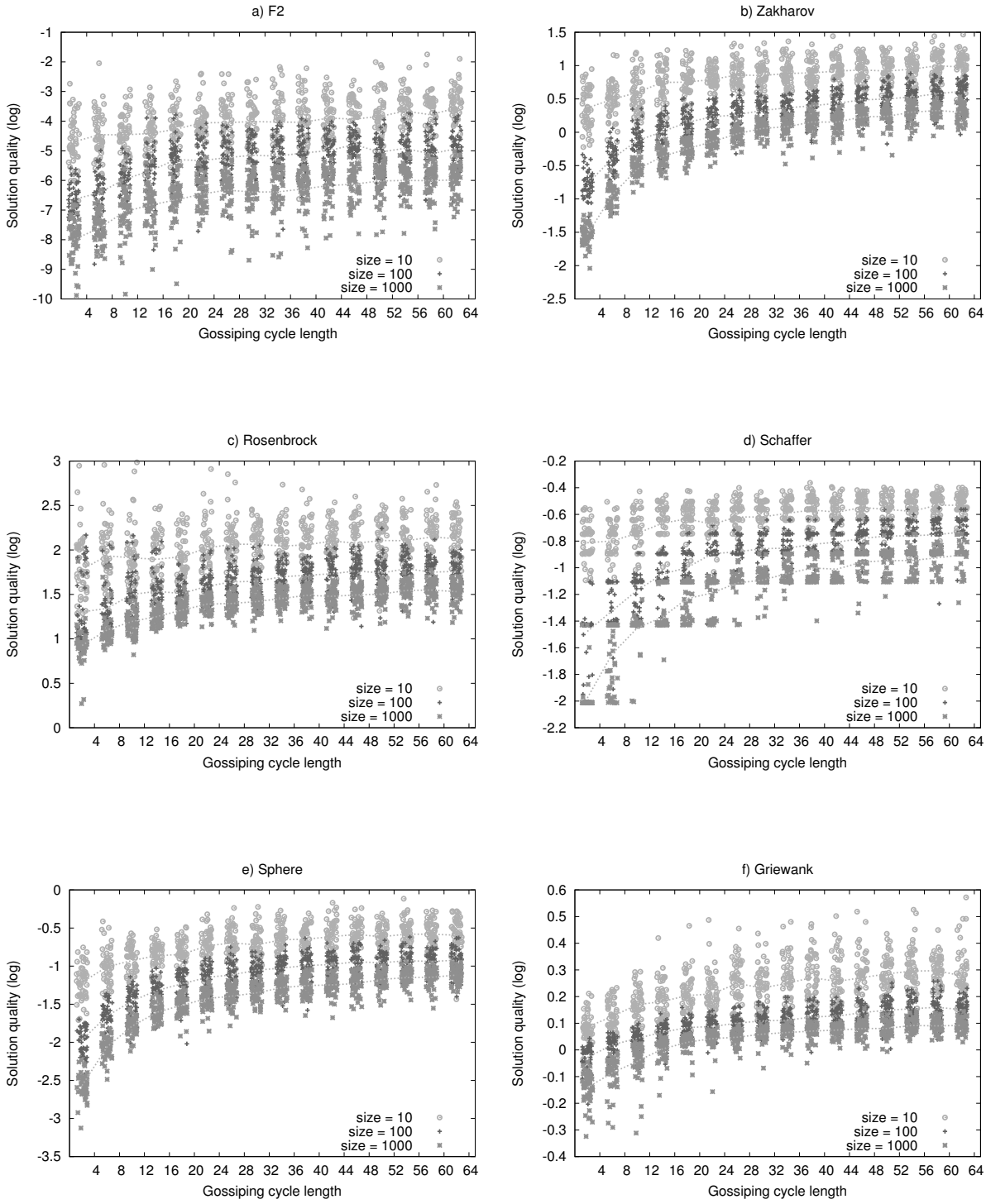


Figure 6.3: Third set - Solution quality w.r.t. cycle length

between the number of nodes and the solution quality. But this fact only holds while particles per node are bounded from 8 to 16.

The second set of experiments, reported in Figure 6.2, start with a different assumption: we are willing to provide up to 2^{20} evaluations (total) of the function. What are the best number of nodes and the best configuration parameters to obtain the maximum solution quality?

The best results are obtained when 8 to 256 particles are working, no matter how they are partitioned among the nodes. We can actually slightly narrow the empirically estimated range; a look at the raw data tells us that the most reliable particles range to evaluate the "nice functions" is 16 to 64 particles. Besides, the overhead due to gossiping communications is practically negligible. We can see this by paying attention to the fact that differently sized networks reach the same performance as soon as their number of active particles becomes the same.

So what about having different number of nodes working at the same task? The good news is given exactly by the combination of what we have just observed. The performance of PSO is mostly related to the number of working particles and not to their belonging to a particular node. This means we can choose to have different numbers of involved nodes while keeping the solution quality as much as accurate. Thus we have an effective way to distribute the load of a PSO computation through different machines while obtaining the same performance we would have on a single, but much more powerful, machine.

6.1.3 Evaluation of Cycle Length

An interesting issue is to understand if and how the cycle length can affect the effectiveness of the computation. For this reason, we ran a set of experiments in which cycle length varies between 2 and 64 local function evaluations, while all nodes have 16 particles.

Figure 6.3 shows the results. We discovered that performance is not heavily influenced by the gossip rate by itself, but rather by ratio between the gossip

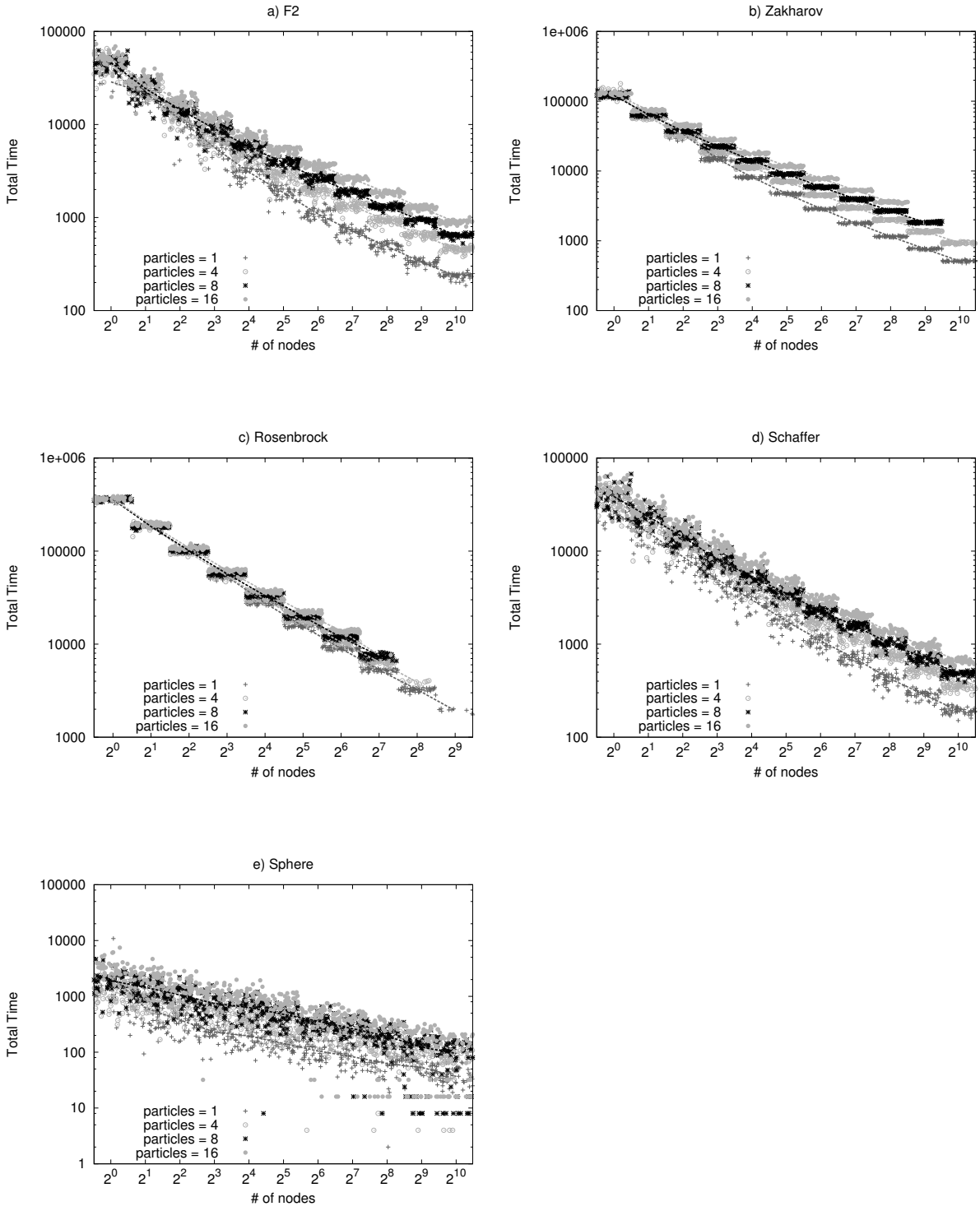


Figure 6.4: Fourth set - Total time vs network size

rate and the number of evaluations performed since the last message exchange. As this ratio tends to one, performance is sensibly better. We can therefore state that the more the swarms are exchanging information, the better the solution quality is expected to be. What follows is good news for us: the distributed nodes interaction through the adopted protocol is effective and well mimics the information sharing mechanism among the particles within the ilk swarm.

We can also see that the size of the network is important, if all other parameters (like the number of particles per swarm) are fixed. But it is also worth noticing that if the problem is inherently hard to solve for the algorithm, the cycle length is obviously less crucial, because no remarkably better value becomes available for the nodes since the former updates.

6.1.4 Evaluating Time

The final task is to evaluate how the total time required to obtain a given solution quality changes with respect to network and swarm size. In the last set of experiments, we stopped the simulation as soon as the global solution quality reaches a reasonable threshold ($1\mathrm{E}-10$).

Figure 6.4 shows that the required time is inversely proportional to the number of nodes, but proportional to the size of the swarms. This means that what is crucial here is the convergence speed of the best values within the swarms. Of course the exact amount of time required to converge – in an absolute sense – is strictly dependent on the 'niceness' of the evaluated function w.r.t. the adopted algorithm. The good news we can collect is that performing the optimization task in a distributed and decentralized fashion causes no detriment to the computation and does not adversely affect the quality of the results.

To conclude, it is worth recalling that it is not the performance of the PSO algorithm on the various functions that matters. What is interesting in our research is seeing how the performance varies when the number of nodes and all the other P2P-related settings change. Our experiments point out that a dis-

tributed P2P-networking design of the system can actually improve the solution quality. Furthermore, they give us some guidelines about which features are crucially related to performance enhancing and which seem not to be related at all.

6.2 Experimental Results with HH

The experimental results we present in this section have been published in [22]. In this work we devised and tested various novel distributed HH. The detail description of our own HH is given in Chapter 4.4. Here we recall the main data and the experimental settings, focusing on the discussion of the results.

6.2.1 Heuristics and HH characterization

A typical HH takes low level operators often classified as simple hillclimbers and mutation operators [112]. Instead, in our approach the HH operates over meta-heuristics as well. These meta-heuristics can still be classified as leaning towards exploration (diversification) or towards exploitation (intensification); the presence of both kinds of algorithms is crucial for all HH.

code	name
A1	DE/best/1/exp
A2	DE/localBest/1/exp
A3	DE/rand/1/exp
A4	DE/rand/2/exp
A5	DE/randToBest/1/exp
A6	DE/randToLocalBest/1/exp
A7	particle swarm optimization
A8	random sample

Table 6.6: The set of heuristics $\mathcal{A}$ input to the HH

name	short description
StatEq	equal share for heuristics in space
DynEq	equal share for heuristics in time
Tabu	an island-based version of [31]
SDigmo	static variant of the HH inspired by [135]
DDigmo	dynamic variant of the HH inspired by [135]
Pruner	focusing search on best heuristics
Scanner	attempting to give a chance to all heuristics

Table 6.7: Summary of our pool of HH

The set of algorithms is shown in Table 6.6. Heuristics A1-A6 are variants of

differential evolution. We use standard notation as proposed in [127]. Due to the parallel setting, some explanations are in order. Here, "best" means the global best solution in the network (as learned through gossip, see above). Notation "localBest" in A2 implies the "best" variant with the best solution interpreted as the local best solution within the island: this variant ignores the global best solution so the islands are isolated. Similarly, "rand" variants are also interpreted to be local to the island. Heuristic A5 is like the "2" variant but using one random solution from the population and the global best; A6 is the isolated version of A5.

Algorithm A7 is described in [28]. It is a simple island-based PSO algorithm, assuming the best solution PSO relies on is the global best, propagated through gossip. Finally, A8 returns a random solution from the range of the function at hand.

All these algorithms are population-based (except A8, which is stateless). We assume that all islands maintain a population of size 8. This makes it possible for a HH to change the algorithm while keeping the population.

Table 6.7 shows our pool of HH. We include in our pool two trivial HH as a baseline. The first is called StatEq, a shorthand for "static equal share". StatEq assigns a heuristic to each island at the beginning of the run and does not change this assignment anymore. Furthermore, it assigns an equal number of islands to all heuristics. Note that StatEq can easily be implemented as a local algorithm without global consensus, if necessary: for example, each node can select an algorithm at random at the beginning, and then stick to it throughout the run (depending on network size, this introduces some variance). The second is called DynEq, for "dynamic equal share". It assigns a random heuristic to all islands after each cycle (where one cycle within an island denotes generating one new solution using a heuristic) at random, giving an equal probability to all heuristics. For the description of SDigmo and DDigmo HH, we refer to our paper [22].

	Function $f(x)$	D	$f(x^*)$	K
Sphere10	$\sum_{i=1}^{10} x_i^2$	$[-5.12, 5.12]^{10}$	0	1
Rosenbrock10	$\sum_{i=1}^{9} 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2$	$[-100, 100]^{10}$	0	1
Zakharov10	$\sum_{i=1}^{10} x_i^2 + (\sum_{i=1}^{10} ix_i/2)^2 + (\sum_{i=1}^{10} ix_i/2)^4$	$[-5, 10]^{10}$	0	1
Griewank10	$\sum_{i=1}^{10} x_i^2/4000 - \prod_{i=1}^{10} \cos\left(x_i/\sqrt{i}\right) + 1$	$[-600, 600]^{10}$	0	$\approx 10^{19}$
Schaffer10	$0.5 + (\sin^2(\sqrt{\sum_{i=1}^{10} x_i^2}) - 0.5)/$ $(1 + (\sum_{i=1}^{10} x_1^2)/1000)^2$	$[-100, 100]^{10}$	0	≈ 63 spheres
Levy4	$\sin^2(3\pi x_1) + \sum_{i=1}^{3}(x_i - 1)^2(1 + \sin^2(3\pi x_{i+1}))+$ $(x_4 - 1)(1 + \sin^2(2\pi x_4))$	$[-10, 10]^4$	-21.502356	71000
Cassini1	description available from ESA at `http://www.esa.int/gsp/ACT/inf/op/globopt/evvejs.htm`			
Cassini2	description available from ESA at `http://www.esa.int/gsp/ACT/inf/op/globopt/edvdvdedjds.htm`			

Table 6.8: Test functions. D: search space; $f(x^*)$: global minimum value; K : number of local minima.

We selected well-known test functions as shown in Table 6.8. We included Sphere10 as an easy unimodal function. Rosenbrock10 and Zakharov10 are included as non-trivial unimodal functions. The rest of the functions are multimodal. Griewank10 is similar to Sphere10 with high frequency sinusoidal "bumps" superimposed on it. Schaffer10 is a sphere-symmetric function where the global minimum is surrounded by deceptive spheres. Levy4 is not unlike Griewank10, but more asymmetric, and involves higher amplitude noise as well.

Cassini1 and Cassini2 are realistic applications related to the Cassini spacecraft trajectory design problem of the European Space Agency (ESA). The two problems have 6 and 22 real variables, respectively, and an unknown number of local optima. These problems have been studied extensively and are known to contain an enormous number of local optima and to be strongly deceptive for local optimizers [2].

6.2.2 Experimental Setup

In our experiments we varied the following parameters:

- **network size** (N) the number of nodes (islands) in the network;

- **function evaluations** (E) the number of *overall* function evaluations performed in the network.

For a combination of network size N and overall function evaluations E, all islands are assigned an equal number of function evaluations: E/N.

We ran 10 independent experiments for all parameter combinations using

$$N \in \{2^0, 2^1, \ldots, 2^{16}\} \text{ and } E \in \{2^{10}, 2^{13}, 2^{17}, 2^{20}\},$$

combined with all possible algorithms in Table 6.7 *and* the standalone versions of the algorithms in Table 6.6, on all test functions. The outcome of a single experiment is the best solution found in the network during the experiment.

Our primary goal was to compare the algorithms from the point of view of stability and reliable good performance across a wide range of parameters, since these are the trademark features of a good HH.

To clean the generated data from noise, before analyzing the results we first selected only one value for parameter E for each function. The reason is that if E is too large, then the results are inconclusive: all algorithms produce almost identical results very close to the global optimum, which makes it impossible to differentiate between the algorithms. This was problematic especially for the very easy functions: Sphere and Zakharov.

If E is very small, then none of the algorithms produce very good results, so comparison is again not very realistic or interesting. We selected the value that differentiates most among the algorithms: $E = 2^{20}$ for Cassini1, Cassini2, Griewank, Schaffer and Rosenbrock; $E = 2^{17}$ for Levy, and $E = 2^{13}$ for Sphere and Zakharov.

From the remaining dataset, we filtered out those network sizes that, for a very similar reason, also hinder meaningful comparison: too large sizes ($N \geq 2^{14}$) allow too few evaluations per island even for 2^{20}, the largest value of E we used. Besides, in small networks ($N \leq 2^2$) the behavior of the algorithms is rather different, and, quite interestingly, results for the same value of E are of

	Number of times best, 2nd best,. . .,10th best									
StatEq	4	12	8	7	4	7	3	5	3	1
SDigmo	6	4	6	11	10	6	3	1	2	4
Pruner	5	6	11	7	7	4	3	3	3	3
A1	9	2	1	3	5	11	7	3	2	2
Scanner	7	5	6	1	5	2	5	2		5
A4	8	7	1	1	3	4	3	5	3	5
DynEq	2	2	3	4	2	5	7	7	7	5
DDigmo	1	3	4	3	2	2	9	8	6	7
A5	4	3	4	5	4	2	1	3	1	4
A7	5	6	2	1		3	2	7	3	3
A6	2	1	2	7	4	2	3	3	2	1
A3	2	2	4	3	1	1	3	1	5	8
Tabu	1	2	2		3	3	4	5	8	4
A2		1	2	2	4		1	2	7	2
A8				1	2	4	2	1	4	2

Table 6.9: Mean best fitness rank statistics.

lower quality than in larger networks. Since we are interested in relatively large networks, these differences distort our conclusions as well.

6.2.3 Dominance Analysis

In the remaining data, we were interested in characterizing dominant protocols, that perform well in all cases. In our paper we give a detailed dominance analyses by discussing several dominance matrixes that show an insightful comparison of the HH performances. We recall here the tables and the plot which summarize our outcomes.

We show ranking information regarding mean best fitness in Table 6.9. In the table the first column contains the number of different parameter settings where the mean best fitness of the given algorithm was best; the second column contains the number of times it was second best, etc.

First of all, we can see that the most dominant HH is one of our baseline

	Number of times best, 2nd best,. . .,10th best									
A1	20		4	2	6	6	5	5		1
StatEq	5	9	6	7	4	8	5	4	2	2
SDigmo	4	7	10	7	7	3	5	3	2	1
Pruner	3	8	6	5	1	4	4	4	4	4
DynEq	2	6		6	4	6	5	5	8	7
A4	4	7	1	5	4	5	5	4	2	7
Scanner		4	8	6	11	5	1	2		
A7	4	4	5	1	3	6	6	9	6	1
DDigmo	5		3	3	4	4	8	7	5	5
Tabu	1	2	1	6	6	4	4	1	4	2
A5	1	3	3	2	1	1	3	5	4	5
A3	2	2	1	2	2	2		2	7	6
A2	5	2	3				1	2	7	5
A6		1	3	3	1	1	3	3	3	5
A8		1	2	1	2	1	1		2	5

Table 6.10: Minimal best fitness rank statistics.

heuristics, StatEq. As a general pattern, we can observe that approaches that tend to be static and do not change the heuristic on an island often tend to be better (more dominant) than the dynamic variants, so this feature seems to be desirable in an island model.

Another observation is that HH consistently and very convincingly dominate all algorithms in $\mathcal{A}$, which clearly underlines the main advantage of HH. The best performing algorithm according to this measure is A1, which ranks 4th.

Looking at Table 6.9 however, we can observe that A1 has the largest number of wins among the possible parameter settings. There is a catch though: its ranking distribution is bimodal: it has another peak at around rank 6; this means that A1 is often the best, but when it is not best, it is rather bad. HH show a more reliable and stable pattern.

This is illustrated even better by Table 6.10 which, instead of the mean best fitness, is calculated based on the *best* result of the 10 independent runs: we

can see that A1 can be very good, but this performance is very unreliable. Of course, dominance depends on the set of test functions we have examined. We tried to remove the easiest functions from the dataset: Sphere and Zakharov. These functions are too easy for most of the algorithms so they should have less weight in the comparison.

Algorithm A1 now jumped back: in fact, it turned out A1 excels on the easy functions primarily. However, the best three HH were still the same, which gave further evidence that a good HH can in fact achieve a better performance than any of the basic algorithms it is based on, and this performance is rater stable as well.

6.2.4 Performance on Test Functions

We identified StatEq, Pruner, and Scanner as the best HHs, and A1 and A4 as the best basic heuristics. Figure 6.5 presents mean best fitness as a function of network size for the non-trivial test functions.

We can observe that StatEq is very stable and tends to be at the lower bound of the other algorithms (or even better than all, see Cassini1) except for a few special cases where A4 and Scanner show a good performance in a small region of the parameters.

Finally, we note that Scanner actually improved the best known solution to Cassini1.[1] Scanner, Pruner and SDigmo produced competitive results for Cassini2 as well, e.g. SDigmo reached 8.410157744690402, although with tuned parameters and $E = 2^{23}$. However, this might serve as an reminder that although StatEq is the most stable dominant method, and as such the most preferable HH in our set, for specific problems other heuristics could produce a better peak performance.

[1] Cassini1$(-789.7652528252638,\ 158.30958439573675,\ 449.38588149034445,\ 54.713196036801925,\ 1024.7266958960276,\ 4552.859162641155) = 4.930707804754513$

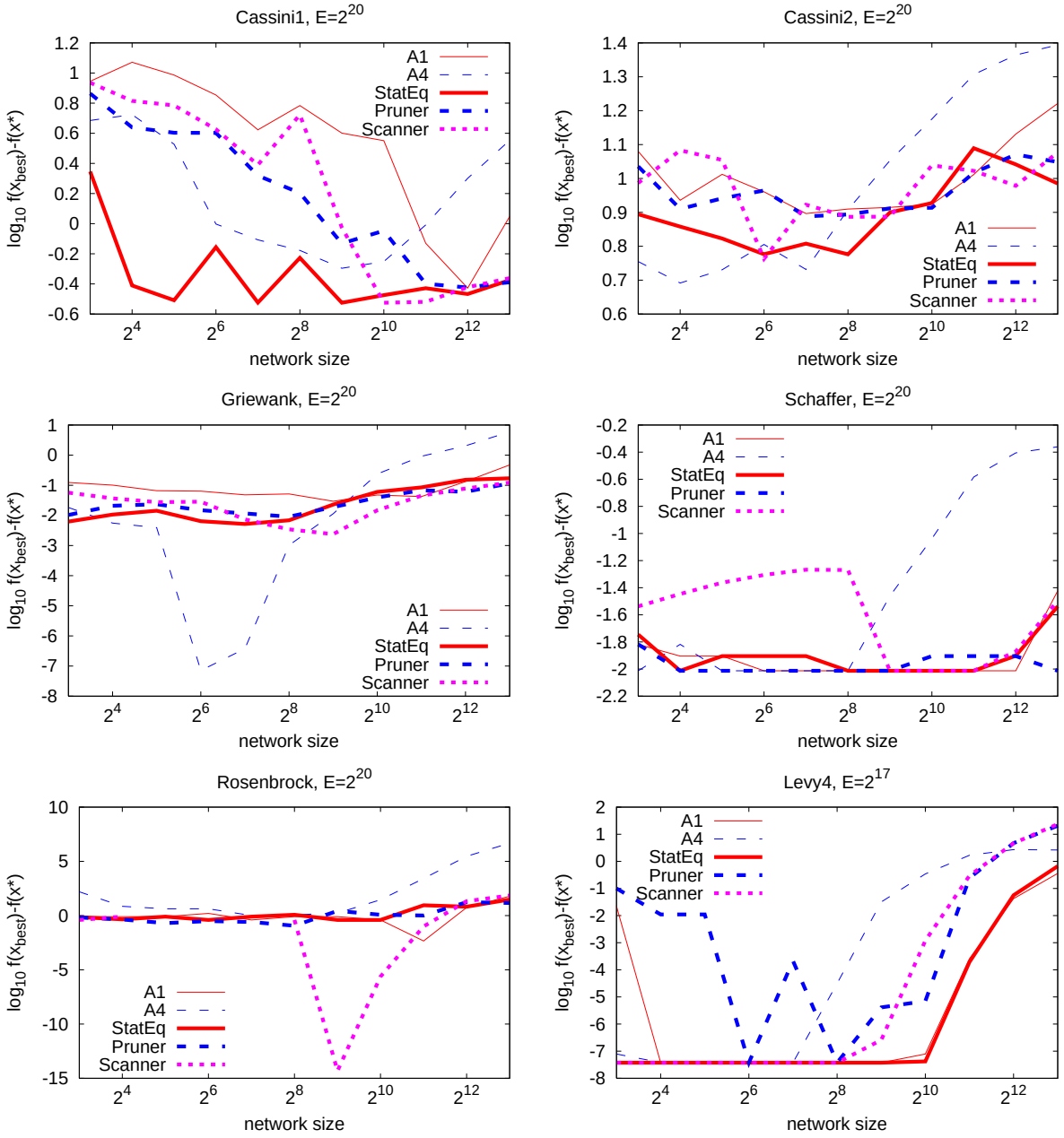


Figure 6.5: Mean best fitness (expressed as the difference from the optimal solution) on the non-trivial test functions as a function of network size.

6.3 Novel Experimental Results with Differential Evolution

So far, some interesting results have been shown about performing distributed optimization by means of DE in P2P simulated networks. Exploiting the facilities provided by GOOF we have been able to achieve brand new outcomes that reveal novel ways of applying DE in such a decentralized fashion, fully exploiting the possibility to divvy up the optimization task among a large number of solvers in a more effective way.

[22] described an original strategy for applying DE to a given problem. Several nodes cooperate in a P2P fashion to optimize a function, each of them running an instance of DE on a local population. At the end of every DE iteration (selection of individuals, application of an operator, evaluation of the function on the resulting individual, solution quality assessment) each node send a message to another node, randomly chosen by a peer sampling service. The message contains the best individual found so far. This way the nodes (in that case, we can say the solvers) share useful information and may use the updates coming from other peers to improve their own solution quality.

While paying great attention to spreading good results, no interaction was provided among distinct populations on different nodes. That is precisely the issue we address in this set of experiment, by plugging the very same DE source code on GOOF (with only minor modification needed to wrap the code to obtain entity interface compliance) and adding a more sophisticate information sharing policy, whose results are somehow surprising.

We enable a mechanism for sharing individuals among different populations at a given rate. This way every population, previously modified just by the "local" DE solver, is actually changing also because of the substitution of one of its individuals with a "foreign" one, coming from a different node (provided that the newcomer has a better solution quality). This procedure definitely makes sense, because it makes the various DE instances capable to exploit a far richer

population while operating on the usual amount of local individuals. Moreover, the poorly performing individuals are *gradually* substituted by better newcomers, so that a local population is not simply "killed" and overtaken by a bald new one. This preserves the precious local diversity of the global population and the fair partitioning of the function domain space among the solvers.

In the following, we give a more technical description of our experimental settings and present some results.

6.3.1 Experimental settings

The results we present are obtained by using the GOOF implementation on PEERSIM. As explained in Section 3.3, we rely on PEERSIM's peer sampling facility to maintain a random neighborhood structure in each node: at any point in time a node can request a random node address from the local peer sampling service, that returns a random sample from the entire network.

Considering the *optimization service*, we adopt the source code already used in [22] as a legacy and easily plug it in the framework:

– the DE scheme implementing the *Solver* interface

– the various DE operators implementing the *Algo* interface

– the objective functions implementing the *Function* interface

In each node the *SolverBox* follows the steps described in Section 4.2, making DE optimizing the given function by using a given operator. Though fully implemented, no *Meta* entity is used, so we have a network of solvers using the same operator from the beginning till the end of each experiment.

Information are spread by GOOF's epidemic protocol (*communication service*) that works as described in Chapter 5, following a *push-pull* policy. The *Solver* (DE) always propagates the current best solution to another node after

every iteration. Moreover, with a certain probability G (we detail in the following) the *Solver* may also send to the same node the individual that have been updated more recently among those belonging to its local population. So we assume the period of gossip to be one function evaluation for the best result records and proportional to G for the population individuals.

As already mentioned, the time to propagate a new current best solution to every node this way takes $O(\log N)$ periods in expectation where N is the network size [113]. In principle, the time to propagate the selected population individual would asymptotically be the same, but it is quite unlikely that the very same individual can be spread to more than one node without having being modified by the DE operators. We believe that this is the very reason why our mechanism turns out to be so effective in achieving good results: the global population gradually improves, because the single individuals, one at a time in each local population, gently improve, thus focusing the evaluated search domain toward more promising areas.

We recall in Table 6.11 some characteristics of the test functions we challenge. Sphere10 is an easy unimodal function. Rosenbrock10 and Zakharov10 are non-trivial unimodal functions. The rest of the functions are multimodal. Griewank10 is similar to Sphere10 with high frequency sinusoidal "bumps" superimposed on it. Schaffer10 is a sphere-symmetric function where the global minimum is surrounded by deceptive spheres. Rastrigin10 is highly multimodal and the locations of its local minima are regularly distributed. It is considered as difficult for most optimization methods.

The set $\mathcal{A}$ of DE operators we use is shown in Table 6.11 as well. We remind their characteristics. "Best" means the global best solution in the network (as learned through gossip). The "localBest" variant ignores the global best solution and just considers gossiped newcomers among its population individuals. Similarly, "rand" variants are also interpreted to be local to the node with respect to the best known values. Heuristic A5 uses one random solution from the

op code	op name	Function $\mathbf{f}(\mathbf{x})$	D	$\mathbf{f}(\mathbf{x}^*)$	K
A1	DE/best/1/exp	Sphere10	$[-5.12, 5.12]^{10}$	0	1
A2	DE/localBest/1/exp	Rosenbrock10	$[-100, 100]^{10}$	0	1
A3	DE/rand/1/exp	Zakharov10	$[-5, 10]^{10}$	0	1
A4	DE/rand/2/exp	Griewank10	$[-600, 600]^{10}$	0	$\approx 10^6$
A5	DE/randToBest/1/exp	Schaffer10	$[-100, 100]^{10}$	0	≈ 63 spheres
A6	DE/randToLocalBest/1/exp	Rastrigin10	$[-5.12, 5.12]^{10}$	0	$\approx 10^6$

Table 6.11: DE operators and test functions. D: search space; $f(x^*)$: global minimum value; K: number of local minima.

(local) population and the global best; A6 is the "isolated" version of A5.

We assume that every *Solver* maintains a local population of 8 individuals. In all our experiments we vary the following parameters:

- **network size** (N) the number of nodes in the network;
- **individual gossip rate** (G) the probability to send, after every function evaluation, the sub-optimal individual in the local population that has been updated more recently.

For a combination of network size N and individual gossip rate G, all solvers are assigned an equal number of function evaluations: $2^{20}/N$.

We run 10 independent experiments for all parameter combinations using

$$N \in \{2^1, \ldots, 2^{14}\} \text{ and } G \in \{0, 0.016, 0.125, 0.25, 0.5, 1\}$$

combined with all the operators in Table 6.11, on all test functions.

For every experiment, we trace the best solution known in the overall network every time 8 new iterations of the *Solver* have been performed in each node. Thus we collect the current best result at the end of each DE generation cycle. The sets of experiments described so far are run separately in different network conditions. We considered the following scenarios:

– No churn.

– Churn drawn by replacing 5% of nodes during a time interval taken by 20 function evaluations.

– Churn drawn by replacing 10% of nodes during the same time interval.

Our metric of interest is based on the number of function evaluation performed at each node. Objective function can differ in complexity by several orders of magnitude. Thus, generally we cannot fixed in an absolute way the amount of time these churn rates represent. Considering that a non-trivial function may take 1 second per evaluation, we simulate two challenging scenarios, where on average 5% and 10% of the nodes, respectively, fail every 20 seconds and the same amount joins the optimization task with no particular initialization.

6.3.2 Some commented results

In the rest of this section, we show a collection of plots representing a selection of our results. Our comments and remarks derive from the analysis of all sets, of course, but being the overall number of experiment sets more than 3000, we collect here just some representative examples. All the values shown in the plots are the averages of the best values found over ten experiments.

The first evidence we find is that the individual spreading mechanism is definitely effective. All the operators remarkably improve their performance when the gossip rate G is set to a non-zero value. There is an interesting dependence from the network size, though, which is worth investigating in some details. It is not true that setting $G > 0$ is always beneficial to the computation, but it can have a significant impact if done in the appropriate way with respect to the network size, leading to a huge performance boost.

Generally we see that lower gossip rates improve the performances in small and medium networks, while higher gossip rates have a surprisingly good impact in large networks. Of course there are no “magic numbers” that can suit

all the network sizes for all the operators, but this kind of behavior appears in all our experiments, with a surprising consistency. Figure 6.7 illustrates how an appropriate choice of G leads to very good results even when using a very large number of nodes. This fact is not usual at all and shows that exploiting large decentralized P2P networks to perform function optimization tasks can be not only efficient, but most of all effective. A large number of function evaluations can be partitioned among thousands of distributed solvers, resulting in a lighter workload per machine, while ending up in an equally good, if not better, result.

Not only our gradual shuffling of population individuals improves the *final* results, but it also speeds up the improving *during* the computation, with respect of the case in which no population gossiping is provided. As we examine the experiments in which an operator is able to find the optimum of a function even if $G = 0$, we see that most of the times, for $N > 2^4$, a small gossiping rate helps finding the optimum within a smaller number of function evaluations. This effect is even more evident in those cases when the operator is not performing well by itself, as Figure 6.6 clearly illustrates.

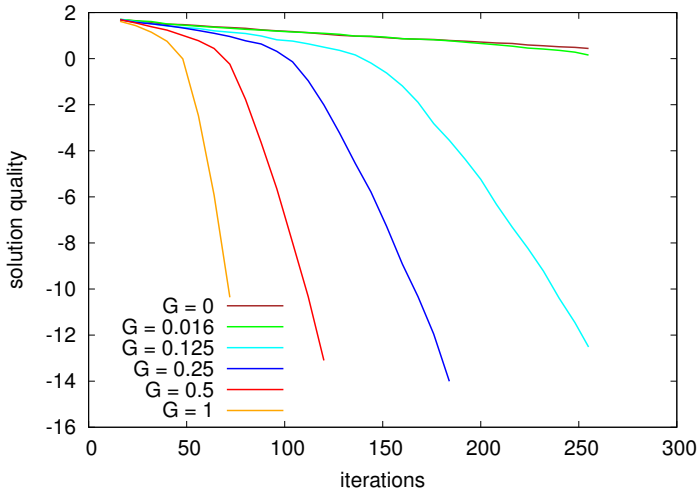
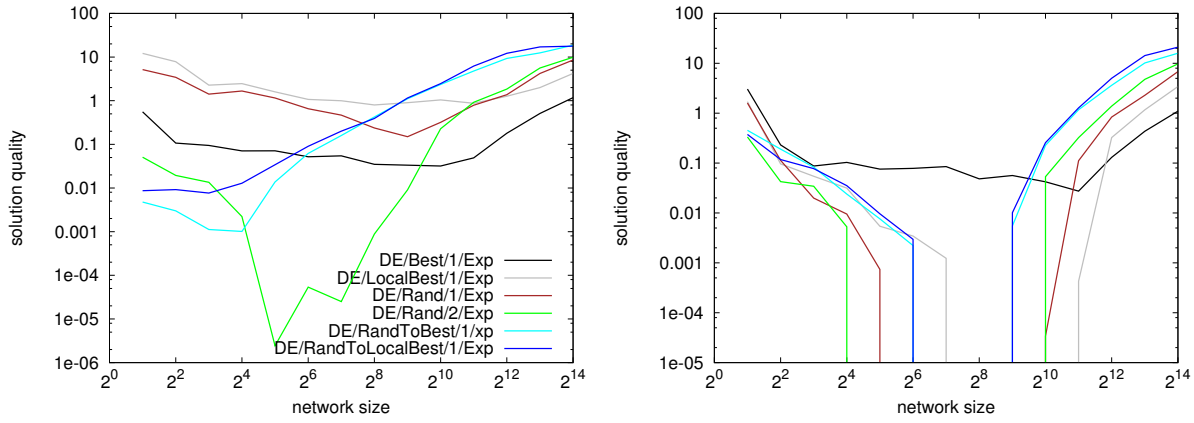
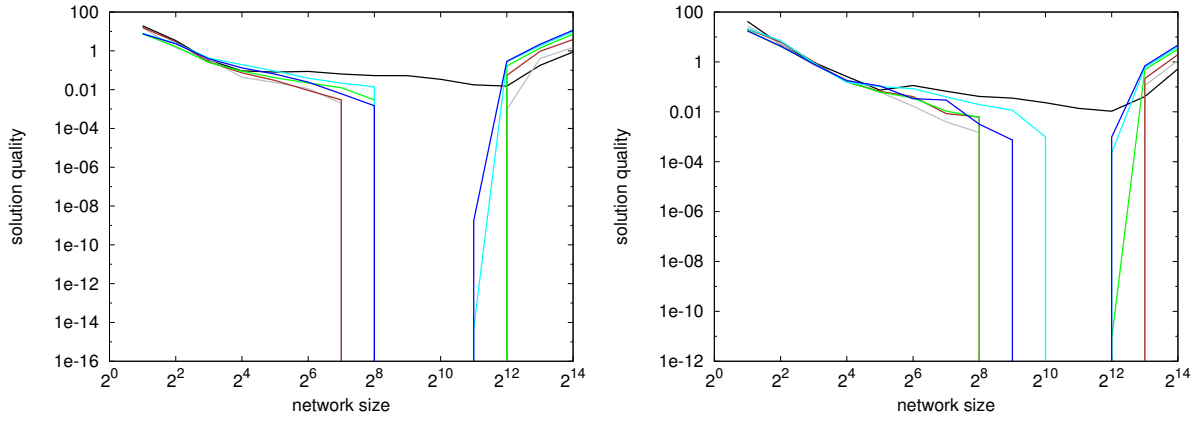


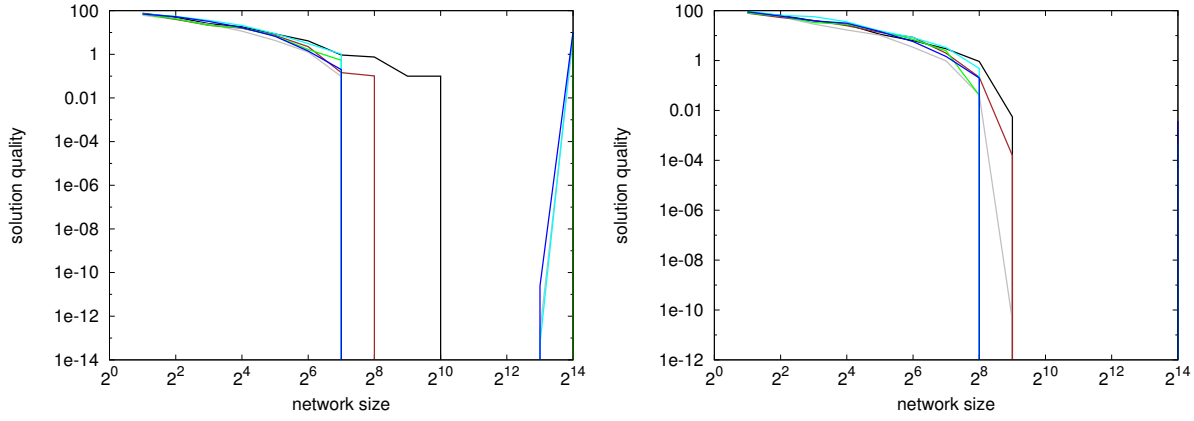
Figure 6.6: Rastrigin - DE/Best/1/Exp. $N = 2^{12}$. How the gossip rate G improves performances.



(a) $G = 0$ (b) $G = 0.016$

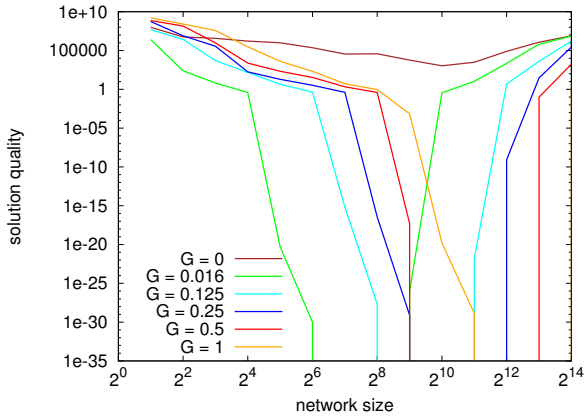


(c) $G = 0.125$ (d) $G = 0.25$

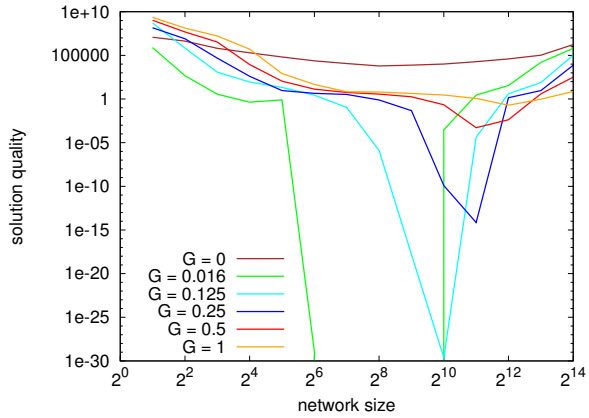


(e) $G = 0.5$ (f) $G = 1$

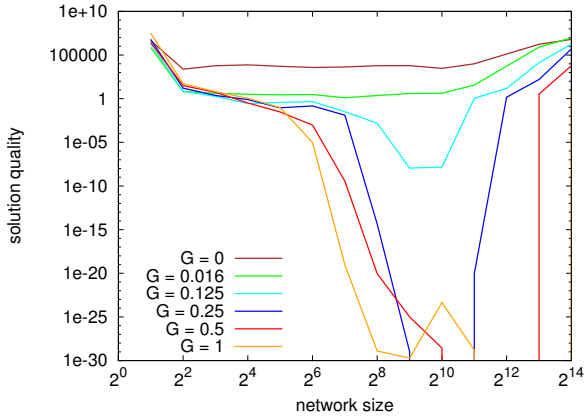
Figure 6.7: Griewank - How the gossip rate G improves performance in different networks. No churn.



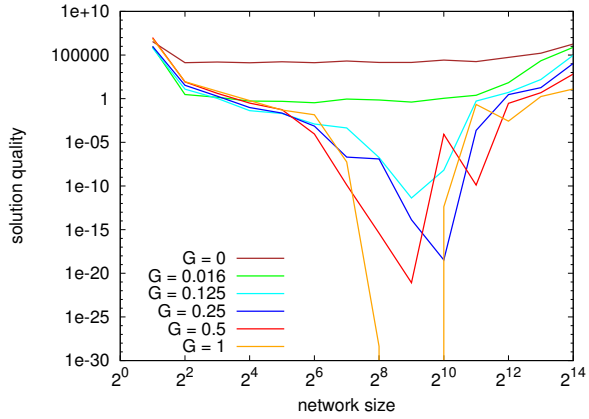
(a) DE/Rand/1/Exp – No churn



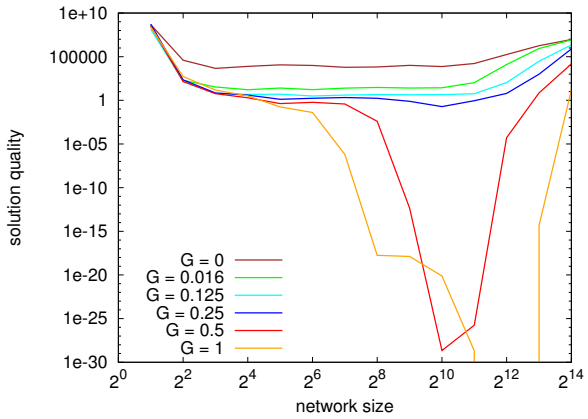
(b) DE/Local/1/Exp – No churn



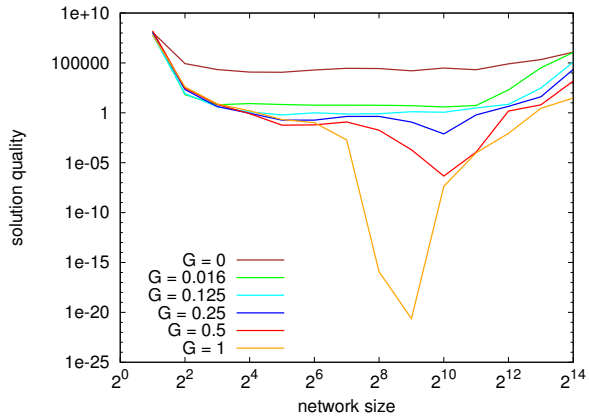
(c) DE/Rand/1/Exp – 5% churn



(d) DE/Local/1/Exp – 5% churn



(e) DE/Rand/1/Exp – 10% churn



(f) DE/Local/1/Exp – 10% churn

Figure 6.8: Rosenbrock - How the gossip rate G improves performances in different networks.

Figure 6.8 shows how performance changes in presence of churn. As we can see, population gossiping helps even in case of high churn. Paying attention to this graphs, we see something somehow unexpected: churn can even be helpful for large networks, given that an adequate gossip rate is enabled! We find — not only in this specific case: as we said, these considerations can be generally drawn out of the whole set of experiments we performed — that when churn is expected, a higher gossip rate is preferable. As a matter of fact, making individuals circulate the network after every function evaluation can successfully cope with the higher churn rate, provided we have networks that are large enough to guarantee a proper amount of individuals. In some cases, churn even helps improving the final outcome, as Figure 6.9 and Figure 6.10 clearly show.

A similar behavior was observed also in a previous work of ours [23], where we show that sometimes churn can be beneficial for our P2P-PSO in some test cases. We think this similar behavior is due to common reasons. When churn occurs during PSO optimization, the particles actually restart in random positions. Churn during DE optimization makes local populations disappear and be substituted by new individuals. These two scenarios present important similarities and, in both cases, what actually happens is that the exploration of the search space has a boost. Of course, the performance benefit is not guaranteed. It may greatly improve the solution quality if the particles/individuals are suffering for lack of possibilities to escape from local minima. But it just spoils the chance to improve, if the solvers are on a good "track" within a promising attraction basin.

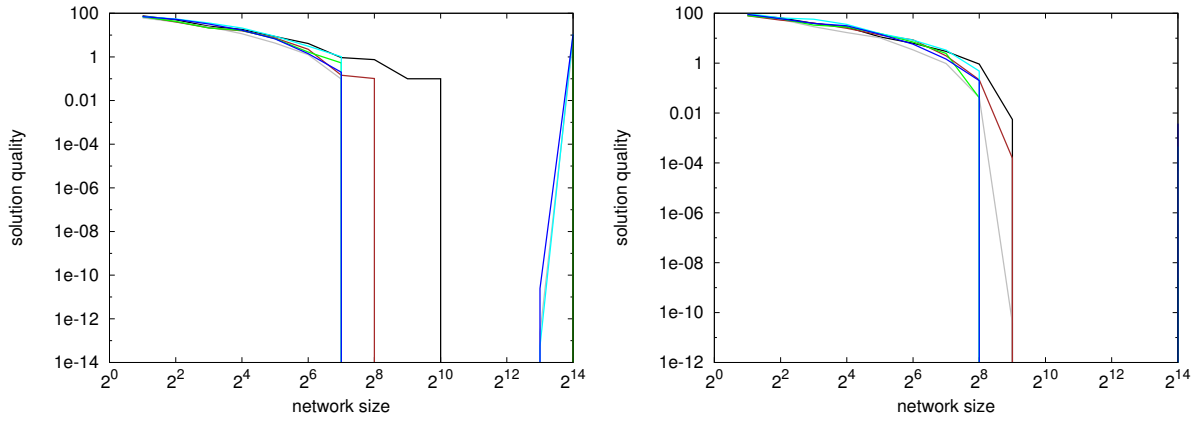
According to our experience, DE shows a higher resilience than PSO with respect of this issue. It seems that the way the individuals are combined by DE operators is more capable to provide recovering from "bad choices" or unfortunate events. Any new generation of DE individuals is not only a bunch of better candidate solutions; it is an better domain region to search within. This happens if and because individuals that increase their distance from the best one, anyway

can help by providing diversification as needed. Whereas PSO particles which are performing poorly are quite useless to the swarm. We think this fact makes DE more able to cope with different problems in those cases when a specific tuning is not possible. We are not talking about state-of-the-art solutions, but about good outcomes given a limited (or null) amount of information about the problem. It is quite understood that a "good parametrization" can change this situation quite radically.

Trying to come up with general guidelines to successfully set up our DE operators for a P2P decentralized optimization task, we may observe what follows:

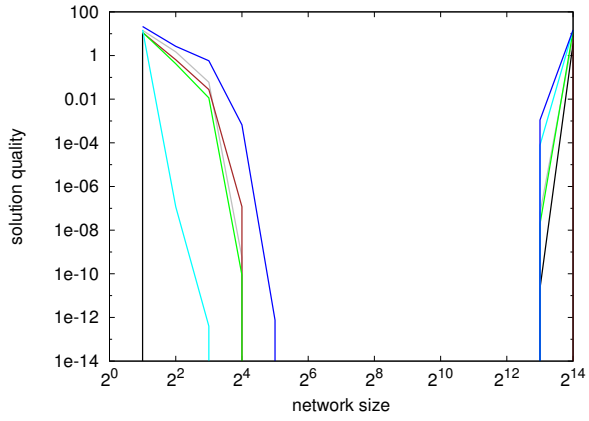
- For small networks (up to 2^6 nodes), a very small gossip rate is preferable, to avoid premature sub-optimal convergence of the local populations.
- For large networks (more than 2^{10} nodes) gossip should better occur with a probability of 0.5 or higher.
- Churn change the situation in a way that is hard to predict with respect to the relation between performances and network size. Anyway, almost in any case a high gossip rate produce better results.
- Operators biased toward exploration generally capitalize more the gossiping of sub-optimal individuals.

It is hard to state a general conclusion, because the behavior of the different operators and their absolute performance always differ as problems change. Anyway, the patterns we describe appear in almost any case and we think they show a clear correlation between performance, gossip rate and network size. More precisely, the gossiping mechanism makes the set of local populations at each node behave like a unique large global one. Anyway the effect is not simply analogous to the one we could see if we had a single node hosting all the individuals that are actually spread on our network. In that case, the generation

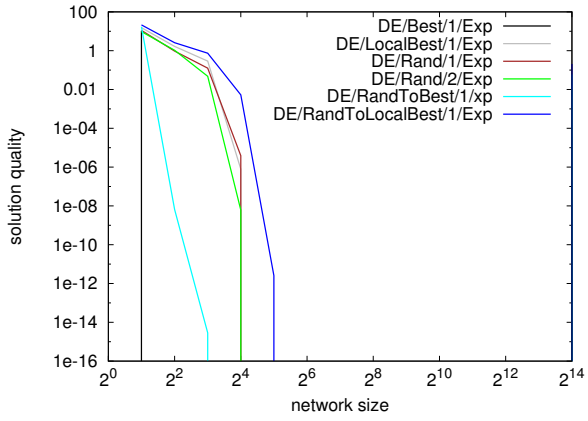


(a) $G = 0.5$ – No churn

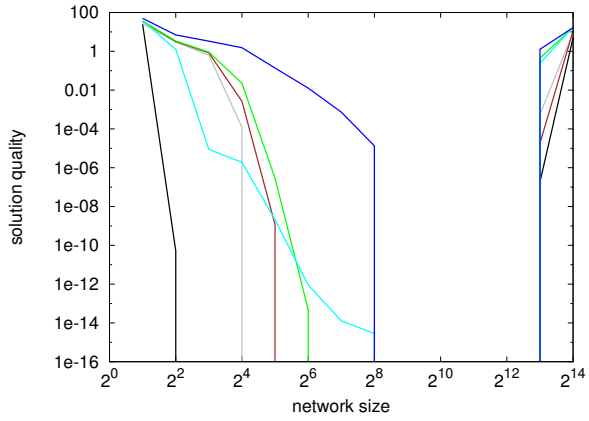
(b) $G = 1$ – No churn



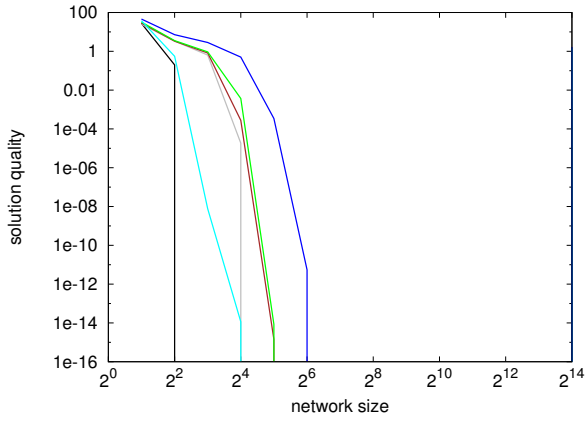
(c) $G = 0.5$ – 5% churn



(d) $G = 1$ – 5% churn

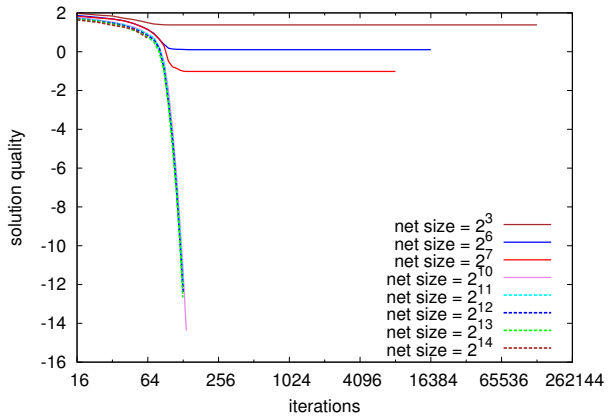


(e) $G = 0.5$ – 10% churn

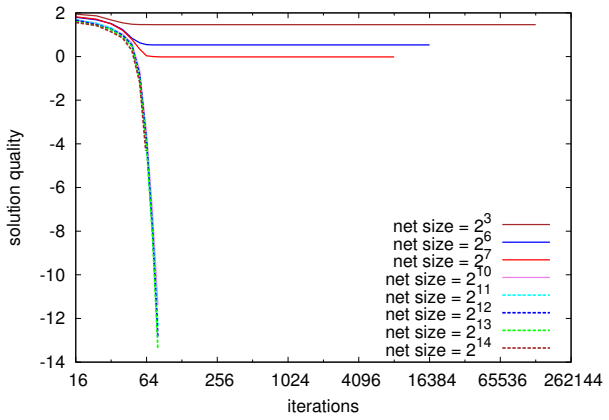


(f) $G = 1$ – 10% churn

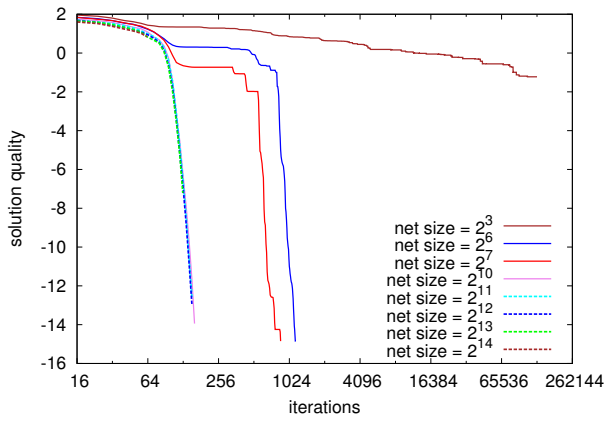
Figure 6.9: Rastrigin - How a high gossip rate G improves performance in different networks.



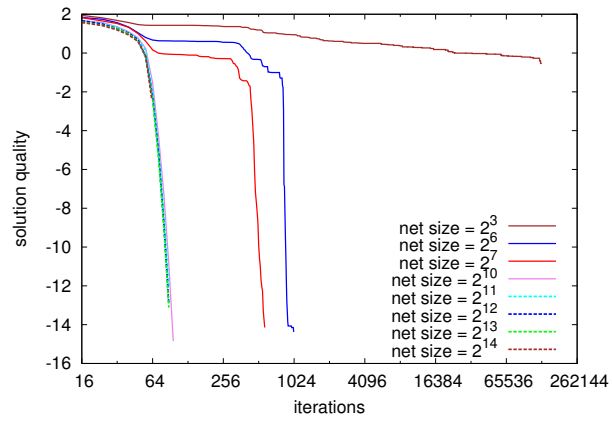
(a) $G = 0.5$ – No Churn



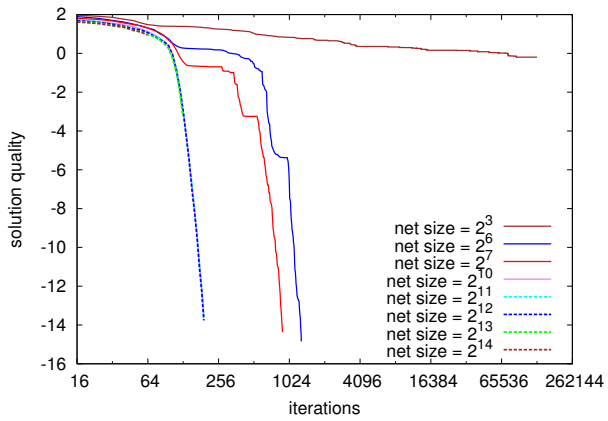
(b) $G = 1$ – No Churn



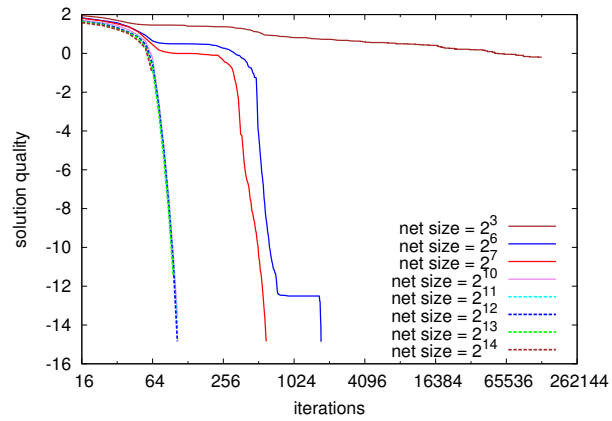
(c) $G = 0.5$ – 5% churn



(d) $G = 1$ – 5% churn



(e) $G = 0.5$ – 10% churn



(f) $G = 1$ – 10% churn

Figure 6.10: Rastrigin - DE/LocalBest/1/exp. G is the individual gossip rate

cycle length would be so huge, that the convergence of the individuals to a good outcome would require an unacceptable amount of time.

The parallel islands are meant to cope with this problem and we have already seen how the literature acknowledges their effectiveness. We have seen as well, though, that island models may suffer from premature convergence to local sub-optima, that pins down the computation to a poor performance. Among the various known migration policies that have been devised to solve this issue, we believe that our variable gossiping diffusion of sub-optimal individuals proves to be both effective and robust.

6.4 Preliminary Results towards the RAPSO meta-heuristic

In Section 4.6 we devised an experimental design of a new meta-heuristic we named RAPSO. We described three possible ways of composing this hybrid from PSO and RASH heuristics. Here we present some preliminary experiments that help us understand the dynamics of the cooperation of the two algorithms. Moreover these experiments are meant to show GOOF in action in a real distributed deployment.

We tried and evaluated our simple design (independent instances of RASH and PSO concurrently running in different nodes and sharing their best results) by tackling the already mentioned *Zakharov*, *Griewank* and *Rastrigin* functions. We assumed that every *Solver* run either one instance of `Rash_AR` (see Section 4.5) or a PSO swarm consisting of 1 particle.

We run sets of 10 independent experiments for each function, on a LAN of 50 Linux workstations. We relied on CLOUDWARE implementation of NEWSCAST to build and connect a P2P network of GOOF *SolverBoxes*. To fasten the set up and to synchronize the start of the optimization task on the various solvers, we changed the bootstrap procedure as follows. A `knownHost` waited until it received a join message from every node; then the `knownHost` sent to each

node a set of 20 node addresses, chosen uniformly at random from the collected messages. Each *SolverBox* started the optimization task only after having received the set of peers from the `knownHost`, and terminated after performing 7000 function evaluations. These setting were just meant to be a quick and easy way to deploy the tasks. We were not interested in probing the connectivity of the peer sampling protocol in this environment, nor in any related issue.

Our purpose in testing GOOF on a real LAN was mainly to assess the framework was able to provide a reliable and effective platform for the optimization task to be smoothly performed. We remind that CLOUDWARE is a multithreaded application and that the *SolverBox* runs in a dedicated thread. Threads are handled differently on different hardware and software architectures and the necessity to have a concurrent I/O activity makes a proper configuration of the application not trivial.

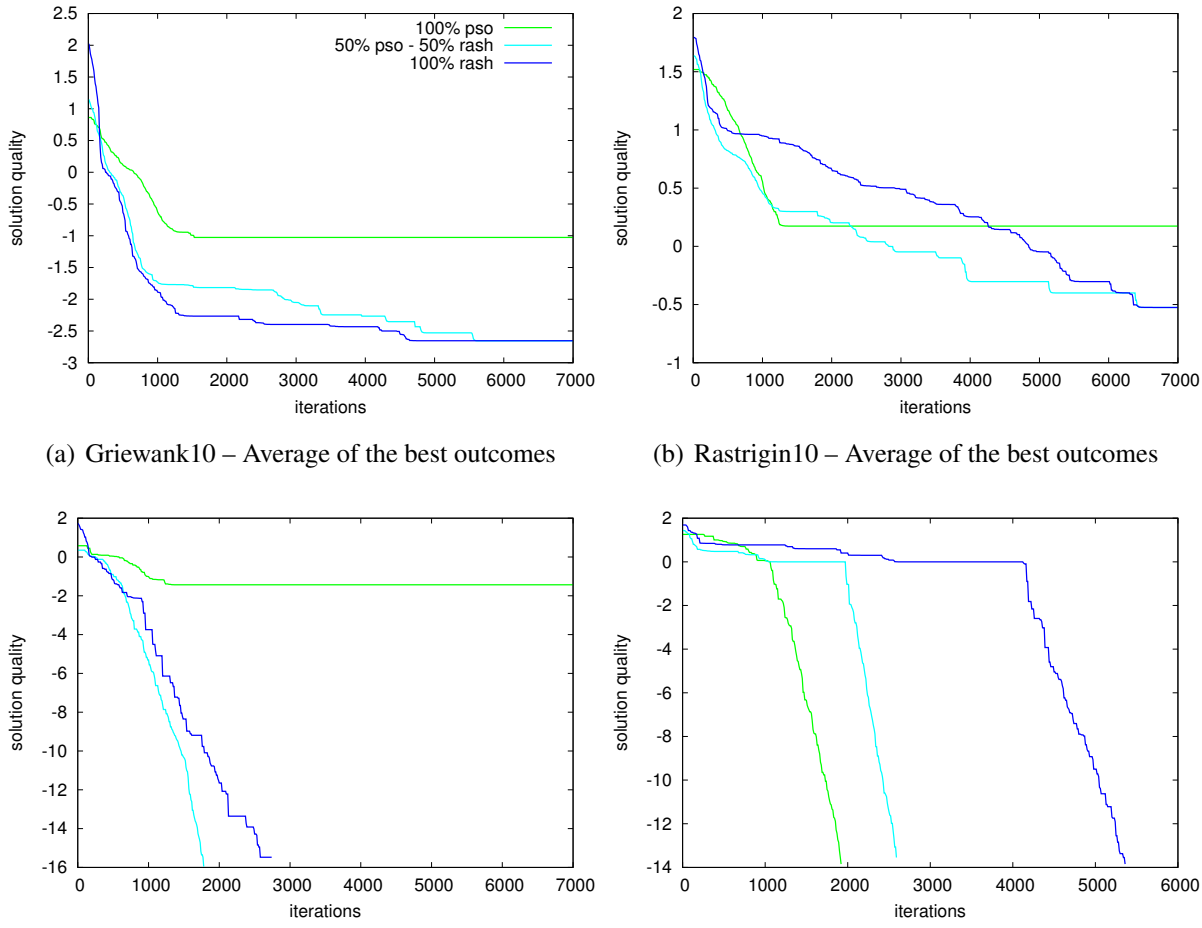
The CLOUDWARE-based implementation of GOOF proved so far to be a reliable LAN platform on which optimization tasks can be effectively performed. More tests are needed to verify the reliability of the framework in a large P2P architecture. We observe that, even if GOOF has been devised to target the latter kind of distributed environment, it provides a highly configurable and quickly deployable way of running P2P optimization experiments that can anyway suit small networks. Thus we think it can become a useful tool for academic research or to build fully decentralized overlay networks of solvers on tightly coupled architectures as well.

For the few experiments we had the opportunity to run in this real deployment, we chose to perform some prodromic tests to assess the simple interaction of PSO and RASH. Figure 6.11 shows some results. Thanks to the configuration mechanism of our *SolverBox* (see Chapter 4.2) it is easy to build a network of different solvers using identical settings at each node. The graphs shows the performance of a network in which the two heuristics are equally represented, compared with the outcomes of the same network when only one of the two is

running.

We see that, even in this very rudimental setting, RASH and PSO can actually cooperate fruitfully. The results of their cooperation are never worse, on average, than the best performing of the two. Considering the overall best results, instead of their averages, we see that their combination can actually lead to some speed up improvement. We cannot speculate further on such a limited set of experimental outcomes.

To better understand the general performance of our prototypic design, we also ran sets of experiment on PEERSIM, tackling the same functions and instan-



(a) Griewank10 – Average of the best outcomes

(b) Rastrigin10 – Average of the best outcomes

(c) Griewank10 – Overall best outcomes

(d) Rastrigin10 – Overall best outcomes

Figure 6.11: Preliminary results towards the RAPSO hybrid meta-heuristic (real deployment).

tiating the same algorithms. In all these experiments we varied the following parameters:

- **network size** (N) the number of nodes in the network
- **optimizer network composition** (C) the percentage of nodes running the first heuristic (being $100 - C$ the percentage of the nodes running the other one).

For any combination of network size N and optimizer network composition C, all solvers were assigned $2^{20}/N$ evaluations. We run 10 independent experiments for all parameter combinations using

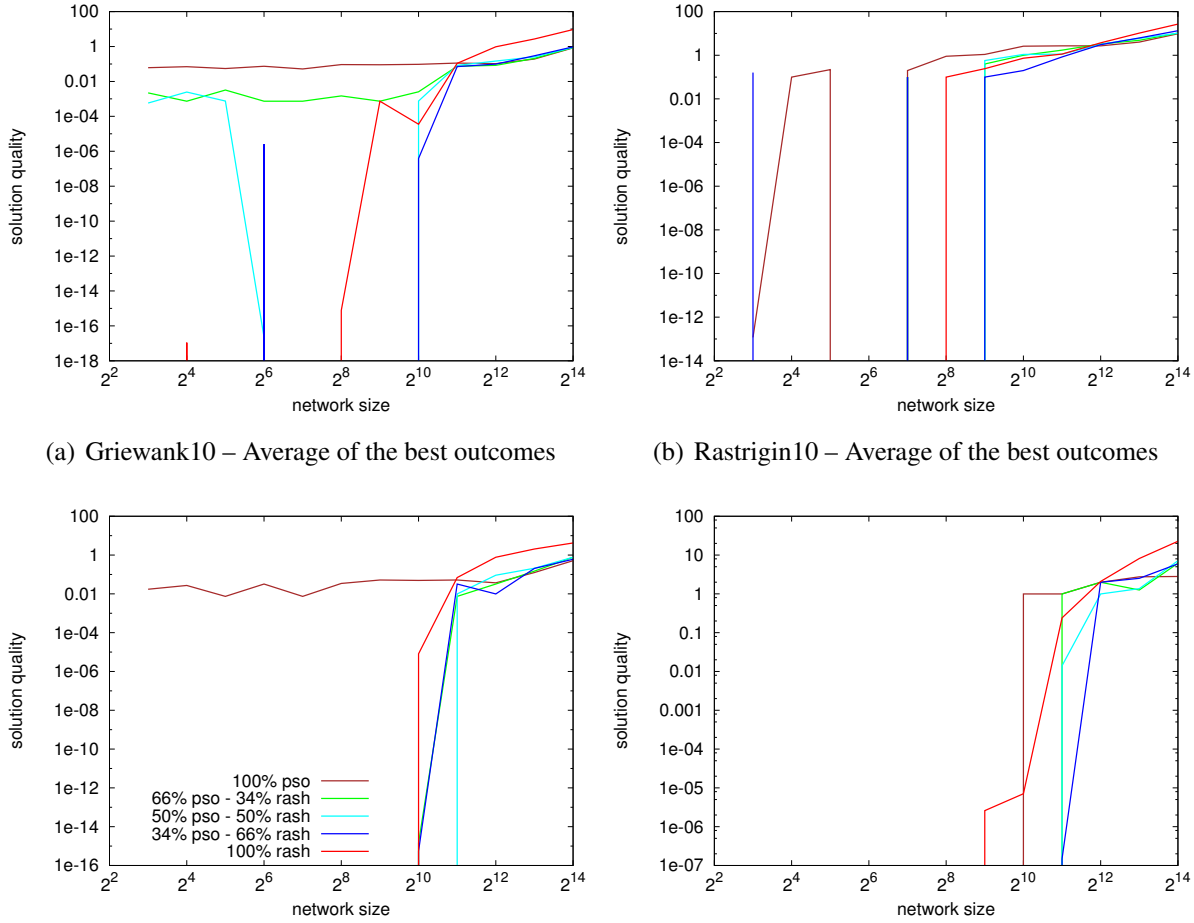
$$N \in \{2^3, \ldots, 2^{14}\} \text{ and } C \in \{100, 66, 50, 34, 0\}$$

on all test functions.

For every experiment, we traced the best solution known in the overall network after every 10 new iterations of the *Solver* in each node. Being these very preliminary tests, we considered the overall best outcomes as well as their averages also in this settings.

The results shown in Figure 6.12 confirm what we saw from our experiments in the real deployment: the combination of the two algorithms is rarely worse than the better heuristic, often faster or better. Especially as the network size grows, the cooperation of the two heuristics lead to an overall improvement. This is not an obvious result, given that the two algorithms are remarkably different in concept, design and operativity.

We also observe that the hybridization has clear limits. RASH still suffers from being occasionally attracted by sub-optimal points. But most of all, *this* PSO design is performing quite poorly. As we had already seen in [23], when designed according to the *Evolutionary Agent* paradigm [87], PSO heuristic can get easily stuck in some local optimum, thus presenting a modest performance on average. Especially with highly multimodal functions, like the ones we used



(a) Griewank10 – Average of the best outcomes

(b) Rastrigin10 – Average of the best outcomes

(c) Griewank10 – Overall best outcomes

(d) Rastrigin10 – Overall best outcomes

Figure 6.12: Preliminary results towards the RAPSO hybrid meta-heuristic (simulations). Comparison among different distributions of the RASH and PSO heuristics.

in this case, PSO agents need some restart policy to be enabled with a smart initialization, in order to achieve a good solution quality.

Thus we conclude that the approach is viable, but the design needs an effective reshape. Given the positive outcomes we could anyway see, we are allowed to think that a more sophisticated option (like the other two we proposed Section 4.6) will likely lead us to the effective new hybrid meta-heuristic we are aiming at.

Chapter 7

Conclusion

Our goal was the realization of an efficient and fully scalable way to perform optimization tasks on a decentralized, large-scale P2P network of non-dedicated clients. Many otherwise unused computational resources (personal workstation, for instance) can be exploited this way, without requiring a special infrastructure.

In order to achieve such an objective, we designed and implemented a *distributed optimization framework* both for simulated and real distributed environments. We designed and implemented a highly configurable framework, which is able to perform the execution of the optimization algorithms in a fully decentralized fashion, transparently handling all the potential networking issues as well. This framework has been plugged both in the PEERSIM emulator and in the CLOUDWARE application, that implements all the basic mechanisms of a cloud computing distributed environment, by means of epidemic protocols.

Our novel framework aims at easing the burden of performing function optimization tasks in a decentralized P2P network of solvers. Distributed optimization has already a long and rich history, but little has been done to make it exploit the (potentially) large computing facilities a reliable P2P network can provide. Our 'GOssiping Optimization Framework' (GOOF) bridges the gap between P2P services that can provide large reliable networks of interconnected

nodes and the needs of optimization practitioners who are often not able to find a reasonably simple way to run their algorithms in such a distributed environment.

We described the architecture of our framework, based on the separation between a *topology service* that maintains the network topology, an *optimization service* that performs the function optimization task and a *communication service* that enables the information sharing mechanism among the optimizers running in different nodes.

We presented the main components of the framework: the *SolverBox*, which is the main local coordinator running in each node; the *entity interfaces*, that make it possible to plug novel and legacy optimization algorithms in the P2P environment, providing a full interaction with the networking services and the communication protocol. We explained in detail how the communication protocol works and how a sophisticated peer-entity communication is provided among the nodes, making them cooperate according to the user decisions.

We tested our framework by means of both extensive simulations and in a LAN environment of tens of machines, which we built our P2P overlay network on. The experiment running in a simulated environment have been thoroughly evaluated and analyzed. We discussed how performance of various heuristics and several novel hyper-heuristics changes with respect to network conditions and parameter choices. Whichever algorithm or set of algorithms is running on a network, the size of the network, the quality of the communication, the number of concurrent instances and their local configuration are all decisive to determine the overall performance. With respect to this matter, we paid great attention to determine which issues are crucial, so to better understand to which extent we can claim our distributed design to be effective and even preferable to the others.

Finally we provided experimental results about how we used GOOF to perform novel experiments using Differential Evolution, a well-known optimiza-

tion technique. Thanks to the extreme versatility of our framework, we could easily enable a specific information sharing mechanism that has been shown to be able to greatly improve the performance of several Differential Evolution operators, working on different non-trivial functions. Furthermore, we described the prototype and gave promising preliminary results of a new meta-heuristic, that also shows the flexibility and the usefulness of GOOF in providing facilities to design and deploy distributed optimization algorithms.

Among the many possible developments we may undertake, we are working to see whether the positive impact of the information sharing mechanism we devised can be extended to other population-based optimization techniques. This will require an enrichment of GOOF, most of all in what concerns the "real deployment" of P2P fully decentralized networks of solvers.

Distributed function optimization on decentralized P2P networks is just at the beginning of a promising story. We think our results prove it is a challenge worth taking.

Bibliography

[1] Carlisle A and Dozier G. An off-the-shelf pso. In *Proceedings of the Workshop on Particle Swarm Optimization, Indianapolis, IN*, 2001.

[2] Bernardetta Addis, Andrea Cassioli, Marco Locatelli, and Fabio Schoen. Global optimization for the design of space trajectories, 2008. Optimization Online eprint archive `http://www.optimization-online.org/DB_HTML/2008/11/2150.html`.

[3] E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Diaz, I. Dorta, J. Gabarro, C. Leon, J. Luna, L. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa. Mallba: A library of skeletons for combinatorial optimisation, 2002.

[4] E. Alba and B. Dorronsoro. *Cellular Genetic Algorithms*. Springer, New York, 2008.

[5] Enrique Alba. *Parallel Metaheuristics: A New Class of Algorithms*. Wiley, 2005.

[6] Enrique Alba, Gabriel Luque, Jose Garcia Nieto, Guillermo Ordonez, and Guillermo Leguizamon. Mallba: a software library to design efficient optimisation algorithms. *Int. J. Innov. Comput. Appl.*, 1(1):74–85, 2007.

[7] David P. Anderson. Boinc: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004.

[8] Alexandre A. Andreatta, Srgio E.R. Carvalho, and Celso C. Ribeiro. An object-oriented framework for local search heuristics, 1998.

[9] Lourdes Araujo, Juan J. Merelo, Antonio Mora, and Carlos Cotta. Genotypic differences and migration policies in an island model. In *GECCO*, 2009.

[10] Maribel G. Arenas, Pierre Collet, A. E. Eiben, Márk Jelasity, Juan J. Merelo, Ben Paechter, Mike Preuß, and Marc Schoenauer. A framework for distributed evolutionary algorithms. In Juan Julián Merelo Guervós, Panagiotis Adamidis, Hans-Georg Beyer, José-Luis Fernández-Villacañas, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VII*, volume 2439 of *LNCS*, pages 665–675. Springer, 2002.

[11] D. V. Arnold and A. MacLeod. Hierarchically organised evolution strategies on the parabolic ridge. In M. Cattolico et al., editor, *GECCO 06: Proceedings of the 2006 Genetic and Evolutionary Computation Conference*, page 437444, 2006.

[12] Dirk V. Arnold and Anthony S. Castellarin. A novel approach to adaptive isolation in evolution strategies. In *GECCO*, 2009.

[13] M. Senthil Arumugam, M. V. C. Rao, and Alan W. C. Tan. A novel and effective particle swarm optimization like algorithm with extrapolation technique. *Appl. Soft Comput.*, 9(1):308–320, 2009.

[14] Salman A. Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. 2004.

[15] R. Battiti. Reactive search: Toward self-tuning heuristics. In V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors, *Modern Heuristic Search Methods*, chapter 4, pages 61–83. John Wiley and Sons Ltd, 1996.

[16] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[17] Roberto Battiti, Mauro Brunato, and Franco Mascia. *Reactive Search and Intelligent Optimization*. Springer Publishing Company, Incorporated, 2008.

[18] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services, 2004.

[19] Ahcene Bendjoudi, Nouredine Melab, and El-Ghazali Talbi. A parallel P2P branch-and-bound algorithm for computational grids. In *Proc. of IEEE CCGRID*, pages 749–754, Rio de Janeiro, Brazil, 2007.

[20] Engelbrecht AP Bergh vdF. Effects of swarm size on cooperative particle swarm optimizers. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) San Francisco, USA*, 2001.

[21] Johan Berntsson. G2dga: an adaptive framework for internet-based distributed genetic algorithms. In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 346–349, New York, NY, USA, 2005. ACM.

[22] Marco Biazzini, Balázs Bánhelyi, Alberto Montresor, and Márk Jelasity. Distributed hyper-heuristics for real parameter optimization. In *GECCO*, pages 1339–1346, 2009.

[23] Marco Biazzini, Balázs Bánhelyi, Alberto Montresor, and Márk Jelasity. Peer-to-peer optimization in large unreliable networks with branch-and-bound and particle swarms. In Mario Giacobini,

Anthony Brabazon, Stefano Cagnoni, Gianni A. Di Caro, Anikó Ekárt, Anna Isabel Esparcia-Alcázar, Muddassar Farooq, Andreas Fink, and Penousal Machado, editors, *Applications of Evolutionary Computing*, pages 87–92. Springer, 2009.

[24] Marco Biazzini, Alberto Montresor, and Mauro Brunato. Towards a decentralized architecture for optimization. In *Proc. of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, FL, USA, April 2008.

[25] M. Birattari, L. Paquete, T. Stutzle, and K. Varrentrap. Classification of metaheuristics and design of experiments for the analysis of components. AIDA 2001-05, Intellektik, Technische Universitat Darmstadt, Germany, 2001.

[26] Jürgen Branke, Michael Stein, and Hartmut Schmeck. A unified framework for metaheuristics. In *GECCO*, pages 1568–1569, 2003.

[27] Mauro Brunato and Roberto Battiti. The reactive affine shaker: a building block for minimizing functions of continuous variables. Technical report, Tech. Report DIT-06-012, University of Trento, 2006.

[28] Mauro Brunato, Roberto Battiti, and Alberto Montresor. Gosh! gossiping optimization search heuristics. In *Proceedings of the Learning and Intelligent Optimization Workshop (LION 2007)*, 2007.

[29] Mauro Brunato, Roberto Battiti, and Srinivas Pasupuleti. A memory-based rash optimizer. In Ariel Felner, Robert Holte, and Hector Geffner, editors, *Proceedings of AAAI-06 workshop on Heuristic Search, Memory Based Heuristics and Their applications*, pages 45–51, Boston, Mass., 2006. ISBN 978-1-57735-290-7.

[30] E. K Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, , and R. Qu. A survey of hyper-heuristics. Computer Science Technical Report NOTTCS-TR-SUB-0906241418-2747, School of Computer Science and Information Technology, University of Nottingham, 2009.

[31] E. K. Burke, G. Kendall, and E. Soubeiga. A tabu-search hyperheuristic for timetabling and rostering. *Journal of Heuristics*, 9(6):451–470, 2003.

[32] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In *Handbook of Metaheuristics*, pages 457–474. 2003.

[33] M. Hyde G. Kendall G. Ochoa E. Ozcan Burke, E. K and J. Woodward. *A Classification of Hyper-heuristics Approaches*, volume Handbook of Metaheuristics of *International Series in Operations Research & Management Science*. Springer, 2009.

[34] S. Cahon, N. Melab, and E.-G. Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[35] E. Cantú-Paz. Migration policies, selection pressure and parallel evolutionary algorithms. Journal of Heuristics, 7(4), 2001.

[36] Erick Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Springer, 2000.

[37] L. G. Casado, J. A. Martinez, I. Garcia, and E. M. T. Hendrix. Branch-and-bound interval global optimization on shared memory multiprocessors. *Optimization Methods and Software*, 23(5):689–701, 2008.

[38] A. Colorni, M. Dorigo, and V. Manniezzo. Distributed optimization by ant colonies. In F.J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life (ECAL-91)*, page 134142. The MIT Press. Cambridge MA., 1992.

[39] J. Costa, N. Lopes, and P. Silva. Jdeal: The java distributed evolutionary algorithms library, 1999.

[40] A. Das and B. K. Chakrabarti. *Quantum Annealing and Related Optimization Methods*. 2005.

[41] Juan de Vicente, Juan Lanchares, and Romn Hermida. Placement by thermodynamic simulated annealing. *Physics Letters A*, 317(5-6):415 – 423, 2003.

[42] Abdallah Deeb, I. Al Zain, Phil Trinder, and Greg Michaelson (supervisors. Implementing high-level parallelism on computational grids. Technical report, 2006.

[43] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12. ACM Press, August 1987.

[44] J. Denzinger and T. Oerman. On cooperation between evolutionary algorithms and other search paradigms. In *Proceedings of Congress on Evolutionary Computation (CEC'1999)*, 1999.

[45] G Dueck and T Scheuer. Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing. *J. Comp. Physics*, (90), 1990.

[46] E. A. Eiben, M. Schoenauer, J. L. J. Laredo, P. A. Castillo, A. M. Mora, and J. J. Merelo. Exploring selection mechanisms for an agent-based distributed evolutionary algorithm. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2801–2808, New York, NY, USA, 2007. ACM.

[47] Alvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michele Sebag. Dynamic multi-armed bandits and extreme value-based rewards for adaptive operator selection in evolutionary algorithms. In Springer, editor, *Proceedings of Learning and Intelligent OptimizatioN (LION) 3*, 2009.

[48] Ian Foster. The anatomy of the grid: Enabling scalable virtual organizations. *INTERNATIONAL JOURNAL OF SUPERCOMPUTER APPLICATIONS*, 15(3):2001, 2001.

[49] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[50] Matteo Gagliolo and Jurgen Schmidhuber. Dynamic algorithm portfolios. In *AIMATH*, 2006.

[51] Matteo Gagliolo and Jürgen Schmidhuber. Learning restart strategies. In *IJCAI 2007 — Twentieth International Joint Conference on Artificial Intelligence*, January 2007. To appear.

[52] Christian Gagne, Marc Parizeau, and Marc Dubreuil. Distributed beagle: An environment for parallel and distributed evolutionary computations. 2003.

[53] Luca Di Gaspero and Andrea Schaerf. Easylocal++: An object-oriented framework for flexible design of local search algorithms, 2000.

[54] F. Glover. Tabu thresholding: Improved search by nonmonotonic trajectories. to appear in ORSA Journal on Computing.

[55] F. Glover. Tabu search. 1987. draft of paper presented at ORSA/TIMS Joint National Meeting, St. Louis.

[56] F. Glover, J. P. Kelly, and M. Laguna. Genetic algorithms and tabu search: hybrids for optimization. 1992.

[57] F. Glover and M. Laguna. Tabu search. In Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, volume C, pages 70–141. Blackwell Scientific Publishing, 1993.

[58] Carla P. Gomes and Bart Selman. Practical aspects of algorithm portfolio design. In *In Proc. of Third ILOG International Users Meeting*, pages 200–1, 1997.

[59] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001.

[60] V. Granville, M. Krivánek, and J.P. Rasson. Simulated annealing: A proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:652–656, 1994.

[61] Eric Hand. Head in the clouds. *Nature*, 449:963, 2007.

[62] Cdric Hartland, Sylvain Gelly, Nicolas Baskiotis, Olivier Teytaud, and Michele Sebag. Multi-armed bandit, dynamic environments and meta-bandits, 2006.

[63] J. Ignacio Hidalgo, Juan Lanchares, Francisco Fernández de Vega, and Daniel Lombra na. Is the island model fault tolerant? In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2737–2744, New York, NY, USA, 2007. ACM.

[64] J.H. Holland. Adaptation in natural and artificial systems. The University of Michigan Press., 1975.

[65] Myles Hollander and Douglas A. Wolfe. *Nonparametric Statistical Methods*. Wiley, 2nd edition, 1999.

[66] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, 2005.

[67] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, January 3 1997.

[68] A. Iamnitchi and I. Foster. On death, taxes, and the convergence of peer-to-peer and grid computing. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems IPTPS*, volume 3, 2003.

[69] S.H. Jacobson and E. Ycesan. Global optimization performance measures for generalized hill climbing algorithms. Journal of Global Optimization 29, 2004.

[70] Márk Jelasity and Ozalp Babaoglu. T-Man: Gossip-based overlay topology management. In *Engineering Self-Organising Systems: Third International Workshop (ESOA 2005)*, volume 3910 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2006.

[71] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *Middleware 2004*, volume 3231 of *Lecture Notes in Computer Science*, pages 79–98. Springer-Verlag, 2004.

[72] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In Giovanna Di Marzo Serugendo, Anthony Karageorgos, Omer F. Rana, and Franco Zambonelli, editors, *Engineering Self-Organising Systems*, volume 2977 of *LNAI*, pages 265–282. Springer, 2004. invited paper.

[73] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, August 2005.

[74] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[75] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8, August 2007.

[76] Joshy Joseph and Craig Fellenstein. *Grid Computing*. Prentice Hall, 2003.

[77] Ming-Yang Kao, Yuan Ma, Michael Sipser, and Yiqun Yin. Optimal constructions of hybrid algorithms. In *SODA*, pages 372–381, 1994.

[78] Parsopoulos KE and Vrahatis MN. Recent approaches to global optimization problems through particle swarm optimization. Natural Computing 2002, 1, 2002.

[79] J. Kennedy and R. C. Eberhart. Particle swarm optimization. *IEEE Int. Conf. Neural Networks*, pages 1942–1948, 1995.

[80] James Kennedy. Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance. In *Proc.* 1^{st} *Congress on Evolutionary Computation (CEC'99)*, pages 1931–1938, 1999.

[81] James Kennedy. Stereotyping: Improving particle swarm performance with cluster analysis. In *Proc.* 2^{nd} *Congress on Evolutionary Computation (CEC'00)*, pages 1507–1512, 2000.

[82] James Kennedy and Rui Mendes. Population structure and particle swarm performance. In *Proc.* 4^{th} *Congress on Evolutionary Computation (CEC'02)*, pages 1671–1676, May 2002.

[83] Anne-Marie Kermarrec and Maarten van Steen, editors. *ACM SIGOPS Operating Systems Review 41*. October 2007. Special issue on Gossip-Based Networking.

[84] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[85] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[86] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[87] J. L. J. Laredo, P. A. Castillo, A. M. Mora, and J. J. Merelo. Evolvable agents, a fine grained approach for distributed evolutionary computing: walking towards the peer-to-peer computing frontiers. *Soft Comput.*, 12(12):1145–1156, 2008.

[88] J. L. J. Laredo, E. A. Eiben, M. Schoenauer, P. A. Castillo, A. M. Mora, F. Fernandez, and J. J. Merelo. Self-adaptive gossip policies for distributed population-based algorithms. 2007.

[89] J. L. J. Laredo, E. A. Eiben, Maarten van Steen, P. A. Castillo, A. M. Mora, and J. J. Merelo. P2P evolutionary algorithms: A suitable approach for tackling large instances in hard optimization problems. In *Proc. of Euro-Par*, volume 5168 of *LNCS*, pages 622–631. Springer-Verlag, 2008.

[90] J.L.J. Laredo, P.A. Castillo, A.M. Mora, and J.J. Merelo. Exploring population structures for locally concurrent and massively parallel evolutionary algorithms. In *Evolutionary Computation,*

2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on, pages 2605–2612, June 2008.

[91] J.L.J. Laredo, P.A. Castillo, A.M. Mora, J.J. Merelo, and C. Fernandes. Resilience to churn of a peer-to-peer evolutionary algorithm. International Journal of High Performance Systems Architecture, 2008. Volume 1, Number 4.

[92] Juan Luís Laredo, Pedro Angel Castillo, Ben Paechter, Antonio Miguel Mora, Eva Alfaro-Cid, Anna I. Esparcia-Alcázar, and Juan Julián Merelo. Empirical validation of a gossiping communication mechanism for parallel eas. In *Proceedings of the 2007 EvoWorkshops 2007 on EvoCoMnet, EvoFIN, EvoIASP,EvoINTERACTION, EvoMUSART, EvoSTOC and EvoTransLog*, pages 129–136, Berlin, Heidelberg, 2007. Springer-Verlag.

[93] Juan Luis J. Laredo, Carlos Fernandes, Juan Julin Merelo, and Christian Gagné. Improving genetic algorithms performance via deterministic population shrinkage. In *GECCO*, 2009.

[94] Coromoto Leòn, Gara Miranda, and Carlos Segura. Parallel hyperheuristic. a self-adaptive island-based model for multi-objective optimization. In *Proceedings of GECCO08*, 2008.

[95] S. Luke, L. Panait, J. Bassett, R. Hubley, C. Balan, and A. Chircop. Ecj: A java-based evolutionary computation and genetic programming research system, 2002.

[96] Sean Luke. *Essentials of Metaheuristics*. 2009. available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[97] M. Lunacek and D. Whitley. Searching for balance: Understanding self-adaptation on ridge functions. In T. P. Runarsson et al., editor, *Parallel Problem Solving from Nature PPSN IX*, page 8291. Springer Verlag, 2006.

[98] Clerc M and Kennedy J. The particle swarm explosion, stability, and convergence in a multidimensional complex space. IEEE Transactions on Evolutionary Computation, 2002.

[99] Malik Magdon-Ismail and Amir F. Atiya. The early restart. *Neural Computation*, 2000.

[100] Oded Maron and Andrew W. Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11:193–225, 1997.

[101] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems: First International Workshop*, 2002.

[102] Michael Meissner, Michael Schmuker, and Gisbert Schneider. Optimized particle swarm optimization (opso) and its application to artificial neural network training. http://www.biomedcentral.com/1471-2105/7/125, 2006.

[103] N. Melab, M. Mezmaz, and E-G. Talbi. Parallel hybrid multi-objective island model in peer-to-peer environment. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 6*, page 190.2, Washington, DC, USA, 2005. IEEE Computer Society.

[104] Rui Mendes, James Kennedy, and José Neves. The fully informed particle swarm: Simpler, maybe better. *IEEE Transactions on Evolutionary Computation*, 8(3):204–210, June 2004.

[105] Alberto Montresor. Cloudware. `http://cloudware.sf.net`.

[106] Alberto Montresor. Intelligent gossip. In *IDC*, pages 3–10, 2008.

[107] Pablo Moscato. A gentle introduction to memetic algorithms. In *Handbook of Metaheuristics*, volume 57, pages 105–144. Kluwer Academic Publishers, 2003.

[108] Pablo Moscato, La Plata, La Plata, and Michael G. Norman. A "memetic" approach for the traveling salesman problem implementation of a computational ecology for combinatorial optimization on message-passing systems. In *In Proceedings of the International Conference on Parallel Computing and Transputer Applications*, pages 177–186. IOS Press, 1992.

[109] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 1999.

[110] Stephan Olariu and Albert Y. Zomaya. *Handbook Of Bioinspired Algorithms And Applications (Chapman & Hall/Crc Computer & Information Science)*. Chapman & Hall/CRC, 2005.

[111] Djamila Ouelhadj and Sanja Petrovic. A cooperative distributed hyper-heuristic framework for scheduling. In *Proc. of the IEEE Int conference on Systems, Man and Cybernetics (SMC 2008)*, Singapore, 2008.

[112] Ender Özcan, Burak Bilgin, and Emin Erkan Korkmaz. A comprehensive analysis of hyper-heuristics. *Intelligent Data Analysis*, 12(1):3–23, 2008.

[113] Boris Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, February 1987.

[114] Helmut Ratschek and Jon Rokne. Interval methods. In R. Horst and P. M. Pardalos, editors, *Handbook of Global Optimization*. Kluwer, 1995.

[115] Prapa Rattadilok, Andy Gaw, and Raymond S.K. Kwan. Distributed choice function hyper-heuristics for timetabling and scheduling. In *Practice and Theory of Automated Timetabling PATAT V*, number 3616 in LNCS, pages 51–67. Springer, 2005.

[116] C. R. Reeves. Diversification in genetic algorithms: Some connection with tabu search. Technical report, Coventry University, United Kingdom, 1993.

[117] C.R. Reeves and J.E. Rowe. Genetic algorithms principles and perspectives. Kluwer. Boston, 2003.

[118] Andrea Roli and Michela Milano. Magma: A multiagent architecture for metaheuristics. *IEEE Trans. on Systems, Man and Cybernetics - Part B*, 34:2004, 2002.

[119] Juan Romero and Carlos Cotta. Optimization by island-structured decentralized particle swarms. In *Fuzzy Days*, pages 25–33, 2004.

[120] Gunter Rudolph. Massively parallel simulated annealing and its relation to evolutionary algorithms. *Evolutionary Computation*, 1(4):361–383, 1994.

[121] Tahir Sag and Mehmet unkas. A tool for multiobjective evolutionary algorithms. *Advances in Engineering Software*, 40(9):902 – 912, 2009.

[122] Tzur Sayag, Shai Fine, and Yishay Mansour. *Combining Multiple Heuristics*, pages 242–253. Springer Berlin / Heidelberg, 2006.

[123] I. Scriven, A. Lewis, D. Ireland, , and J. Lu. Distributed multiple objective particle swarm optimisation using peer to peer networks. In *IEEE Congress on Evolutionary Computation (CEC)*, 2008.

[124] I. Scriven, A. Lewis, and S. Mostaghim. Dynamic search initialisation strategies for multi-objective optimisation in peer-to-peer networks. *IEEE Congress on Evolutionary Computation, CEC '09*, pages 1515 – 1522, 2009.

[125] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-peer-systems and applications*. Number 3485 in Lecture Notes in Computer Science. Springer, 2005.

[126] J. Stender, editor. *Parallel Genetic Algorithm: Theory & Applications*. IOS Press, 1993.

[127] Rainer Storn and Kenneth Price. Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, December 1997.

[128] Matthew Streeter and Stephen F. Smith. New techniques for algorithm portfolio design. In *UAI*, 2008.

[129] Felix Streichert and Holger Ulmer. JavaEvA - a java framework for evolutionary algorithms. Technical Report WSI-2005-06, Centre for Bioinformatics Tübingen, University of Tübingen, 2005.

[130] E.D. Taillard, L.M. Gambardella, M. Gendreau, and J-Y. Potvin. Adaptive memory programming: A unified view of metaheuristics. In *European Journal of Operational Research 135*, pages 1–16. 2000.

[131] E. G. Talbi. *Parallel Combinatorial Optimization*. John Wiley and Sons, USA, 2006.

[132] El-Ghazali Talbi. *Metaheuristics: From design to Implementation*. Wiley, 2009.

[133] Marco Tomassini. *Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time*. Springer, 2005.

[134] M. G. A. Verhoeven and E. H. L. Aarts. Parallel local search. *Journal of Heuristics*, pages 43–65, 2006.

[135] T. Vinkó and D. Izzo. Learning the best combination of solvers in a distributed global optimization environment. In *Proceedings of AGO 2007*, pages 13–17, Mykonos, Greece, 2007.

[136] Stefan Vo and David Woodruff, editors. *Optimization Software Class Libraries*. Springer, 2002.

[137] Spyros Voulgaris, Mark Jelasity, and Maarten Van Steen. A robust and scalable peer-to-peer gossiping protocol, 2003.

[138] S. Wagner and M. Affenzeller. *A Generic and Extensible Optimization Environment*, pages 538–541. Springer Vienna, 2005.

[139] Darrell Whitley, Soraya Rana, and Robert B. Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7:33–47, 1998.

[140] W. R. M. U. K. Wickramasinghe, M. van Steen, and A. E. Eiben. Peer-to-peer evolutionary algorithms with adaptive autonomous selection. In *Proc. of GECCO*, pages 1460–1467. ACM Press New York, NY, USA, 2007.

[141] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 1(1):67–82, 1997.

[142] Shi Y and Eberhart R. C. Parameter selection in particle swarm optimization. In *In Evolutionary Programming VII: Proceedings of the Seventh Annual Conference on Evolutionary Programming New York, USA*, pages 591–600, 1998.

[143] Yoji Yamato and Hiroshi Sunaga. Peer-to-peer non-document content searching method using user evaluation of semantic vectors. *IEICE - Transactions on Communications*, E89-B(9):2309–2318, 2006.

[144] Lee Breslau Nick Lanham Yatin Chawathe, Sylvia Ratnasamy and Scott Shenker. Making gnutella-like p2p systems scalable, 2003.

[145] Jinhua Zhang, Jian Zhuang, Haifeng Du, and Sun'an Wang. Self-organizing genetic algorithm based tuning of pid controllers. *Inf. Sci.*, 179(7):1007–1018, 2009.

Appendix A : Interface *Solver*

```
package goof.core;


/**
 * A Solver is the core component of the framework, the one that implements the
 * algorithmic procedures to evaluate the {@link Function}, keep track of the results
 * and move toward the minimum. The internal complexity of a Solver may vary greatly,
 * depending on the implementer's choices. It may be worth having a 'very basic' Solver
 * that repeatedly apply a given {@link Algo} (in which the real complexity resides)
 * and simply keep track of the best solutions found so far. Or the implementer could feel it
 * better to concentrate everything in the Solver and even make no use of any {@link Algo}
 * at all.<br>
 * This interface leaves the greatest freedom to the implementer. He's allowed to shape the
 * solver in the most suitable fashion, as long as these method are properly implemented.
 * Some method may have post-conditions the implementer has to guarantee, as specified
 * in the methods' documentation.<br>
 * A Solver may need to fetch data from a configuration file. Then can be initialized
 * using the {@link #init} method, by which any information about the
 * {@link Function} to be optimized can be used. This initialization is performed exactly
 * once, right after the object instantiation.
 *
 * @author Marco Biazzini
 * @version 1.0
 */
public interface Solver {

/**
 * Initialize the solver as needed, using data provided by the {@link Function}
 * that will be evaluated. This method is called exactly <b>once</b> by the
 * {@link SolverBox}, right after the instantiation of the {@link Solver} object.
 * @param f {@link Function} to be evaluated.
 */
public void init(Function f);


/**
 * Perform the next iteration of the optimization task, including one or more
 * {@link Function} evaluations, using the {@link Algo} that would be returned by
 * {@link #getCurrentAlgo()}, if defined. Whenever this method returns,
 * a consistent behavior of {@link #getBestAlgo}, {@link #getBestTimestamp},
 * {@link #getBestValue}, {@link #getBest}, {@link #getCurrentTimestamp},
 * {@link #getCurrentValue}, {@link #getCurrent} must be guaranteed.
 * @param f {@link Function} to be evaluated.
 * @return {@code true} if this iteration improves the current best result;
 *  {@code false} otherwise.
```

```
 */
public boolean next(Function f);


/**
 * This method is called whenever the {@link SolverBox} asks the {@link Solver} for
 * data to be sent to another node in the network. The returned data will be the
 * argument of {@link Solver#update(Object[])} method of a {@link Solver} object of
 * the same class in the destination node.
 * @return An array containing data to be shared with other {@link Solver}s
 * of the same kind; {@code null} if no data needs to be sent.
 */
public Object[] getDataToSend();


/**
 * Update the {@link Solver} with data coming from a {@link Solver} of the same class
 * (running on another node in the network), at the beginning of each iteration in
 * the {@link SolverBox}.
 * @param data Array of data to share among {@link Solver}s that are alike.
 */
public void update(Object[] data);


/**
 * Update with new best results as soon as they are available (by gossip) in the
 * {@link SolverBox}.
 * They could have been obtained by means of a different kind of {@link Solver}.
 * @param boxBestValue Best result (minimum)
 * @param boxBest Vector of coordinates of the minimum
 */
public void update(double boxBestValue, double[] boxBest);


/**
 * Provides the {@link Algo} which this {@link Solver} is currently using, if any.
 * If you call this method right after {@link #setCurrentAlgo}, it must return an
 * {@link Algo} which is identical to the latter's argument.
 * @return The {@link Algo} currently used by the {@link Solver}, if any;
 * {@code null} otherwise.
 */
public Algo getCurrentAlgo();


/**
 * This method is used by a {@link Meta} to change the {@link Algo} in use.
 * It sets the argument as the {@link Algo} to be used in the next step, if any.
 * @param a The {@link Algo} to be used in the next step.
 */
public void setCurrentAlgo(Algo a);


/**
 *
```

```
     * @return The best performing {@link Algo}, if any; {@code null} otherwise.
     */
    public Algo getBestAlgo();


    /**
     *
     * @return The current best result (minimum)
     */
    public double getBestValue();


    /**
     *
     * @return The vector of coordinates of the currently known minimum of the
     * {@link Function}
     */
    public double[] getBest();


    /**
     * It gives the number of {@link Function} evaluations performed in the current iteration.
     * Notice that, if the actual solver uses a instance of {@link Algo}, the value returned by
     * this method must be consistent with the value return by {@link Algo#getTime}.
     * @return The number of {@link Function} evaluations performed in the latest
     * execution of the method {@link Solver#next}.
     */
    double getTime();


    /**
     *
     * @return The timestamp as defined for the actual {@link Solver}, in which the
     * currently best known value has been found, cast in a long.
     */
    public long getBestTimestamp();


    /**
     *
     * @return The current timestamp as defined for the actual {@link Solver},
     * cast in a long.
     */
    public long getCurrentTimestamp();


    /**
     * @return The latest value found in a {@link Function} evaluation.
     */
    public double getCurrentValue();


    /**
     *
```

```
     * @return The vector of coordinates of the latest evaluated point in the domain of
     * the {@link Function}.
     */
    public double[] getCurrent();


    /**
     * It gives the probability to send a message to another {@link Solver} in the network.
     * @return A real between 0.0 and 1.0 (both included).
     */
    public double getGossipProb();

}
```

Appendix B : Interface *Algo*

```
package goof.core;


/**
 * An {@link Algo} is a basic algorithms whose designed requires less effort and
 * complexity than a {@link Solver}. Moreover, it can be used by a {@link Solver} to
 * perform a part of its task.<br>
 * This interface allows the implementation of reusable procedures that can be exchanged
 * with one another (paying attention all of them are compatible with the {@link Solver}),
 * e.g. building blocks or heuristics which a Solver can choose among or which a
 * {@link Meta} can select and assign to the {@link Solver} at each iteration.
 * The {@link Algo#apply} method signature is purposely very generic, so that an
 * {@link Algo} can be designed to fulfill any purpose a given {@link Solver} may need
 * it for.
 * @author Marco Biazzini
 * @version 1.0
 */
public interface Algo {

/**
 *
 * @return The number of {@link Function} evaluations performed in a single
 * execution of the method {@link Algo#apply}; 0 if no function evaluation is
 * performed.
 */
public double getTime();

/**
 * Perform the next iteration of the algorithm's task.
 * @param arg It contains all that is needed to perform the task. This array is
 * provided by the solver that hosts the {@link Algo}.
 * @return The coordinates vector of the next point to be evaluated or, if the
 * {@link Algo} itself performs the function evaluation, a vector of values that are
 * meaningful to the {@link Solver} that hosts it.
 */
public double[] apply(Object[] arg);


/**
 * It is required for the {@link Algo} to have a unique identifier (integer) that is
 * assigned by the caller object, by means of this method.
 * @param ind The identifier to be assigned.
 */
public void setIndex(int ind) ;
```

```
/**
 *
 * @return The identifier assigned to this {@link Algo}.
 */
public int getIndex();
}
```

Appendix C : Interface *Meta*

```
package goof.core;


/**
 * This interface can be useful to plug in the framework a meta- or hyper-heuristics,
 * a racing algorithm, a portfolio selector etc. A {@link Meta} object is a component
 * whose goal is to choose among different {@link Algo} instances and assign the chosen
 * one to the {@link Solver}, for it to be used in the next iteration. No {@link Function}
 * evaluation is usually required to perform such a task.<br>
 * Whenever a new {@link SolverBox} iteration takes place, the {@link Meta} is updated
 * with the freshest data available from the solver, then the {@link Meta#apply}
 * method is called, to know the {@link Algo} to be used in the current {@link Solver}
 * iteration. Thus a {@link Meta} must have a 'start-from-scratch' policy to assign
 * an algo at first.<br>
 * This interface leaves the implementer free to choose how to handle the pool of
 * available {@link Algo} instances as well as any functional details.
 * @author Marco Biazzini
 * @version 1.0
 */
public interface Meta {

/**
 * Updates the {@link Meta} as required, with the freshest data available from
 * the local {@link Solver}. Note that these data are not dependent on the latest
 * gossip updates sent to the local {@link Solver}.
 * @param index The same index you can get from the {@link Algo#getIndex} method
 *  of the {@link Algo} which these arguments refer to.
 * @param value The latest function value found by the local {@link Solver}, as
 *  returned by {@link Solver#getCurrentValue}.
 * @param position The latest function point evaluated by the local {@link Solver},
 * as returned by {@link Solver#getCurrent}.
 * @param ts The timestamp which refers to the latest value found by the
 * {@link Solver}, as returned by {@link Solver#getCurrentTimestamp}.
 */
public void updateStats(int index, double value, double[] position, long ts);


/**
 * Perform an decisional step in which the next {@link Algo} for the local
 * {@link Solver} to use is chosen.
 * @return The {@link Algo} to be used in the next iteration by the {@link Solver}.
 */
public Algo apply();
```

```
/**
 * This method is called whenever the {@link SolverBox} asks the {@link Meta} for
 * data to be sent to another node in the network. The returned data will be the
 * argument of {@link Meta#update} method of a {@link Meta} object of the same class
 * in the destination node.
 * @return An array containing data to be shared with other {@link Meta}s
 * of the same kind; {@code null} if no data needs to be sent.
 */
public Object[] getDataToSend();


/**
 * Update with data coming from a {@link Meta} object of the same class
 * (running on another node in the network), at the beginning of each iteration in
 * the {@link SolverBox}.
 * @param data Array of data to share among {@link Meta}s that are alike.
 */
public void update(Object[] data);


/**
 * It gives the probability to send a message to another {@link Meta} in the network.
 * @return A real between 0.0 and 1.0 (both included).
 */
public double getGossipProb();

}
```

Appendix D : Interface *Function*

```
package goof.core;

/**
 * A function to be evaluated in the framework.
 * @author Marco Biazzini
 * @version 1.2
 */
public interface Function
{

  /**
   * Returns the number of dimensions of the space to be investigated.
   */
  public int d();

  /**
   * Evaluates the function over the vector of coordinates v
   * and returns the obtained value.
   * @param v the vector where the function must be evaluated
   */
  double eval(double[] v);

  /**
   * Returns the minimal values of the space to be considered
   */
  public double[] getRangeMin();

  /**
   * Returns the maximal values of the space to be considered
   */
  public double[] getRangeMax();

}
```