POSL: A Parallel-Oriented Solver Language

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
sous le sceau de l'Université Bretagne Loire

Alejandro Reyes Amaro

École doctorale : Sciences et technologies de l'information, et mathématiques
Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'Informatique de Nantes-Atlantique (LINA)

Directeur de thèse : M. Eric Monfroy, Professeur, Université de Nantes
Co-encadrant : Florian Richoux, Maître de Conferences, Université de Nantes

UNIVERSITÉ DE NANTES

Submitted: dd/mm/2016

Jury:

Prof. Arnaud LALLOUET
Huawei Technologies Ltd., France

Prof. Frédéric LARDEUX
Univeristé d'Angers, France

Prof. Christophe LECOUTRE
Université d'Artois, France

Prof. Salvador ABREU
Universidade de Évora, Portugal

POSL: A Parallel-Oriented Solver Language

Short abstract:
The multi-core technology and massive parallel architectures are nowadays more accessible for a broad public through hardware like the Xeon Phi or GPU cards. This architecture strategy has been commonly adopted by processor manufacturers to stick with Moore's law. However, this new architecture implies new ways of designing and implementing algorithms to exploit their full potential. This is in particular true for constraint-based solvers dealing with combinatorial optimization problems.

Furthermore, the developing time needed to code parallel solvers is often underestimated. In fact, conceiving efficient algorithms to solve certain problems takes a considerable amount of time. In this thesis we present POSL, a Parallel-Oriented Solver Language for building solvers based on meta-heuristic, in order to solve Constraint Satisfaction Problems (CSP) in parallel. The main goal of this thesis is to obtain a system with which solvers can be easily built, reducing therefore their development effort, by proposing a mechanism of code reusing between solvers. It provides a mechanism to implement solver-independent communication strategies. We also present a detailed analysis of the results obtained when solving some CSPs. The goal is not to outperform the state of the art in terms of efficiency, but showing that it is possible to rapidly prototyping with POSL in order to experiment different communication strategies.

Keywords: Constraint satisfaction, meta-heuristics, language.

Contents

Part I

Presentation

1

Introduction

In this chapter the Constraint Satisfaction Problem (CSP) is introduced as the main target problem. Since they are extremely complicated, they are tackled by meta-heuristic methods. However, this methods are not sufficient for some problem instances. The new era of parallelism has opened new ways to solve CSPs. POSL *is proposed as an alternative, a Parallel-Oriented Solver Language to build many different solvers working in parallel, that provides simple mechanisms to construct communication strategies with few effort. Finally, the structure of this document is presented.*

Contents

Combinatorial optimization is the act of obtaining the best result for a problem from a finite set of possibilities. These possibilities are the different combinations of values that the variables can take (configurations), taking into account their domains (integer values), and sometimes a given finite set of restrictions (constraints). These problems have several applications in many fields. In airlines, the crew scheduling is an example of Combinatorial Optimization Problem. It consists of covering flights of the company scheduled in a given time window, with minimum cost, making an efficient and realistic use of the available personal. The problem of task assignment for parallel programs, is one of the most important in parallelism, because it represents the core of a good program performance running in several machines [1]. Combinatorial Optimization Problems are also present in electrical engineering, for example, when an efficient circuit layout design is needed [2].

In some cases, all feasible solutions, i.e., configurations fulfilling all constraints, are equally important, hence, the main goal is only to find one of these solutions. This is the case of *Constraint Satisfaction Problems (CSP)*. In other words, a solution is an assignment of variables satisfying the constraint set. There exist many different techniques to solve such problems, mainly classified into two categories. In the first category are tree-search based algorithms, exploring the full search space. Two of the most important methods in this category are: a) The *constraint propagation* method. It proceeds as follows: when a given variable is assigned a value, the algorithm recomputes the possible value sets and assigned values of all other (not yet assigned) variables. The process continues recursively until there are no more changes in the model. These way, an equivalent but smaller and easier to solve problem is obtained. b) The *backtracking* method. It incrementally constructs candidates to the solutions, by assigning values to variables. Each time an assignment cannot possibly be completed to a valid solution, the previews assignation is discarded.

In practice, Constraint Satisfaction Problems are intractable. Their search space is huge enough to make tree-search based algorithms useless to solve them. In contrast, in the second category are the *meta-heuristics* methods, algorithms that have been shown to be effective in the resolution of these kind of problems. They are an iterative generation process which guides algorithms by combining intelligently different mechanisms for exploring and exploiting the search space, in order to find efficiently near-optimal solutions [3]. In this category, two groups of methods can be found [4]: a) Single-solution based methods (also known as *local search*). They start with an initial solution and move trying to improve it, inside the search space. b) Population-based methods. in these methods, a set of solutions is modified through operators (recombination, mutation, etc.). Both are nature-inspired methods.

On the other hand, the development of computer architecture is leading us toward massively multi/many–core computers. These architectures unlock new algorithmic possibilities to tackle problems sequential algorithms cannot handle, reducing the search times. However, this development must go hand by hand with the development of parallel algorithms. At the time of writing a solution algorithm in parallel, some important decisions have to be taken. The way of organizing the search, either by dividing the search space, or by dividing the problem into smaller and easier to solve sub-problems, can provide an important speedup to the algorithm. Nevertheless, the multi-walk parallel scheme, where solvers work with the whole problem but searching in different zones of the search space, have shown very good results also. Other important decision is whether to use communication between process as a mechanism of cooperation. Theoretically, sharing information between solvers helps to the search process, but in practice, an equilibrium between the contribution of the communication and its inherent overheads is needed.

Many results can be showed in parallel computing. Adaptive Search [5] is an efficient method reaching linear speed-ups on hundreds and even thousands of cores (depending of the problem), using an independent multi-walk local search parallel scheme. Munera et al [6] present another implementation of this algorithm using communication between search engines, showing the efficiency of cooperative multi-walks strategies. All these results use a multi-walk parallel approach and show the robustness and efficiency of this parallel scheme. Although, they all concluded there is room for improvements.

With all these elements, the idea of obtaining a tool to rapidly prototyping solution and communication strategies emerges. In that sense, the **main goal** of this work is to provide a framework to build/use easily and rapidly:

1. *Computation modules*: simple functions easily reusable, which can be used to built meta-heuristic-based algorithm by joining them.
2. *Abstract solvers*: algorithms templates, which can be used to build many different solvers, instantiating different computation modules.
3. Different communication strategies.

1.1 Goals and contributions

The first contribution of this thesis is proposed:

POSL (pronounced "puzzle")
Parallel-Oriented Solver Language

It is a framework based on the creation/utilization of *modules* interconnected through operators, to create local-search meta-heuristic-based solvers. These solvers work in parallel using the multi-walk parallel scheme, and they are connected to each other through communication operators, allowing information sharing. It is well-known software programming is a very time-consuming activity. This is even more the case while developing a parallel software, where debugging is an extremely difficult task. That is why POSL is based on re-usability to propose to CSP solver designers/programmers a parallel framework to quickly build parallel prototypes, speeding-up the design process.

POSL is a language designed to combine modules available in the framework, or to create new ones. There exist two types of modules: computation modules, simple functions receiving an input, then executing an internal algorithm and returning an output, and communication modules, responsible for the information sharing. The created/chosen modules are joined through operators (the POSL's language) to create independent solvers. After that, created solvers can be connected each other using communication operators. This final entity is called a solver set. POSL also provides a framework specification to implement the benchmark (problems to solve), respecting some requirements.

This framework was inspired by a similar idea proposed in [7] without communication, introducing an evolutionary approach that uses a simple composition operator to automatically discover new local search heuristics for SAT and to visualize them as combinations of a set of building blocks. Renaud De Landtsheer et al. present in [8] a framework to facilitate the development of search procedures by using *combinators* to design features commonly found in search procedures as standard bricks and joining them. This approach can speed-up the development and experimentation of search procedures when developing a specific solver based on local search. In [9] is proposed an approach of using cooperating meta-heuristic-based local search processes, using an asynchronous message passing protocol. The cooperation is based on the general strategies of pattern matching and reinforcement learning. POSL uses the combination of both ideas, by combining features of the search process through provided operators, but it also provides an operator-based mechanism to connect solvers, creating communication strategies.

The second contribution of the present work is a detailed study of the solution process of some Constraint Satisfaction Problems chosen as benchmarks, using POSL. In this study some different strategies are proposed, in order to show how the communication can play an important role in the solution of CSPs.

A first study was made using the *Social Golfers Problem*, in which a *standard* communication strategy is used: the communication of the current configuration. This strategy shows to be effective, because it helps to preserve the equilibrium between exploration and exploitation, necessary in the efficient resolution of this problem.

With the *Costas Array Problem*, a similar study was performed, with the slightly difference that the configuration was transmitted in different places of the algorithm.

Another study was performed using the *N-Queens Problem*, in which it was observed that a standard communication strategy is not enough to improve the results without communication. However, with this benchmark a strategy of search-pace partitioning was implemented. This strategy was able to accelerate the beginning of the search, hence the final result in terms of runtime and iterations.

Finally, the *Golomb Ruler Problem* was used to study a different communication strategy, in which the current configuration is communicated, but in contrast to the previews strategies, this configuration is not used to be improved, but to be avoided. The principle is to communicate potential local minima to some solvers, and they will avoid them every time they perform a restart.

In every case, it was possible to show the positive effect of the inter-processes communication. Despite intrinsic overheads, the solver cooperation scheme can help significantly in the search process, if it is studied and chosen correctly. For that reason this study has allowed the validation of the effectiveness of POSL to this purpose.

1.2 Structure of the document

Chapter 2 presents an overview to the state of the art of Combinatorial Optimization Problems. Its definition and the link with Constraint Satisfaction Problems (CSP) are presented, as well as the principal methods to solve them. This chapter is a travel among basic techniques, like *Constraint Propagation*, *meta– and hyper–heuristic methods*; advanced techniques, like *hybridization*, *parallel computing*, and *Solvers cooperation*; and ıparameter setting techniques.

In Chapter **??** prior works leading to POSL are presented. The problem subdivision approach was adopted to divide the domain of a given problem in parallel, in particular, to solve the *K-Medoids Problem*. It contains also a performed study applying the ParamILS tool to find the optimum parameter configuration to *Adaptive Search* solver. ParamILS (version 2.3) is a tool for parameter optimization for parametrized algorithms.

In Chapter **??** is presented formally POSL, the Parallel-Oriented Solver Language that is the heart of this thesis and its main contribution to the community. Its characteristics, main advantages, and a general procedure to be followed in order to use it to solve Constraint Satisfaction Problems is presented.

Results for each study using POSL to build communication strategies to solve the proposed benchmarks are presented in Chapter **??**. In each section, a benchmark problem is defined, the used solver set for each communication strategy are presented, and results are analyzed (details of experiments can be found in Appendices).

Finally, the main results of this thesis are summarized in Chapter **??**, where possible new lines of investigation are also discussed.

2

State of the art

This chapter presents an overview to the state of the art of Combinatorial Optimization Problems and different approaches to tackle them. In Section 2.1 the definition of a Combinatorial Optimization Problem and its links with Constraint Satisfaction Problems (CSP) are introduced, where I concentrate the main efforts, and I give some examples. The basic techniques used to solve these problems are introduced: constraint programing (Section 2.2) and meta–heuristic methods (Section 2.3). I also present some advanced techniques like hyper–heuristic methods in Section 2.4, hybridization in Section 2.5, parallel computing in Section 2.6, and solvers cooperation in Section 2.7. Finally, before ending the chapter with a brief summary, I present parameter setting techniques in Section 2.8.

Contents

2.1 Combinatorial Optimization

Combinatorial Optimization problems come from industrial and real world. We can find them in *Resource Allocations* [10], *Task Scheduling* [11], *Master-keying* [12], *Traveling Salesman* and *Knapsack* problems, among others, which are well-known examples of combinatorial optimization problems [13]. They are a particular case of *optimization* problems, and their main goal is to find an optimum value (minimal or maximal, depending on the problem) for a discrete function f, called *objective function*, involving a set variables $X = \{x_1, \ldots, x_n\}$ defined over a set $D = \{D_1, \ldots, D_n\}$ of discrete domains. These problems generally contain restrictions on the variables called *constraints*, defining the set of forbidden combinations of values for variables in X, tacking into account the problem characteristics.

A *configuration* $s \in D_1 \times D_2 \times \cdots \times D_n$ is a combination of values for the variables in X. The fact of assigning values $v_i \in D_i$ to all variables in $x_i \in X$ is called *evaluation.* When this evaluation is only performed to a given set of variables in X, we called *partial evaluation.* In combinatorial optimization, a *feasible* configuration is a configuration fulfilling all constraints. Finally, a *solution* s^* to the problem is a configuration such that $f(s^*)$ is optimal.

In many practical cases, the main goal is not to find the optimal solution, but finding one feasible configuration. This is the case of Constraint Satisfaction Problems. Formally, we present the definition of a CSP.

Definition 1 (Constraint Satisfaction Problem) *A Constraint Satisfaction Problem (CSP, denoted by $\mathcal{P}$) is a triple $\langle X, D, C\rangle$, where:*

- $X = \{x_1, \ldots, x_n\}$ *is finite a set of variables,*
- $D = \{D_1, \ldots, D_n\}$ *is the set of associated domains. Each domain D_i specifies the set of possible values to the variable x_i.*
- $C = \{c_1, \ldots, c_m\}$ *is a set of constraints. Each constraint is defined involving some variables from X, and specifies the possible combinations of values for these variables.*

In CSPs, a solution is a configuration satisfying all constraints $c_i \in C$.

Definition 2 (Solution of a CSP) *Given a CSP $\mathcal{P} = \langle X, D, C\rangle$ and a configuration $s \in D_1 \times D_2 \times \cdots \times D_n$ we say that it is a solution if and only if:*

$$c_i\,(s) \;\; \textit{is true}\; \forall c_i \in C$$

Let $Var(c_i)$ be the set of involved variables $\{x_1, \ldots, x_p\}$ in the constraint c_i, with $p \leq n$. Then, $c_i\,(s)$ denotes the evaluation using the values from the configuration s to the variables $Var(c_i)$. The set of all solutions of $\mathcal{P}$ is denoted by $Sol(\mathcal{P})$.

A CSP can be considered as a special case of combinatorial optimization problems, where the objective function is to reduce to the minimum the number of violated constraints in the model. A solution is then obtained when the number of violated constraints reach the value zero.

Galinier et al. present in [14] a general approach for solving CSPs. In this work, authors present the concept of *penalty functions* that I pick up in order to write a CSP as an *Unrestricted Optimization Problem* (UOP). This formulation was useful in this thesis for modeling the tackled benchmarks. In this formulation, the objective function of this new problem must be such that its set of optimal solutions is equal to the solution set of the original (associated) CSP.

Definition 3 (Local penalty function) *Let a* **CSP** $\mathcal{P}\langle X, D, C\rangle$ *and a configuration* s *be. We define the operator* **local penalty function** *as follow:*

$$\begin{aligned}
&\omega_i : D\,(X) \times 2^{D(X)} \rightarrow \mathbb{R}^+ \textit{ where:}\\
&\omega_i\,(s, c_i) = \begin{cases} 0 & \textit{if} \qquad c_i(s) \textit{ is true} \\ k \in \mathbb{R}^+ \setminus 0 & \textit{otherwise} \end{cases}
\end{aligned}$$

where $D\,(X) = D_1 \times D_2 \times \cdots \times D_n$

This penalty function defines the cost of a configuration with respect to a given constraint, so if $\omega_i\,(s, c_i) = k$ we say that the configuration s has a local cost k with respect to the constraint c_i. In consequence, we define the *global penalty function*, to define the cost of a configuration with respect to all constraint on a CSP:

Definition 4 (Global penalty function) *Let a* **CSP** $\mathcal{P}\langle X, D, C\rangle$ *and a configuration* s. *We define the operator* **global penalty function** *as follows:*

$$\begin{aligned}
&\Omega : D\,(X) \times 2^{D(X)} \rightarrow \mathbb{R}^+ \textit{ where:}\\
&\Omega\,(s, C) = \sum_{i=1}^{m} \omega_i\,(s, c_i)
\end{aligned}$$

This global penalty function defines the cost of a configuration with respect to a given set of constraints, so if $\Omega\,(s, C) = k$ we say that the configuration s has a cost k with respect to C. We can now formulate a Constraint Satisfaction Problem as an *unrestricted optimization problem*:

Definition 5 (CSP's Associated Unrestricted Optimization Problem) *Given a* **CSP** $\mathcal{P}\langle X, D, C\rangle$ *we define its associated Unrestricted Optimization Problem* $\mathcal{P}_{opt}\langle X, D, C, f\rangle$ *as follows:*

$$\min_{X} f(X, C)$$

Where: $f(X, C) \equiv \Omega(X, C)$ *is the objective function to be minimized over the variable* X

It is important to note that a given s is optimum if and only if $f(s, C) = 0$, which means that s satisfies all the constrains in the original CSP $\mathcal{P}$. This work focuses in solving the Constraint Satisfaction Problem using this formulation.

2.2 Constraint programming

CSPs find a lot of "real-world" applications in the industry. In practice, these problems are tackled through different techniques. One of the most popular is *constraint programming*, a combination of three main ingredients: i) a declarative model of the problem, ii) constraint reasoning techniques like *filtering* and *propagation*, and iii) search techniques. This field is a famous research topic developed by the field of artificial intelligence in the middle of the 70's, and a programming paradigm since the end of the 80's.

Modeling a constrained problem, to be solved using constraint programing techniques, means properly choosing variables and their domains, and a right and efficient representation of the constraints set, aiming to declare as explicitly as possible, the solutions space. On the right election of the problem's model depends not only finding the solution, but also doing it in a fast and efficient way. For modeling CSPs, two tools can be highlighted. MiniZinc is a simple but expressive constraint programming modeling language which is suitable for modeling problems for a range of solvers. It is the most used language for codding CSPs [15]. XCSP is a readable, concise and structured XML-like language for coding CSPs. This format allows us to represent constraints defined either extensionally or in intensionally. Is not more used than MiniZinc but although it was mainly used as the standard in the *International Constraint Solver Competition* (ended in 2009), the *ICSC* dataset is for sure the biggest dataset of CSPs instances existing today.

Constraint reasoning techniques are filtering algorithms applied for each constraint to prune provably infeasible values from the domain of the involved variables. This process is called *constraint propagation*, and they are methods used to modify a Constraint Satisfaction Problem in order to reduce its variables domains, and turning the problem into one that is equivalent, but usually easier to solve [16]. The main goal is to choose one (or some) constraint(s) and enforcing certain consistency levels in the constraint. Achieving global consistency is desirable, because only using brute force algorithms one can arrive to a solution, but this is extremely hard to obtain. For that reason, some other consistency levels, easier to achieve, have been defined, like for example, *arc consistency* and *bound consistency*, which means trying to find values in the variables domain which make constraint unsatisfiable, in order to remove them from the domain. The applied procedure to reduce the variable domains is called *reduction function*, and it is applied until a new, "smaller" and easier to solve is obtained, and it can not be further reduced: a *fixed point*. Local consistency restrictions on the filtering algorithms are necessary to ensure not loosing solution during the propagation process.

We said that a variable $x \in c$, if it is involved into the constraint c. Let the set $Var(c) = \{x_1 \dots x_k\}$ the set of variables involved into a constraint c be, denoted by $Var(c)$. Then, a constraint c is called *arc consistent* if for all $x_i \in Var(c)$ with $1 \leq i \leq k$, and for all $v_j \in D_j$ with $1 \leq j \leq \|D_j\|$:

$$\exists(v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_k) \in D_1 \times \dots \times D_{i-1} \times D_{i+1} \times \dots \times D_k$$

such that $c(v_1, \dots, v_k)$ is fulfilled. In other words, c is arc consistent if for each value of each variable, there exist values for the other variables fulfilling c. In that case, we said that each value in the domain of x_i has a *support* in the domain of the other variables.

We denote by $Bnd(D_i) = \{\min(D_i), \max(D_i)\}$ the bounds of the domain D_i. Then, a constraint is *bound consistent* if for all $x_i \in Var(c)$ with $1 \leq i \leq k$, and for all $v_j \in Bnd(D_j)$ with $1 \leq j \leq 2$:

$$\exists(v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_k) \in Bnd\,(D_1) \times \dots \times Bnd\,(D_{i-1}) \times Bnd\,(D_{i+1}) \times \dots \times Bnd\,(D_k)$$

such that $c(v_1, \dots, v_k)$ is fulfilled. It means that each bound (min/max) in the domain of x_i has a support in the bounds (min/max) of the other variables. As we can notice that arc consistency is a stronger property, but heavier to enforce.

Apt and Monfroy have been proposed in [17] and [18], respectively, a formalization of constraint propagation through *chaotic iterations*, which is a technique that comes from numerical analysis to compute limits of iterations of finite sets of functions, and adapted for computer science needs for naturally explain constraint propagation [19, 20]. Another approach is presented by Monfroy in [21], a coordination-based chaotic iteration algorithm for constraint propagation, which is a scalable, flexible and generic framework for constraint propagation using coordination languages, not requiring special modeling of CSPs. Zoeteweij provides an implementation of this algorithm in DICE (Distributed Constraint Environment) [22] using the MANIFOLD coordination language. Coordination services implement existing protocols for constraint propagation, termination detection and splitting of CSPs. DICE combines these protocols with support for parallel search and the grouping of closely related components into cooperating solvers.

Another implementation of constraint propagation is proposed by Granvilliers et al. in [23], using composition of reductions. It is a general algorithmic approach to tackle strategies that can be dynamically tuned with respect to the current state of constraint propagation, using composition operators. A composition operator models a sub–sequence of an iteration, in which the ordering of application of reduction functions is described by means of combinators for sequential, parallel or fixed–point computation, integrating smoothly the strategies to the model. This general framework provides a good level of abstraction for designing an object-oriented architecture of constraint propagation. Composition can be handled by the *Composite Design Pattern* [24], supporting inheritance between elementary and compound reduction functions. The propagation mechanism uses the *Observer (Listener) Design Pattern* [25], that makes the connection between domain modifications and re–invocation of reduction functions (event-based relations between objects); and the generic algorithm has been implemented using the *Strategy Design Pattern* [26], that allows to parametrize parts of algorithms.

A propagation engine prototype with a *Domain Specific Language* (DSL) was implemented by Prud'homme et al. in [27]. It is a solver–independent language able to configure constraint propagations at the modeling stage. The main contributions are a DSL to ease configure constraint propagation engines, and the exploitation of the basic properties of DSL in order to ensure both completeness and correctness of the produced propagation engine. Some characteristics are required to fully benefit from the DSL. Due to their positive impact on efficiency, modern constraint solvers already implement these techniques: i) Propagators are discriminated thanks to their priority (deciding which propagator to run next): lighter propagators (in the complexity sense) are executed before heavier ones. ii) A controller propagator is attached to each group of propagators. iii) Open access to variable and propagator properties: for instance, variable cardinality, propagator arity or propagator priority. To be more flexible and more accurate, they assume that all arcs from the current *CSP*, are explicitly accessible. This is achieved by explicitly representing all of them and associating them with *watched literals* [28] (controlling the behavior of variable–value pairs to trigger propagation) or *advisors* [29] (a method for supporting incremental propagation in propagator–centered setting).

Most of the times, we can not solve CSPs only applying constraint propagation techniques. It is necessary to combine them with search algorithms. The complete search process consists in testing all possible configurations in an ordered way. Each time a partial evaluation is executed (evaluating just a set of variables), new constraints are posted, meaning that the propagation process can be relaunched. The simplest approach is using a backtracking search.

It can be seen as performing a depth–first traversal of a search tree. This search tree is generated as the search progresses and represents alternative choices that may have to be examined in order to find a solution. Constraints are used to check whether a node may possibly lead to a solution of the CSP and to prune subtrees containing no solutions. A node in the search tree is a *dead-end* if it does not lead to a solution. Differences between searches lies in the selection criteria of the order of variables to be evaluated, and the order of the values to be assigned to the variables. A *static* search strategy is based on selecting the variable with me minimum index, to be evaluated first with the minimum value of its domain. Using this search, the tree *structure* of the search space does not change, but is good for testing propagators. A *dynamic* search strategy is based on selecting the variable with me minimum domain element, to be evaluated first with the minimum value of its domain. In this search strategy, propagators affect the variable selection order. A classical search strategy is based on on selecting the variable with me minimum domain size, to be evaluated first with any values of its domain. Based on the *first fail* principle which tells "*Focus first on the variable that is more likely to cause a fail*". This strategy works pretty well in many cases because by branching early on variables with a few value, the search tree becomes smaller.

In the field of constraint programing we can find a lot of solvers, able to solve constrained problems using these techniques. As examples, we can cite CPLEX, *OR-tools*, GECODE and `Choco`. CPLEX is an analytical decision support toolkit for rapid development and deployment of optimization models using mathematical and constraint programming, to solve very large, real-world optimization problems. GECODE is an efficient open source environment for developing constraint-based system and applications, that provides a modular and extensible constraint solver [30], written in C++ (winner of all gold medals in the *MiniZinc Challenge* from 2008 to 2012). During the formation phase of this PhD, I had the opportunity to perform some pedagogical experiments using two other important and recognized solvers: *OR-tools* and `Choco`. The *OR-tools* is an open source, portable and documented software suite for combinatorial optimization. It contains an efficient constraint programming solver, used internally at Google, where speed and memory consumption are critical.

`Choco` is a free and open-source tool written in java, to describe hard combinatorial problems in the form of CSPs and solving them using Constraint Programing techniques. Mainly developed by people at École des Mines de Nantes (France), is a solver with a nice history, wining some awards, including seven medals in four entries in the *MiniZinc Challenge*. This solver uses multi-thread approach for the resolution, and provide a problem modeler able to manipulate a wide variety of variable types. This problem modeler accepts over 70 constraints, including all classical arithmetical constraints, the possibility of using boolean operations between constraints, table constraints, i.e., defining the sets of tuples that verify the intended relation for a set of variables and a large set of useful classical global constraints including the *alldifferent* constraint, the global *cardinality* constraint, the *cumulative* constraint, among others. Choco also contains a MiniZinc and *XCSP* instance parser. Choco can either deal with satisfaction or optimization problems. The search can be parameterized using a set of predefined variable and value selection heuristics, and also the variable and/or value selectors can be parametrized [31, 32].

Although constraint programing techniques have shown very good results solving constrained problems, the search space in practical instances becomes intractable for them. For that reason, these constrained problems are mostly tackled by *meta-heuristic methods* or hybrid approaches.

2.3 Meta-heuristic methods

Meta-heuristic methods are algorithms generally applied to solve problems deprived of satisfactory problem-specific algorithms to solve them. They are general purpose techniques widely used to solve complex optimization problems in industry and services, in areas ranging from finance to production management and engineering, with relatively few modifications: i) they are nature-inspired (based on some principles from physics or biology), ii) they involve random variables as an stochastic component, therefore approximate and usually non-deterministic, iii) and they have several parameters that need to be fitted [33].

A Meta-heuristic Method is formally defined as an iterative process which guides a subordinate heuristic by combining different concepts for *exploration* (also called *diversification*) i.e. guiding the search process through a much larger portion of the search space, and *exploitation* (also called *intensification*) i.e. guiding the search process into a limited, but promising, region of the search space [3].

In contrast with tree-search based methods, which are subject to combinatorial explosion (required time to find solutions of NP-hard problems increases exponentially w.r.t. the problem size), they do dot perform an ordered and complete search. For that reason they are not able to provide a proof that the optimal solution will be found in a finite (although often prohibitively large) amount of time. Meta-heuristics are therefore developed specifically to find a "*acceptably good*" solution "*acceptably*" fast. In the case of CSPs, finding a feasible solution is enough, for that reason, these methods have been proven to be effective solving these kind of problems.

Sometimes meta-heuristics use domain-specific knowledge in the form of heuristics controlled by an upper level strategy. Nowadays more advanced meta-heuristics use search experience to guide the search [34].

Meta-heuristics are divided into two groups:

1. *Single Solution Based:* more exploitation oriented, intensifying the search in some specific areas. This work focuses its attention on this first group.

2. *Population Based:* more exploration oriented, identifying areas of the search space where there are (or where there could be) the best solutions.

2.3.1 Single Solution Based Meta-heuristic

Methods of the first group are also called *trajectory methods*. They usually start from a candidate configuration s (usually random) inside the search space, and then iteratively make local moves consisting of applying some local modifications to s to create a set of configuration called *neighborhood* $\mathcal{V}(s)$, and selecting a new configuration $s' \in \mathcal{V}(s)$, following some criteria, to be the new candidate solution for the next iteration. This process is repeated until a solution for the problem is found. These methods can be seen as an extension of *local search methods* [4]. Local search methods are the most widely used approaches to solve Combinatorial Optimization Problems because they often produces high–quality solutions in reasonable time [35].

Simulated Annealing (SA) [36] is one of the first algorithms with an explicit strategy to escape from local minima. It is a method inspired by the annealing technique used by metallurgists to obtain a "well ordered" solid state of minimal energy. Its main feature is to allow moves resulting in solutions of worse quality than the current solution under certain probability, in order to scape from local minima, which is decreased during the search process [34]. As an example of an implementation of this algorithm obtaining good results, it can be cited a work presented by Anagnostopoulos et al. in [37] which is an adaptation of a SA algorithm (TTSA) for the Traveling Tournament Problem (TPP) that explores both feasible and infeasible schedules that includes advanced techniques such as strategic oscillation to balance the time spent in the feasible and infeasible regions by varying the penalty for violations; and reheats (increasing the temperature again) to balance the exploration of the feasible and infeasible regions and to escape local minima.

Tabu Search (TS) [38], is a very classic meta-heuristic for Combinatorial Optimization Problems. It explicitly maintains a history of the search, as a short term memory keeping track of the most recently visited solutions, to scape from local minima, to avoid cycles, and to deeply explore the search space. A TS meta-heuristic guides the search on the approach presented in [39] by Iván Dotú and Pascal Van Hentenryck to solve instances of the *Social Golfers* problem, showing that local search is a very effective way to solve this problem. The used approach does not take symmetries into account, leading to an algorithm which is significant simpler than constraint programming solutions.

Guided Local Search (GLS) [40] consists of dynamically changing the objective function to change the search landscape, helping the search escape from local minima. The set of solutions and the neighborhood are fixed, while the objective function is dynamically changed with the aim of making the current local optimum less attractive [34]. Mills et al. propose in [41] an implementation of a GLS, which is used to solve the satisfiability (SAT) problem, a special case of a CSP where variables take booleans values and constraints are disjunctions of literals (i.e. variables or theirs negations).

The *Variable Neighborhood Search* (VNS) is another meta-heuristic that systematically changes the neighborhood size during the search process. This neighborhood can be arbitrarily chosen, but often a sequence $|\mathcal{N}_1| < |\mathcal{N}_2| < \cdots < |\mathcal{N}_{k_{max}}|$ of neighborhoods with increasing cardinality is defined. The choice of neighborhoods of increasing cardinality yields a progressive diversification of the search [42, 34]. Bouhmala et al. introduce in [43] a *generalized Variable Neighborhood Search* for Combinatorial Optimization Problems, where the order in which the neighborhood structures are selected during the search process offers a more effective mechanism for diversification and intensification.

Greedy Randomized Adaptive Search Procedures (GRASP) is an iterative randomized sampling technique in which each iteration provides a solution to the target problem at hand through two phases (constructive and search). The first one constructs an initial solution via an adaptive randomized greedy function. This function construct a solution performing partial evaluations using values of a restricted candidate list (RCL) formed by the best values, incorporating to the current partial solution values resulting in the smallest incremental costs (the greedy aspect of the algorithm). The value to be incorporated into the partial solution is randomly selected from those in the RCL (the random aspect of the algorithm). Then the candidate list is updated and the incremental costs are reevaluated (the adaptive aspect of the algorithm). The second phase applies a local search procedure to the constructed solution in to find an improvement [44]. GRASP does not make any smart use of the history of the search process. It only stores the problem instance and the best found solution. That is why GRASP is often outperformed by other meta-heuristics [34]. However, Resende et al. introduce in [45] some extensions like alternative solution construction mechanisms and techniques to speed up the search are presented.

Many other implementations of local search algorithms have been presented with good results. *Adaptive Search* is an algorithm based local search method, taking advantage of the structure of the problem in terms of constraints and variables. It uses also the concept of *penalty function*, based on this information, seeking to reduce the *error* (a projected cost of a variable, as a measure of how responsible is the variable in the cost of a configuration) on the worse variable so far. It computes the penalty function of each constraint, then combines for each variable the *errors* of all constraints in which it appears. This allows to chose the variable with the maximal *error* will be chosen as a "culprit" and thus its value will be modified for the next iteration with the best value, that is, the value for which the total error in the next configuration is minimal [5, 46, 47]. In [48] Munera et al. based their solution method in Adaptive Search to solve the *Stable Marriage with Incomplete List and Ties* problem [49], a natural variant of the *Stable Marriage Problem* [50], using a cooperative parallel approach. Michel and Van Hentenryck propose in [51] a constraint-based, object-oriented architecture to significantly reduce the development time of local search algorithms. This architecture consists of two main components: a declarative component which models the application in terms of constraints and functions, and a search component which specifies the meta-heuristic, illustrated using COMET, an optimization platform that provides a Java-like programming language to work with constraint and objective functions [52, 53], supporting the local search architecture. It also provides abstraction features to make a clean separation between the model an the search (promoting the reusing of the later) and novel control structures to implement nondeterminism.

2.3.2 Population Based Meta-heuristic

In the second group of meta-heuristic algorithms we can find the methods based on populations. These methods do not work with a single configuration, but with a set of configurations named *population*. They were not part of the main investigation of this thesis, so I will not get into details, but I thinks it is fair to mention some of the most important methods.

The most popular algorithms in this group are population-based methods. They are related to i) *Evolutionary Computation* (EC), inspired by the "Darwin's principle", where only the best adapted individuals will survive, where a population of individuals is modified through recombination and mutation operators, and ii) *Swarm Intelligence* (SI), where the idea is to produce computational intelligence by exploiting behaviors of social interaction [4].

Algorithms based on evolutionary computation have a general structure. Every iteration of the algorithm corresponds to a *generation*, where a population of candidate solutions (called individuals) to a given problem, is capable of reproducing and is subject to genetic variations and environmental pressure that causes natural selection. New solutions are created by applying recombination, by combining two or more selected individuals (parents) to produce one or more new individuals (the offspring). Mutation can be applied allowing the appearance of new traits in the offspring to promote diversity. The fitness (how good the solutions are) of the resulting solutions is evaluated and a suitable selection strategy is then applied to determine which solutions will be maintained into the next generation. As a termination condition, a predefined number of generations (or function evaluations) of simulated evolutionary process is usually used. The evolutionary algorithm's operators are another branch of study, because they have to be selected properly according to the specific problem, due to they will play an important roll in the algorithm behavior [54].

Probably the most popular evolutionary algorithms are *Genetic Algorithms* (GA) [55], where operators are based on the simulation of the genetic variation process to achieve individuals (solutions in this case) more adapted. GAs are usually differently implemented according to the problem: representation of solution (chromosomes), selection strategy, type of crossover (the recombination operator) and mutation operators, etc. The most common representation of the chromosomes is a fixed-length binary string, because simple bit manipulation operations allow the easy implementation of crossover and mutation operations.

Swarm intelligence based methods are inspired by the collective behavior in society of groups different form of live. SI systems are typically made up of a population of simple element, capable of performing certain operations, interacting locally with one another and with their environment. These elements have very limited individual capability, but in cooperation with others can perform many complex tasks necessary for their survival. Ant colony optimization, Particle Swarm Optimization and Bee Colony Optimization are examples to this approach.

Ant Colony optimization algorithms are inspired by the behavior of real ants. Ants searching for food, initially explore the area surrounding the nest by performing a randomized walk. Along the path between food source and nest, ants deposit a pheromone trail on the ground in order to mark some promising path that should guide other ants to the food source. After some time, the shortest path between the nest and the food source has a higher concentration of pheromone, so it attracts more ants [56].

Particle Swarm optimization uses the metaphor of the flocking behavior of birds to solve optimization problems. Each element of the swarm is a candidate solution to the problem, stochastically generated in the search space, and they are connected to some others elements called *neighbors*. It is represented by a velocity, a location in the search space and has a memory which helps it to remember its previous best position. This values describe the *influence* of each element over its neighbors [57].

Bee Colony optimization consists of three groups of bees: employed bees, onlookers and scout bees. A food source is a possible solution to the problem. Employed bees are currently exploiting a food source. They exploit the food source, carry the information about food source back to the hive and share it with onlooker bees. Onlookers bees wait in the hive for the information to be shared with the employed bees to update their knowledge about discovered food sources. Scouts bees are always searching for new food sources near the hive. Employed bees share information about the nectar amount of a food source by dancing in the designated dance area inside the hive. This information represents the quality of the solution. The nature of dance is proportional to the nectar content of food source. Onlooker bees watch the dance and choose a food source according to the probability proportional to the quality of that food source. In that sense, good food sources attract more onlooker bees. Whenever a food source is fully exploited, all employed bees associated with it abandon the food source, and become scouts [58].

2.4 Hyper-heuristic Methods

Hyper-heuristics are automated methodologies for selecting or generating meta-heuristics algorithms to solve hard computational problems [59]. This can be achieved with a learning mechanism that evaluates the quality of the algorithm solutions, in order to become general enough to solve new instances of a given problem. *Hyper-heuristics* are related with the *Algorithm Selection Problem*, so they establish a close relationship between a problem instance, the algorithm to solve it and its performance [60]. Hyper-heuristic frameworks are also known as *Algorithm-Portfolio*–based frameworks. Their goal is predicting the running time of algorithms using statistical regression. Then the fastest predicted algorithm is used to solved the problem until a suitable solution is found or a time-out is reached [61].

This approach have been followed for solving constrained problems. Hyperion^2 [62] is a Java framework for meta– and hyper– heuristics which allows the analysis of the trace taken by an algorithm and its constituent components through the search space. It promotes interoperability via component interfaces, allowing rapid prototyping of meta- and hyper-heuristics, with the potential of using the same source code in either case. It also provides generic templates for a variety of local search and evolutionary computation algorithms, making easier the construction of novel meta- and hyper-heuristics by hybridization (via interface interoperability) or extension (subtype polymorphism). Hyperion^2 is faithful to "*only pay for what you use*", a design philosophy that attempts to ensure that generality doesn't necessarily imply inefficiency. *hMod* is inspired by the previous frameworks, but using a new object-oriented architecture. It encodes the core of the hyper-heuristic in several modules, referred as algorithm containers. *hMod* directs the programmer to define the heuristic using two separate XML files; one for the heuristic selection process and the other one for the acceptance criteria [63].

Evolving evolutionary algorithms are specialized hyper-heuristic method which attempt to readjust an evolutionary algorithm to the problem needs. An evolutionary algorithm (EA) discover the rules and knowledge to find the best algorithm to solve a problem. In [64] Dioşan et al. use linear genetic programming and multi-expression genetic programming to optimize the EA solving unimodal mathematical functions and another EA to adjust the sequence of genetic and reproductive operators. A solution consists of a new evolutionary algorithm capable of outperforming genetic algorithms when solving a specific class of unimodal test functions. An different but interesting point of view is presented in [65], where Samulowitz et al. present *Snappy*, a *Simple Neighborhood-based Algorithm Portfolio* written in *Python*. It is a very resent framework that aims to provide a tool able to improve its own performances through on-line learning. Instead of using the traditional off-line training step, a neighborhood search predicts the performance of the algorithms. It incorporates available knowledge coming from portfolio's runs, by considering the following ways incrementally: 1- Every time a test instance is considered, it is added to the current set of training instances. 2- After selecting an algorithm for a given test instance, the actual runtime information for the selected algorithm on this instance is added to the data set. It means that the difference between neighborhoods of different algorithms represent how often algorithms will be selected. Other interesting idea is proposed by Swan et al. in TEMPLAR, a framework to generate algorithms changing predefined components using hyper-heuristics methods [66].

2.5 Hybridization

The *Hybridization* approach is the one who combine different approaches into the same solution strategy, and recently, it leads to very good results in the constraint satisfaction field. We can find hybridization in combining algorithms in the same branch of investigation. This is the case of *ParadisEO*, a framework to design parallel and distributed hybrid meta-heuristics showing very good results, including a broad range of reusable features to easily design evolutionary algorithms and local search methods [67]. But we can find hybridization also in combining very different techniques, like the work of El-Ghazali Talbi presented in [68], which is a taxonomy of hybrid optimization algorithms is presented in an attempt to provide a mechanism to allow qualitative comparison of hybrid optimization algorithms, combining meta-heuristics with other optimization algorithms from mathematical programming, machine learning and constraint programming.

However, maybe one of the most common in this field, is the combination of meta-heuristic methods and constraint programming techniques. Constraint programming algorithms are based on backtracking mechanisms. These algorithms, also called *complete method* usually explore the search space systematically, and thus guarantee to find a solution if one exists. Meta-heuristic methods may find a solution to a problem, but they can fail even if the problem is satisfiable, because of its local nature. They perform a probabilistic exploration of the search space, so they are not able to guarantee finding a solution. For that reason they are also know as *incomplete methods*. However, they are more efficient (wit respect to response time) than complete methods. The challenge is trying to get the best of both of these methods: exploration of a neighborhood from meta-heuristics, and the power of propagation from constraint programming [69, 70, 71].

Hooker J.N. presents in [72] some ideas to illustrate the common structure present in exact and heuristic methods, to encourage the exchange of algorithmic techniques between them. The goal of this approach is to design solution methods ables to smoothly transform its strategy from exhaustive to non-exhaustive search as the problem becomes more complex. Following this direction, Monfroy et al. present in [73, 74] a general hybridization framework, proposed to combine complete constraints resolution techniques with meta-heuristic optimization methods in order to reduce the problem through domain reduction functions, ensuring not loosing solutions.

A popular way of hybridization is the *portfolio approach*, which is a methodology exploiting the significant variety in performances of different algorithms and combining them in order to create a globally better solver. In [75], Amadini et al. propose `xcsp2mzn`, a tool for converting problem instances from the XCSP format, to MiniZinc. and `mzn2feat`, a tool to extract static and dynamic features from the MiniZinc representation, with the help of the Gecode interpreter, allowing a better and more accurate selection of the solvers to use according to the instances to solve. Based also on the portfolio approach, Amadini et al. propose in [76] a *time splitting* technique to solve optimization problems. Given a problem P and a schedule $Sch = [(\Sigma_1, t_1), \dots, (\Sigma_n, t_n)]$ of n solvers, the corresponding time-split solver is defined as a particular solver such that: 1. runs solver Σ_1 on P for a period of time t_1, 2. then, for $i = 1, \dots, n-1$, runs solver Σ_{i+1} on P for a period of time t_{i+1} exploiting or not the best solution found by the previous solver Σ_i during t_i units of time. *Autonomous search* is a technique based on supervised or controlled learning. This system are another portfolio point of view presented by Hamadi et al. in [77], which improves its performance while it solves problems, either modifying its internal components to take advantage of the opportunities in the search space, or choosing adequately the solver to use.

An interesting hybridization point of view, is the integration of operations research into constraint programming. Fontaine et al. use in [78] a generalization of the optimization paradigm *Lagrangian relaxation*, to relax the hard constraints into the objective function, and applying them into constraint-programming and local search models. It combine the concepts of constraint violation (typically used in constraint programming and local search) and constraint satisfiability (typically used in mathematical programming). Hooker J.N. presents in [79] a detailed description of how operation research models like mixed integer linear programming (MILP) models (which can themselves be relaxed), Lagrangian relaxations, and dynamic programming models can be applied to constraint programming.

2.6 Parallel computing

Despite advances previously presented, hard instances of many problems are still complicated to solve through these techniques. Thanks to *parallel computing*, we have been capable of going one step further in solving CSPs. Parallel computing is a way to solve problems using several computation resources at the same time. It is a powerful alternative to solve problems which would require too much time by using sequential algorithms [80].

Since the late 2000's all processors in modern machines are multi-core. Massively parallel architectures, previously expensive and so far reserved for super–computers, become now a trend available to a broad public through hardware like the Xeon Phi or GPU cards. The power delivered by massively parallel architectures allow us to treat faster constrained problems [81]. However this architectural evolution is a non-sense if algorithms do not evolve at the same time: the development and the implementation of algorithms should take this into account and tackling the problems with very different methods, changing the sequential reasoning of researchers in Computer Science [82, 83].

In the literature on parallel constraint solving [84], two main limiting factors on performance are addressed: 1- inter-process communication overheads (explained in details in Section 2.7), and 2- the *Amdahl*'s law. The Amdahl's law of parallel computing states that the *speed-up* of a parallel algorithm is limited by the fraction of the program that must be executed sequentially. It means that adding more processors may not make the program run faster. It assumes that some percentage of the program or code cannot be parallelized ($T_{sequential}$), and states that the ratio called speed-up of $T_{sequential}$ over $1 - T_{Sequerntial} = T_{parallel}$ is bounded by $1 \setminus T_{sequential}$ when the number of processors $P \to \infty$:

$$Sepeed - Up = \frac{T_{sequential}}{T_{parallel}} \leq \frac{1}{T_{sequential}} \tag{2.1}$$

Another issue, usually underestimated, is the codification. Writing efficient code for parallel machines is less trivial, as it usually involves dealing with low-level APIs such as OpenMP and message-passing interfaces (MPI), among others. However, years of experience have shown that using those frameworks is difficult and error-prone. Usually many undesired behaviors (like deadlocks) make parallel software development very slow compared to sequential approaches. In that sense, Falcou proposes in [85] a programming model: *parallel algorithmic skeletons* (along with a C++ implementation called QUAFF) to make parallel application development easier. This model is a high-order pattern to hide all low-level, architecture or framework dependent code from the user, and provides a decent level of organization. QUAFF is a skeleton-based parallel programming library, which has demonstrated its efficiency and expressiveness solving some application from computer vision. It relies on C++ template meta-programming to reduce the overhead traditionally associated with object-oriented implementations of such libraries allowing some code generation at compilation time. Cahon et al. also propose *ParadisEO*, a framework to design parallel and distributed hybrid meta-heuristics showing very good results, including a broad range of reusable features to easily design evolutionary algorithms and local search methods [67].

The development of techniques and algorithms to solve problems in parallel focuses principally on three fundamentals aspects:

1. *Problem subdivision*,
2. *Search paralelization* and
3. *Inter-process communication.*

They all pursue the same objective: achieving good levels of *scalability.* Scalability is the ability of a system to handle the increasing growth of workload. *Adaptive Search* is a good example of local search method that can scale up to a larger number of cores, e.g., a few hundreds or even thousands of cores [5]. For this algorithm, an implementation of a cooperative multi-walks strategy has been published by Munera et al. in [6]. In this framework, the processes are grouped in teams to achieve search intensification, which cooperate with others teams through a head node (process) to achieve search diversification. Using an adaptation of this method, authors propose a parallel solution strategy able to solve hard instances of *Stable Marriage with Incomplete List and Ties Problem* quickly. This technique has been combined in [86] with an *Extremal Optimization* procedure: a nature-inspired general-purpose meta-heuristic [87].

The issue of subdividing a given problem in some smaller sub-problems is sometimes not easy to address. Even when we can do it, the time needed by each process to solve its own part of the problem is rarely balanced. For that reason it is imperative to apply some complementary techniques to tackle this problem, taking into account that sometimes, the more can be sub-divided a problem, the more balanced will be the execution times of the process [88, 89]. In [90] Arbab and Monfory propose a mechanism to create sub-CSPs (whose union contains all the solutions of the original CSP) by splitting the domain of the variables. The coordination is achieved though communication between processes. The contribution of this work is explained in details in Section 2.7.

In [91] Yasuhara et al. propose a new search method called *Multi-Objective Embarrassingly Parallel Search* (MO–EPS) to solve multi-objective optimization problems, based on: i) Embarrassingly Parallel Search (EPS), where the initial problem is split into a number of independent sub-problems, by partitioning the domain of decision variables [88, 92]; and ii) Multi-Objective optimization adding cuts (MO–AC), an algorithm that transforms the multi-objective optimization problem into a feasibility one, searches a feasible solution and then the search is continued adding constraints to the problem until either the problem becomes infeasible or the search space gets entirely explored [93]. Multi-objective optimization problems involve more than one objective function to be optimized simultaneously. Usually these problems do not have an unique optimal solution because there exist a trade-off between one objective function and the others. For that reason, in a multi-objective optimization problem, the concept of *pareto optimal* points is used. A pareto optimal point is a solution that improving oe ore some objective function values, implies the deterioration of at least one of the other objective function. A collection of pareto optimal points defines a pareto front.

Related to parallelizing the search process, we can find two main approaches. First, the *single walk* approach, in which all the processes try to follow the same path towards the solution, solving their corresponding part of the problem, with or without cooperation (communication). The other is known as *multi walk*, consisting of the execution of various independent processes to find the solution. Each process applies its own strategies (portfolio approach) or simply explores different places inside the search space. Although this approach may seem too trivial and not so smart, it is fair to say that it is in fashion due to the good obtained results using it [5].

Kishimoto et al. present in [94] a comparison between *Transposition-table Driven Scheduling* (TDS) and a parallel implementation of a best-first search strategy (Hash Distributed A*), that uses the standard approach of *work stealing* for partitioning the search space. This technique is based on maintaining a local work queue, (provided by a root process through hash-based distribution that assign an unique processor to each work) accessible to other process that "steal" work from it if they become unoccupied. Authors use MPI, the paradigm of *Message Passing Interface* that allows parallelization, not only in distributed memory based architectures, but also in shared memory based architectures and mixed environments (clusters of multi-core machines) [95].

using MPI, a paradigm of *Message Passing Interface* that allows parallelization, not only in distributed memory based architectures, but also in shared memory based architectures and mixed environments (clusters of multi-core machines) [95]

The same approach is used by Jinnai and Fukunaga in [96] to evaluate *Zobrist Hashing*, an efficient hash function designed for table games like Chess and Go, to mitigate communication overheads.

In [97] Arbelaez et al. present a study of the impact of space-partitioning techniques on the performance of parallel local search algorithms to tackle the *k-medoids* clustering problem. Using a parallel local search method, this work aims to improve the scalability of the sequential algorithm, which is measured in terms of the quality of the solution within a given timeout. Two main techniques are presented for domain partitioning: first, *space-filling curves*, used to reduce any N-dimensional representation into a one-dimension space (this technique is also widely used in the nearest-neighbor-finding problem [98]); and second, *k-Means* algorithm, one of the most popular clustering algorithms [99].

An interesting work is presented by Truchet et al. in [100], which is an estimation of the speed-up (a performance prediction of a parallel algorithm) through statistical analysis of its sequential algorithm is presented. Using this approach it is possible to have a rough idea of the resources needed to solve a given problem in parallel. In this work, authors study the parallel performances of *Las Vegas* algorithms (randomized algorithms whose runtime might vary from one execution to another, even with the same input) under independent multi-walk scheme, and predict the performances of the parallel execution from the runtime distribution of their sequential runs. These predictions are compared to actual speedups obtained for a parallel implementation of the same algorithm and show that the prediction can be quite accurate.

The other important aspect in parallel computing is the inter-process communication, also called *solver cooperation* and it is treated in the next section.

2.7 Solvers cooperation

The interaction between solvers exchanging information is called *solver cooperation.* Its main goal is to improve some kind of limitations or inefficiency imposed by the use of unique solver. In practice, each solver runs in a computation unit, i.e. thread or processor. The cooperation is performed through inter–process communication, by using different methods: *signals*, asynchronous notifications between processes in order to notify an event occurrence; *semaphore*, an abstract data type for controlling access, by multiple processes, to a common resource; *shared memory*, a memory simultaneously accessible by multiple processes; *message passing*, allowing multiple programs to communicate using messages; among others.

Many times a close collaboration between process is required, in order to achieve the solution. But the first inconvenient is the slowness of the communication process. Some work have achieved to identify what information is viable to share. One example is the work presented by Hamadi et al. in [101], where an idea to include low-level reasoning components in the SAT problems resolution is proposed, dynamically adjusting the size of shared clauses to reduce the possible blow up in communication. This approach allows to perform the clause-sharing, controlling the exchange between any pair of processes.

This is a very changeling field, that is way we can find a lot of interesting ideas in the literature to improve parallel solutions through solver cooperation techniques.

Kishimoto et al. present in [102] a parallelization of an algorithm A* (Hash Distributed A*) for *optimal sequential planning* [103], exploiting distributed memory computers clusters, to extract significant speedups from the hardware. In classical planning solving, both the memory and the CPU requirements are main causes of performance bottlenecks, so parallel algorithms have the potential to provide required resources to solve changeling instances.

In [104] Pajot and Monfroy present a paradigm that enables the user to properly separate strategies combining solver applications, from the way the search space is explored in solver cooperations. The cooperation must be supervised by the user, through *cooperation strategy language*, which defines the solver interactions during the search process.

Meta–S is an implementation of a theoretical framework proposed in [105] by Franc et al., which allows to tackle constrained problems, through the cooperation of arbitrary domain–specific constraint solvers. Through its modular structure and its extensible strategy specification language, it also serves as a test–bed for generic and problem–specific (meta-)solving strategies, which are employed to minimize the incurred cooperation overhead. Treating the employed solvers as black boxes, the meta–solver takes constraints from a global pool and propagates them to the individual solvers, which are in return requested to provide newly gained information (i.e., constraints) back to the meta–solver, through variable projections. The major advantage of this approach lies in the ability to integrate arbitrary, new or pre–existing constraint solvers, to form a system that is capable of solving complex mixed–domain constraint problems, at the price of increased cooperation overhead. This overhead can however be reduced through more intelligent and/or problem–specific cooperative solving strategies.

Arbab and Monfory propose in [90] a technique to guide the search by splitting the domain of variables. A *master* process builds the network of variables and domain reduction functions, and sends this information to the *worker* processes. Workers concentrate their efforts on only one sub-CSP and the master collects solutions. The main advantage is that by changing only the search agent, different kinds of search can be performed. The coordination process is managing using the Manifold coordination language [106].

A component-based constraint solver in parallel is proposed in [107] by Zoeteweij and Arbab. In this work, a parallel solver coordinates autonomous instances of a sequential constraint solver, which is used as a software component. The component solvers achieve load balancing of tree search through a time-out mechanism. It is implemented a specific mode of solver cooperation that aims at reducing the turn-around time of constraint solving through parallelization of tree search. The main idea is to try to solve a CSP before a time-out. If it cannot find a solution, the algorithm defines a set of disjoint sub-problems to be distributed among a set of solvers running in parallel. The goal of the time-out mechanism is to provide an implicit load balancing: when a solver is idle, and there are no subproblems available, another solver produces new sub-problems when its time-out elapses.

Munera et al. present in [6] a new paradigm that includes cooperation between processes, in order to improve the independent multi-walk approach. In that case, cooperative search methods add a communication mechanism to the independent walk strategy, to share or exchange information between solver instances during the search process. This proposed framework is oriented towards distributed architectures based on clusters of nodes, with the notion of *teams* running on nodes and controlling several search engines (*explorers*) running on cores. All teams are distributed and thus have limited inter–node communication. This tool provides diversification through communication between teams, extending the search to different regions of the search space. Intensification is ensured through communication between explorers, and it is achieved swarming to the most promising neighborhood found by explorers. This framework was developed using the *X10 programming language*, which is a novel language for parallel processing developed by IBM Research, giving more flexibility than traditional approaches, e.g. MPI communication package.

A similar approach is presented by Guo et al. in [108], exploring principles of diversification and intensification in portfolio–based parallel SAT solving. To study their trade–off, they define two roles for the computational units. Some of them classified as *masters* perform an original search strategy, ensuring diversification. The remaining units, classified as *slaves* are there to intensify their master's strategy. Results lead to an original intensification strategy which outperforms the best parallel SAT solver *ManySAT*, and solves some open SAT instances.

Hamadi et al. propose in [109] the first *Deterministic Parallel DPLL* (a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulas in conjunctive normal form) engine. The experimental results show that their approach preserves the performance of the parallel portfolio approach while ensuring full reproducibility of the results. Parallel exploration of the search space, defines a controlled environment based on a total ordering of solvers interactions through synchronization barriers. The frequency of exchanges (conflict-clauses) influences considerably the performance of the solver. The paper explores the trade off between frequent synchronizing which allows the fast integration of foreign conflict–clauses at the cost of more synchronizing steps, and infrequent synchronizing at the cost of delayed foreign conflict-clauses integration.

Considering the problem of parallelizing restarted backtrack search (the problem of finding the right time to to restart the search after some fails), Cire et al. have developed in [110] a simple technique for parallelizing restarted search deterministically. They demonstrate experimentally that they can achieve near–linear speed–ups in practice, when the number of processors is constant and the number of restarts grows to infinity. The proposed technique is the following: each parallel search process has its own local copy of a scheduling class which assigns restarts and their respective fail–limits to processors. This scheduling class computes the next *Luby* restart fail–limit and adds it to the processor that has the lowest number of accumulated fails so far, following an *earliest–start–time–first strategy.* Like this, the schedule is filled and each process can infer which is the next fail–limit that it needs to run based on the processor it is running on – without communication. Overhead is negligible in practice since the scheduling itself runs extremely fast compared to CP search, and communication is limited by informing other processes when a solution has been found.

2.8 Parameter setting techniques

Most of previously exposed methods to tackle combinatorial problems, involve a number of parameters that govern their behavior, and they need to be well adjusted. Most of the times they depend on the nature of the specific problem, so they require a previous analysis to study their behavior [111]. That is way another branch of the investigation arises: *parameter tuning.* It is also known as a meta optimization problem, because the main goal is to find the best solution (parameter configuration) for a program, which will try to find the best solution for some problem as well. In order to measure the quality of some found parameter setting for a program (solver), one of these criteria are taken into consideration: the speed of the run or the quality of the found solution for the problem that it solves. The selection of proper parameters for a particular algorithm is a quite complicate subject. This is the reason why many researchers are motivated to develop techniques to find good parameter settings automatically.

There exist tow classes to classify these methods:

1. *Off-line tunning*: Also known just as parameter tuning, were parameters are computed before the run.
2. *On-line tunning*: Also known as parameter control, were parameters are adjusted during the run, and

2.8.1 Off-line tunning

The technique of parameter tuning or off-line tunning, is used to compute the best parameter configuration for an algorithm before the run (solving a given instance of a problem), in order to obtain the best performance. The found parameters configuration does not change once the run is started.

There exist some techniques to tune algorithms. The most common is the so called *racing procedure* which is based on a simple idea: sequentially evaluating candidates (parameter setups) on a series of benchmark instances and eliminate parameter configurations as soon as they are found too far behind the candidate with the overall best performance at a given stage of the race (*incumbent*). A popular variant of this technique is the *F-Race* method. It is also based on eliminating parameter configurations in some given steps. However, rather than just performing pairwise comparisons with the incumbent, it first uses the *rank-based Friedman test*: for ni independent s random variables, it assesses whether the N_c configurations in the race show no significant performance differences on the N_i given instances; if there exist some configurations show better results than others, a series of pairwise tests between the incumbent and all other configurations is performed. All configurations found to have substantially worse results than the incumbent are removed from the race. The procedure is terminated either when only one configuration remains, or after a given timeout [112]. Eiben et al. present in [113] a study of this methods and their applications tuning Evolutionary Algorithms (EA).

REVAC is a method presented in [114] by Nannen et al., based on information theory to measure parameter relevance, to calibrate parameters of EAs in a robust way. Instead of estimating the performance of an EA for different parameter values, the method estimates the expected performance when parameter values are chosen from following a probability density distribution C. The method iteratively refines the probability distribution C over possible parameter sets, and starting with a uniform distribution C_0 over the initial parameter space $\mathcal{X}$, the method gives a higher probability to regions of $\mathcal{X}$ that increase the expected performance of the target EA. In [115] Smit et al. present a case study demonstrating that using the REVAC the "world champion" EA (the winner of the CEC-2005 competition) can be improved with few effort.

In [116], Riff et al. present *EVOCA*, a tool which allows meta-heuristics designers to obtain good parameter configuration with few effort. *EVOCA* use an EA to find a good parameter setting for a meta-heuristic, and it is used as a step of the iterative design process. This tool allows designer to find parameter settings both for quantitative parameters (numerical values) and for qualitative parameters (e.g. crossover operators). Another tool for this end, but this time using local search methods to find the parameter setting is proposed in [117] by Hutter et al.. It has been applied with success in many combinatorial problems in order to find the best parameter configuration. PARAMILS It is an open source program written in *Ruby*, and the public source include some examples and a detailed and complete user guide with a compact explanation about how to use it with a specific solver [118].

A different but interesting technique was successfully used to tune parameters for EAs through a model based on a *case-based reasoning* system. The work presented in [119] by Yeguas et al. attempts to imitate the human behavior in solving problems: look in the memory how we have solved a similar problem.

2.8.2 On-line tunning

Although parameter tunning shows to be an effective way to adjust parameters to sensibles algorithms, in some problems the optimal parameter settings may be different for various phases of the search process. This is the main motivation to use on-line tuning techniques to find the best parameter setting, also called *parameter control techniques*. Parameter control techniques are divided into i) *deterministic parameter control*, where the value of a parameter is altered by some deterministic rule, ignoring any feedback; ii) *adaptive parameter control*, which continually update their parameters using feedback from the population or the search, and this feedback is used to determine the direction or magnitude of the parameter changes; and iii) *self-adaptive parameter control*, which assigns different parameters to each individual. Here, parameters to be adapted are coded into the chromosomes that undergo mutation and recombination [120].

Drozdik et al. present in [121] a study of various approaches to find out if one can find an inherently better one in terms of performance and whether the parameter control mechanisms can find favorable parameters in problems which can be successfully optimized only with a limited set of parameters. They focused in the most important parameters: i) the *scaling factor*, which controls the structure of new invidious; and ii) the *crossover probability*.

Differential Evolution (DE) algorithm has been demonstrated to be an efficient and robust optimization method. However, its performance is very sensitive to the parameters setting, and this sensibility changes from a problem to another. Liu et al. propose in [122] an adaptive approach which uses fuzzy logic controllers to guide the search parameters, with the novelty of changing the mutation control parameter and the crossover operator during the optimization process. *SaDE* is a self-adaptive DE algorithm proposed by Qin et al. in [123], where both trial vector generation strategies and their associated control parameter values are gradually adjusted by learning from the way they have generated their previous promising solutions, eliminating this way the time-consuming exhaustive search for the most suitable parameter setting. This algorithm has been generalized by Huang et al. to multi-objective realm, with objective-wise learning strategies (*OW-MOSaDE*) [124].

META-GAS is a genetic self-adapting algorithm proposed by Clune et al., adjusting operators of genetic algorithms. In this paper the authors propose an approach of moving towards a

Genetic Algorithm that does not require a fixed and predefined parameter setting, because it evolves during the run [125].

2.9 Summary

In this chapter I have presented an overview of the different techniques to solve Constraint Satisfaction Problems. The classics are the so called tree-search methods, which perform an exhaustive search of the solution inside the search space. The main disadvantage of this methods is that in practice, real-world instances of the problems are very complicated to solve, due to the search space's huge size, making them in most of the cases intractable. Some techniques can be applied to reduce the search space. This is the case of constraint propagation, which are methods used to modify a CSP in order to reduce its variables domains, and turning the problem into another that is equivalent, but usually easier to solve. However, sometimes it is not enough, and we must resort to more powerful methods, like meta-heuristics.

Meta-heuristic methods have shown good results solving large and complex CSPs. They are algorithms applying different techniques to guide the search as direct as possible through the solution. Meta-heuristic methods are divided into two categories. In the first category we can find population-based methods (e.g. evolutionary algorithms). These methods work with a set of configurations (population). Every iteration the population is capable of reproducing and is subject to variations. New configurations are created by applying some modification's operators. The fitness of the resulting population is evaluated and a selection strategy is applied to determine configurations to the next generation. In the second category we can find single-solution-based methods (e.g. tabu search). Also called trajectory methods, they start from a candidate configuration inside the search space. To this configuration some local modifications are applied to create a set of configuration called neighborhood. Then a selection criteria is applied to choose a new configuration for the next iteration. In this thesis, special attention was given to techniques in this category.

With the goal of improving even more the obtained results, some other techniques have been implemented. This is the case of hyper-heuristics, which are automated techniques for selecting or generating meta-heuristics algorithms to solve hard computational problems. They are also known as portfolio-based algorithm, and their goal is predicting the performance of algorithms using different techniques, in order to select the predicted algorithm to solved the problem. An other popular approach is the hybridization, which combines different approaches into the same solution strategy. We can find hybridization in combining algorithms in the same branch of investigation, but not only there. Interesting results have been published when combining constraint programming, meta-heuristics and operations research techniques.

The era of multi/many-core computers, and the development of parallel algorithms have opened new ways to solve constrained problems. A lot of results have been presented suggesting that this approach can substantially improve the efficiency in solving these problems, with or without cooperation. For that reason the present work attempts to propose new algorithms to solve Constraint Satisfaction Problems in parallel in Chapter **??**, and to show the importance and the success of this approach by providing a deep study of some parallel communication strategies in Chapter **??**.

3

Bibliography

[1] Vangelis Th Paschos, editor. *Applications of combinatorial optimization*. John Wiley & Sons, 2013.

[2] Francisco Barahona, Martin Groetschel, Michael Juenger, and Gerhard Reinelt. An Application of Combinatorial Optimization To Statistical Physics and Circuit Layout Design. *Operations Research*, 36(3):493 – 513, 1988.

[3] Ibrahim H Osman and Gilbert Laporte. Metaheuristics : A bibliography. *Annals of Operations research*, 63(5):511–623, 1996.

[4] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82–117, jul 2013.

[5] Daniel Diaz, Florian Richoux, Philippe Codognet, Yves Caniou, and Salvador Abreu. Constraint-Based Local Search for the Costas Array Problem. In *Learning and Intelligent Optimization*, pages 378–383. Springer, 2012.

[6] Danny Munera, Daniel Diaz, Salvador Abreu, and Philippe Codognet. A Parametric Framework for Cooperative Parallel Local Search. In *Evolutionary Computation in Combinatorial Optimisation*, volume 8600 of *LNCS*, pages 13–24. Springer, 2014.

[7] Alex S Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary computation*, 16(1):31–61, 2008.

[8] Renaud De Landtsheer, Yoann Guyot, Gustavo Ospina, and Christophe Ponsard. Combining Neighborhoods into Local Search Strategies. In *11th MetaHeuristics International Conference*, Agadir, 2015. Springer.

[9] Simon Martin, Djamila Ouelhadj, Patrick Beullens, Ender Ozcan, Angel A Juan, and Edmund K Burke. A Multi-Agent Based Cooperative Approach To Scheduling and Routing. *European Journal of Operational Research*, 2016.

[10] Mahuna Akplogan, Jérome Dury, Simon de Givry, Gauthier Quesnel, Alexandre Joannon, Arnauld Reynaud, Jacques Eric Bergez, and Frédérick Garcia. A Weighted CSP approach for solving spatio-temporal planning problem in farming systems. In *11th Workshop on Preferences and Soft Constraints Soft 2011.*, Perugia, Italy, 2011.

[11] Louise K. Sibbesen. *Mathematical models and heuristic solutions for container positioning problems in port terminals*. Doctor of philosophy, Technical University of Danemark, 2008.

[12] Wolfgang Espelage and Egon Wanke. The combinatorial complexity of masterkeying. *Mathematical Methods of Operations Research*, 52(2):325–348, 2000.

[13] Barbara M Smith. Modelling for Constraint Programming. *Lecture Notes for the First International Summer School on Constraint Programming*, 2005.

[14] Philippe Galinier and Jin-Kao Hao. A General Approach for Constraint Solving by Local Search. *Journal of Mathematical Modelling and Algorithms*, 3(1):73–88, 2004.

[15] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. MiniZinc: Towards A Standard CP Modelling Language. In *Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[16] Christian Bessiere. Constraint Propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 3, pages 29–84. Elsevier, 1st edition, 2006.

[17] Krzysztof R. Apt. From Chaotic Iteration to Constraint Propagation. In *24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, pages 36–55, 1997.

[18] Éric Monfroy and Jean-Hugues Réty. Chaotic Iteration for Distributed Constraint Propagation. In *ACM symposium on Applied computing SAC '99*, pages 19–24, 1999.

[19] Daniel Chazan and Willard Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199–222, 1969.

[20] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence amd Programming Languages*, volume 12, pages 1–12, Rochester, NY, 1977.

[21] Éric Monfroy. A coordination-based chaotic iteration algorithm for constraint propagation. In *Proceedings of The 15th ACM Symposium on Applied Computing, SAC 2000*, pages 262–269. ACM Press, 2000.

[22] Peter Zoeteweij. Coordination-based distributed constraint solving in DICE. In *Proceedings of the 18th ACM Symposium on Applied Computing (SAC 2003)*, pages 360–366, New York, 2003. ACM Press.

[23] Laurent Granvilliers and Éric Monfroy. Implementing Constraint Propagation by Composition of Reductions. In *Logic Programming*, pages 300–314. Springer Berlin Heidelberg, 2001.

[24] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. The Iterator and Composite Patterns. Well-Managed Collections. In *Head First Design Patterns*, chapter 9, pages 315–384. O'Relliy, 1st edition, 2004.

[25] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. The Observer Pattern. Keeping your Objects in the know. In *Head First Design Patterns*, chapter 2, pages 37–78. O'Relliy, 1st edition, 2004.

[26] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. Introduction to Design Patterns. In *Head First Design Patterns*, chapter 1, pages 1–36. O'Relliy, 1st edition, 2004.

[27] Charles Prud'homme, Xavier Lorca, Rémi Douence, and Narendra Jussien. Propagation engine prototyping with a domain specific language. *Constraints*, 19(1):57–76, sep 2013.

[28] Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched Literals for Constraint Propagation in Minion. *Lecture Notes in Computer Science*, 4204:182–197, 2006.

[29] Mikael Z. Lagerkvist and Christian Schulte. Advisors for Incremental Propagation. *Lecture Notes in Computer Science*, 4741:409–422, 2007.

[30] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. *Modeling and Programming with Gecode*. 2013.

[31] Narendra Jussien, Hadrien Prud'homme, Charles Cambazard, Guillaume Rochart, and François Laburthe. Choco: an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08),*, Paris, France, 2008.

[32] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. Choco Documentation. Technical report, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[33] Johann Dréo, Patrick Siarry, Alain Pétrowski, and Eric Taillard. Introduction. In *Metaheuristics for Hard Optimization*. Springer, 2006.

[34] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

[35] Stefan Voss, Silvano Martello, Ibrahim H. Osman, and Catherine Roucairol, editors. *Meta-heuristics: Advances and trends in local search paradigms for optimization*. Springer Science+Business Media, LLC, 2012.

[36] Alexander G. Nikolaev and Sheldon H. Jacobson. Simulated Annealing. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 1, pages 1–39. Springer, 2nd edition, 2010.

[37] Aris Anagnostopoulos, Laurent Michel, Pascal Van Hentenryck, and Yannis Vergados. A simulated annealing approach to the travelling tournament problem. *Journal of Scheduling*, 2(9):177—-193, 2006.

[38] Michel Gendreau and Jean-Yves Potvin. Tabu Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 2, pages 41–59. Springer, 2nd edition, 2010.

[39] Iván Dotú and Pascal Van Hentenryck. Scheduling Social Tournaments Locally. *AI Commun*, 20(3):151—-162, 2007.

[40] Christos Voudouris, Edward P.K. Tsang, and Abdullah Alsheddy. Guided Local Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 11, pages 321–361. Springer, 2 edition, 2010.

[41] Patrick Mills and Edward Tsang. Guided local search for solving SAT and weighted MAX-SAT problems. *Journal of Automated Reasoning*, 24(1):205–223, 2000.

[42] Pierre Hansen, Nenad Mladenovie, Jack Brimberg, and Jose A. Moreno Perez. Variable neighborhood Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 3, pages 61–86. Springer, 2010.

[43] Noureddine Bouhmala, Karina Hjelmervik, and Kjell Ivar Overgaard. A generalized variable neighborhood search for combinatorial optimization problems. In *The 3rd International Conference on Variable Neighborhood Search (VNS'14)*, volume 47, pages 45–52. Elsevier, 2015.

[44] Thomas A. Feo and Mauricio G.C. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, (6):109–134, 1995.

[45] Mauricio G.C Resende. Greedy randomized adaptive search procedures. In *Encyclopedia of optimization*, pages 1460–1469. Springer, 2009.

[46] Philippe Codognet and Daniel Diaz. Yet Another Local Search Method for Constraint Solving. In *Stochastic Algorithms: Foundations and Applications*, pages 73–90. Springer Verlag, 2001.

[47] Yves Caniou, Philippe Codognet, Florian Richoux, Daniel Diaz, and Salvador Abreu. Large-Scale Parallelism for Constraint-Based Local Search: The Costas Array Case Study. *Constraints*, 20(1):30–56, 2014.

[48] Danny Munera, Daniel Diaz, Salvador Abreu, Francesca Rossi, and Philippe Codognet. Solving Hard Stable Matching Problems via Local Search and Cooperative Parallelization. In *29th AAAI Conference on Artificial Intelligence*, Austin, TX, 2015.

[49] Kazuo Iwama, David Manlove, Shuichi Miyazaki, and Yasufumi Morita. Stable marriage with incomplete lists and ties. In *ICALP*, volume 99, pages 443–452. Springer, 1999.

[50] David Gale and Lloyd S. Shapley. College Admissions and the Stability of Marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[51] Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. *ACM SIGPLAN Notices*, 37(11):83–100, 2002.

[52] Dynamic Decision Technologies Inc. *Dynadec. Comet Tutorial.* 2010.

[53] Laurent Michel and Pascal Van Hentenryck. The comet programming language and system. In *Principles and Practice of Constraint Programming*, pages 881–881. Springer Berlin Heidelberg, 2005.

[54] Jorge Maturana, Álvaro Fialho, Frédéric Saubion, Marc Schoenauer, Frédéric Lardeux, and Michèle Sebag. Adaptive Operator Selection and Management in Evolutionary Algorithms. In *Autonomous Search*, pages 161–189. Springer Berlin Heidelberg, 2012.

[55] Colin R. Reeves. Genetic Algorithms. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 5, pages 109–139. Springer, 2010.

[56] Marco Dorigo and Thomas Stützle. Ant colony optimization: overview and recent advances. In *Handbook of Metaheuristics*, volume 146, chapter 8, pages 227–263. Springer, 2nd edition, 2010.

[57] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

[58] Weifeng Gao, Sanyang Liu, and Lingling Huang. A global best artificial bee colony algorithm for global optimization. *Journal of Computational and Applied Mathematics*, 236(11):2741–2753, 2012.

[59] Konstantin Chakhlevitch and Peter Cowling. Hyperheuristics : Recent Developments. In *Adaptive and multilevel metaheuristics*, pages 3–29. Springer, 2008.

[60] Patricia Ryser-Welch and Julian F. Miller. A Review of Hyper-Heuristic Frameworks. In *Proceedings of the Evo20 Workshop, AISB*, 2014.

[61] Kevin Leyton-Brown, Eugene Nudelman, and Galen Andrew. A portfolio approach to algorithm selection. In *IJCAI*, pages 1542–1543, 2003.

[62] Alexander E.I. Brownlee, Jerry Swan, Ender Özcan, and Andrew J. Parkes. Hyperion 2. A toolkit for {meta- , hyper-} heuristic research. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1133–1140, Vancouver, BC, 2014. ACM.

[63] Enrique Urra, Daniel Cabrera-Paniagua, and Claudio Cubillos. Towards an Object-Oriented Pattern Proposal for Heuristic Structures of Diverse Abstraction Levels. *XXI Jornadas Chilenas de Computación 2013*, 2013.

[64] Laura Dioşan and Mihai Oltean. Evolutionary design of Evolutionary Algorithms. *Genetic Programming and Evolvable Machines*, 10(3):263–306, 2009.

[65] Horst Samulowitz, Chandra Reddy, Ashish Sabharwal, and Meinolf Sellmann. Snappy: A simple algorithm portfolio. In *Theory and Applications of Satisfiability Testing - SAT 2013*, volume 7962 LNCS, pages 422–428. Springer, 2013.

[66] Jerry Swan and Nathan Burles. Templar - a framework for template-method hyper-heuristics. In *Genetic Programming*, volume 9025 of *LNCS*, pages 205–216. Springer International Publishing, 2015.

[67] Sébastien Cahon, Nordine Melab, and El-Ghazali Talbi. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[68] El-Ghazali Talbi. Combining metaheuristics with mathematical programming, constraint programming and machine learning. *4or*, 11(2):101–150, 2013.

[69] Narendra Jussien and Olivier Lhomme. Local Search with Constant Propagation and Conflict-Based Heuristics. *Artificial Intelligence*, 139(1):21–45, 2002.

[70] Gilles Pesant and Michel Gendreau. A View of Local Search in Constraint Programming. In *Second International Conference on Principles an Practice of Constraint Programming*, number 1118, pages 353–366. Springer, 1996.

[71] Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. *Computer*, 1520(Springer):417–431, 1998.

[72] John N. Hooker. Toward Unification of Exact and Heuristic Optimization Methods. *International Transactions in Operational Research*, 22(1):19–48, 2015.

[73] Éric Monfroy, Frédéric Saubion, and Tony Lambert. Hybrid CSP Solving. In *Frontiers of Combining Systems*, pages 138–167. Springer Berlin Heidelberg, 2005.

[74] Éric Monfroy, Frédéric Saubion, and Tony Lambert. On Hybridization of Local Search and Constraint Propagation. In *Logic Programming*, pages 299–313. Springer Berlin Heidelberg, 2004.

[75] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Features for Building CSP Portfolio Solvers. *arXiv:1308.0227*, 2013.

[76] Roberto Amadini and Peter J Stuckey. Sequential Time Splitting and Bounds Communication for a Portfolio of Optimization Solvers. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, volume 1, pages 108–124. Springer, 2014.

[77] Youssef Hamadi, Éric Monfroy, and Frédéric Saubion. An Introduction to Autonomous Search. In *Autonomous Search*, pages 1–11. Springer Berlin Heidelberg, 2012.

[78] Daniel Fontaine, Laurent Michel, and Pascal Van Hentenryck. Constraint-Based Lagrangian Relaxation. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 324–339. Springer, 2014.

[79] John N. Hooker. Operations Research Methods in Constraint Programming. In *Handbook of Constraint Programming*, chapter 15. 2006.

[80] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Introduction to Parallel Computing. In *Introduction to Parallel Computing*, chapter 1, pages 1–9. Addison Wesley, 2nd edition, 2003.

[81] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, 2007. ACM.

[82] Mark D. Hill and Michael R. Marty. Amdahl's Law in the multicore era. *IEEE Computer*, (7):33–38, 2008.

[83] Peter Sanders. Engineering Parallel Algorithms: The Multicore Transformation. *Ubiquity*, 2014(July):1–11, 2014.

[84] Ian P Gent, Chris Jefferson, Ian Miguel, Neil C A Moore, Peter Nightingale, Patrick Prosser, and Chris Unsworth. A Preliminary Review of Literature on Parallel Constraint Solving. In *Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving*, 2011.

[85] Joel Falcou. Parallel programming with skeletons. *Computing in Science and Engineering*, 11(3):58–63, 2009.

[86] Danny Munera, Daniel Diaz, and Salvador Abreu. Solving the Quadratic Assignment Problem with Cooperative Parallel Extremal Optimization. In *Evolutionary Computation in Combinatorial Optimization*, pages 251–266. Springer, 2016.

[87] Stefan Boettcher and Allon Percus. Nature's way of optimizing. *Artificial Intelligence*, 119(1):275–286, 2000.

[88] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly Parallel Search. In *Principles and Practice of Constraint Programming*, pages 596–610. Springer, 2013.

[89] Mark D. Hill. What is Scalability? *ACM SIGARCH Computer Architecture News*, 18:18–21, 1990.

[90] Farhad Arbab and Éric Monfroy. Distributed Splitting of Constraint Satisfaction Problems. In *Coordination Languages and Models*, pages 115–132. Springer, 2000.

[91] M Yasuhara, T Miyamoto, K Mori, S Kitamura, and Y Izui. Multi-Objective Embarrassingly Parallel Search. In *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 853–857, Singapore, 2015. IEEE.

[92] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Improvement of the Embarrassingly Parallel Search for Data Centers. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 622–635, Lyon, 2014. Springer.

[93] Prakash R. Kotecha, Mani Bhushan, and Ravindra D. Gudi. Efficient optimization strategies with constraint programming. *AIChE Journal*, 56(2):387–404, 2010.

[94] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, 195:222–248, 2013.

[95] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Programming Using the Message-Passing Paradigm. In *Introduction to Parallel Computing*, chapter 6, pages 233–278. Addison Wesley, second edition, 2003.

[96] Yuu Jinnai and Alex Fukunaga. Abstract Zobrist Hashing : An Efficient Work Distribution Method for Parallel Best-First Search. *30th AAAI Conference on Artificial Intelligence (AAAI-16).*

[97] Alejandro Arbelaez and Luis Quesada. Parallelising the k-Medoids Clustering Problem Using Space-Partitioning. In *Sixth Annual Symposium on Combinatorial Search*, pages 20–28, 2013.

[98] Hue-Ling Chen and Ye-In Chang. Neighbor-finding based on space-filling curves. *Information Systems*, 30(3):205–226, may 2005.

[99] Pavel Berkhin. Survey Of Clustering Data Mining Techniques. Technical report, Accrue Software, Inc., 2002.

[100] Charlotte Truchet, Alejandro Arbelaez, Florian Richoux, and Philippe Codognet. Estimating Parallel Runtimes for Randomized Algorithms in Constraint Solving. *Journal of Heuristics*, pages 1–36, 2015.

[101] Youssef Hamadi, Said Jaddour, and Lakhdar Sais. Control-Based Clause Sharing in Parallel SAT Solving. In *Autonomous Search*, pages 245–267. Springer Berlin Heidelberg, 2012.

[102] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Scalable, Parallel Best-First Search for Optimal Sequential Planning. *In ICAPS-09*, pages 201–208, 2009.

[103] Claudia Schmegner and Michael I. Baron. Principles of optimal sequential planning. *Sequential Analysis*, 23(1):11–32, 2004.

[104] Brice Pajot and Éric Monfroy. Separating Search and Strategy in Solver Cooperations. In *Perspectives of System Informatics*, pages 401–414. Springer Berlin Heidelberg, 2003.

[105] Stephan Frank, Petra Hofstedt, and Pierre R. Mai. Meta-S: A Strategy-Oriented Meta-Solver Framework. In *Florida AI Research Society (FLAIRS) Conference*, pages 177–181, 2003.

[106] Farhad Arbab. Coordination of Massively Concurrent Activities. Technical report, Amsterdam, 1995.

[107] Peter Zoeteweij and Farhad Arbab. A Component-Based Parallel Constraint Solver. In *Coordination Models and Languages*, pages 307–322. Springer, 2004.

[108] Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Diversification and Intensification in Parallel SAT Solving. *Principles and Practice of Constraint Programming*, pages 252–265, 2010.

[109] Youssef Hamadi, Cedric Piette, Said Jabbour, and Lakhdar Sais. Deterministic Parallel DPLL system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:127–132, 2011.

[110] Andre A. Cire, Sendar Kadioglu, and Meinolf Sellmann. Parallel Restarted Search. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 842–848, 2011.

[111] Mauro Birattari, Mark Zlochin, and Marrco Dorigo. Towards a Theory of Practice in Metaheuristics Design. A machine learning perspective. *RAIRO-Theoretical Informatics and Applications*, 40(2):353–369, 2006.

[112] Holger H. Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous Search*, pages 37–71. Springer Berlin Heidelberg, 2012.

[113] Agoston E Eiben and Selmar K Smit. Evolutionary algorithm parameters and methods to tune them. In *Autonomous Search*, pages 15–36. Springer Berlin Heidelberg, 2011.

[114] Volker Nannen and Agoston E. Eiben. Relevance Estimation and Value Calibration of Evolutionary Algorithm Parameters. *IJCAI*, 7, 2007.

[115] S. K. Smit and A. E. Eiben. Beating the 'world champion' evolutionary algorithm via REVAC tuning. *IEEE Congress on Evolutionary Computation*, pages 1–8, jul 2010.

[116] Maria-Cristina Riff and Elizabeth Montero. A new algorithm for reducing metaheuristic design effort. *IEEE Congress on Evolutionary Computation*, pages 3283–3290, jun 2013.

[117] Frank Hutter, Holger H Hoos, and Kevin Leyton-brown. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[118] Frank Hutter. Updated Quick start guide for ParamILS, version 2.3. Technical report, Department of Computer Science University of British Columbia, Vancouver, Canada, 2008.

[119] E. Yeguas, M.V. Luzón, R. Pavón, R. Laza, G. Arroyo, and F. Díaz. Automatic parameter tuning for Evolutionary Algorithms using a Bayesian Case-Based Reasoning system. *Applied Soft Computing*, 18:185–195, may 2014.

[120] Agoston E. Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[121] Martin Drozdik, Hernan Aguirre, Youhei Akimoto, and Kiyoshi Tanaka. Comparison of Parameter Control Mechanisms in Multi-objective Differential Evolution. In *Learning and Intelligent Optimization*, pages 89–103. Springer, 2015.

[122] Junhong Liu and Jouni Lampinen. A Fuzzy Adaptive Differential Evolution Algorithm. *Soft Computing*, 9(6):448–462, 2005.

[123] A Kai Qin, Vicky Ling Huang, and Ponnuthurai N Suganthan. Differential evolution algorithm with strategy adaptation for global numerical optimization. *IEEE Transactions on Evolutionary Computation*, 13(2):398–417, 2009.

[124] Vicky Ling Huang, Shuguang Z Zhao, Rammohan Mallipeddi, and Ponnuthurai N Suganthan. Multi-objective optimization using self-adaptive differential evolution algorithm. *IEEE Congress on Evolutionary Computation*, pages 190–194, 2009.

[125] Jeff Clune, Sherri Goings, Erik D. Goodman, and William Punch. Investigations in Meta-GAs: Panaceas or Pipe Dreams? In *GECOO'05: Proceedings of the 2005 Workshop on Genetic an Evolutionary Computation*, pages 235–241, 2005.

[126] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1–31, 2009.

[127] Rok Sosic and Jun Gu. Effcient Local Search with Conflict Minimization: A Case Study of the N-Queens Problem. *IEEE Transactions on Knowledge and Data Engineering*, 6:661–668, 1994.

Part II

Appendix

A

French summary

This appendix contains a summary of the thesis in French language.

Contents

A.1 Introduction

L'optimisation combinatoire a plusieurs applications dans différents domaines tels que l'apprentissage de la machine, l'intelligence artificielle, et le génie du logiciel. Dans certains cas, le but principal est seulement de trouver une solution, comme pour les Problèmes de Satisfaction de Contraintes (CSP). Une solution sera une affectation de variables répondant aux contraintes fixées, en d'autres termes: une solution faisable.

Plus formellement, un CSP (dénoté par $\mathcal{P}$) est défini par le trio $\langle X, D, C\rangle$ où $X = \{x_1, x_2, \ldots, x_n\}$ est un ensemble fini de variables; $D = \{D_1, D_2, \ldots, D_n\}$, est l'ensemble des domaines associés à chaque variable dans X; et $C = \{c_1, c_2, \ldots, c_m\}$, est un ensemble de contraintes. Chaque contrainte est définie en impliquant un ensemble de variables, et spécifie les combinaisons possibles de valeurs de ces variables. Une configuration $s \in D_1 \times D_2 \times \cdots \times D_n$, est une combinaison de valeurs des variables dans X. Nous disons que s est une solution de $\mathcal{P}$ si et seulement si s satisfait toutes les contraintes $c_i \in C$.

Les *CSP*s sont connus pour être des problèmes extrêmement difficiles. Parfois les méthodes complètes ne sont pas capables de passer à l'échelle de problèmes de taille industrielle. C'est la raison pour laquelle les techniques méta–heuristiques sont de plus en plus utilisées pour la résolution de ces derniers. Par contre, dans la plupart des cas industriels, l'espace de recherche est assez important et devient donc intraitable, même pour les méthodes méta-heuristiques. Cependant, les récents progrès dans l'architecture de l'ordinateur nous conduisent vers les ordinateurs *multi/many–cœur*, en proposant une nouvelle façon de trouver des solutions à ces problèmes d'une manière plus réaliste, ce qui réduit le temps de recherche.

Grâce à ces développements, les algorithmes parallèles ont ouvert de nouvelles façons de résoudre les problèmes de contraintes: Adaptive Search [5] est un algorithme efficace, montrant de très bonnes performances et passant à l'échelle de plusieurs centaines ou même milliers de cœurs, en utilisant la recherche locale *multi-walk* en parallèle. Munera et al. [6] ont présenté une autre implémentation d'Adaptive Search en utilisant la coopération entre des stratégies de recherche. *Meta–S* est une implémentation d'un cadre théorique présenté dans [105], qui permet d'attaquer les problèmes par la coopération de solveurs de contraintes de domaine spécifique. Ces travaux ont montré l'efficacité du schéma parallèle multi-walk.

De plus, le temps de développement nécessaire pour coder des solveurs en parallèle est souvent sous-estimé, et dessiner des algorithmes efficaces pour résoudre certains problèmes consomme trop de temps. Dans cette thèse nous présentons POSL, un langage orienté parallèle pour construire des solveurs de contraintes basés sur des méta-heuristiques, qui résolvent des *CSP*s. Il fournit un mécanisme pour dessiner des stratégies de communication.

L'autre but de cette thèse est de présenter une analyse détaillée des résultats obtenus en résolvant plusieurs instances des problèmes CSP. Sachant que créer des solveurs utilisant différentes stratégies de solution peut être compliqué et pénible, POSL donne la possibilité de faire des prototypes de solveurs communicants facilement.

A.2 Des travaux reliés

Beaucoup de chercheurs se concentrent sur la programmation par contraintes, particulièrement dans le développement de solution à haut-niveau, qui facilite la construction de stratégies de recherche. Cela permet de citer plusieurs contributions.

Hyperion [62] est un système codé en Java pour méta et hyper-heuristiques basé sur le principe d'interopérabilité, fournissant des patrons génériques pour une variété d'algorithmes de recherche locale et évolutionnaire, et permettant des prototypages rapides avec la possibilité de réutiliser le code source. POSL offre ces avantages, mais il fournit également un mécanisme permettant de définir des protocoles de communication entre solveurs. Il fournit aussi, à travers d'un simple langage basé sur des opérateurs, un moyen de construire des abstract solvers, en combinant des modules déjà définis (computation modules et communication modules). Une idée similaire a été proposée dans [7] sans communication, qui introduit une approche évolutive en utilisant une simple composition d'opérateurs pour découvrir automatiquement les nouvelles heuristiques de recherche locale pour SAT et les visualiser comme des combinaisons d'un ensemble de blocs.

Récemment, [68] a montré l'efficacité de combiner différentes techniques pour résoudre un problème donné (hybridation). Pour cette raison, lorsque les composants du solveur peuvent être combinés, POSL est dessiné pour exécuter en parallèle des ensembles de solveurs différents, avec ou sans communication. Une autre idée intéressante est proposée dans Templar. Il s'agit d'un système qui génère des algorithmes en changeant des composants prédéfinis, et en utilisant des méthodes hyper-heuristiques [66]. Dans la dernière phase du processus de codage avec POSL, les solveurs peuvent être connectés les uns aux autres, en fonction de la structure de leurs communication modules, et de cette façon, ils peuvent partager non seulement des informations, mais aussi leur comportement, en partageant leurs computation modules. Cette approche donne aux solveurs la capacité d'évoluer au cours de l'exécution.

Renaud De Landtsheer et al. présentent dans [8] un cadre facilitant le développement des systèmes de recherches en utilisant des *combinators* pour dessiner les caractéristiques trouvées très souvent dans les procédures de recherches comme des briques et les assembler. Dans [9] est proposée une approche qui utilise des systèmes coopératifs de recherche locale basée sur des méta-heuristiques. Celle-ci se sert de protocoles de transfert de messages. POSL combine ces deux idées pour assembler des composants de recherche locale à travers des opérateurs fournis (ou en créant des nouveaux), mais il fournit aussi un mécanisme basé sur opérateurs pour connecter et combiner des solveurs, en créant des stratégies de communication.

Dans cette thèse, nous présentons quelques nouveaux opérateurs de communication afin de concevoir des stratégies de communication. Avant de clore ce résumé par une brève conclusion, nous présentons quelques résultats obtenus en utilisant POSL pour résoudre certaines instances des problèmes *Social Golfers*, *Costas Array*, *N-Queens* et *Golomb Ruler*.

A.3 Solveurs parallèles POSL

POSL permet de construire des solveurs suivant différentes étapes :

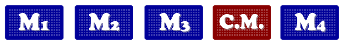
1. L'algorithme du solveur considéré est exprimé via une décomposition en modules de calcul. Ces modules sont implémentés à la manière de *fonctions* séparées. Nous appelons *computation module* ces morceaux de calcul (figure A.1a, blocs bleus). En suite, il faut décider quelles sont les types d'informations que l'on souhaite recevoir des autres solveurs. Ces informations sont encapsulées dans des composants appelés *communication module*, permettant de transmettre des données entre solveurs (figure A.1a, bloc rouge)

2. Une *stratégie générique* est codée à travers POSL, en utilisant les opérateurs fournis par le langage appliqués sur des modules *abstraite* qui représentent les *signatures* des composants donnés lors l'étape 1., pour créer *abstract solvers*. Cette stratégie définit non seulement les informations échangées, mais détermine également l'exécution parallèle de composants. Lors de cette étape, les informations à partager sont transmises via les opérateurs ad-hoc. On peut voir cette étape comme la définition de la colonne vertébrale des solveurs (figure A.1b).

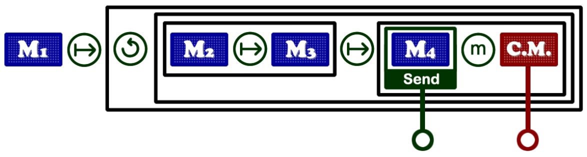
3. Les solveurs sont créés en instanciant l'abstract solver, par computation modules et communication module.

4. Les solveurs sont assemblés en utilisant les opérateurs de communication fournis par le langage, pour créer des stratégies de communication. Cette entité finale s'appelle *solver set* (figure A.1c).

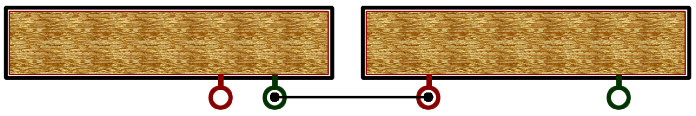
Les sous-sections suivantes expliquent en détail chacune des étapes ci-dessus.



(a) Définition des modules et les communication modules



(b) Définition de l'abstract solver



(c) Définition de la stratégie de communication

Figure A.1: Construire des solveurs parallèles avec POSL

A.3.1 Computation module

Un computation module est la plus basique et abstraite manière de définir un composant de calcul. Il reçoit une entrée, exécute un algorithme interne et retourne une sortie. Dans ce résumé, nous utilisons ce concept afin de décrire et définir les composants de base d'un solveur, qui seront assemblés par l'abstract solver.

Un computation module représente un morceau de l'algorithme du solveur qui est susceptible de changer au cours de l'exécution. Il peut être dynamiquement remplacé ou combiné avec d'autres computation modules, puisque les computation modules sont également des informations échangeables entre les solveurs. De cette manière, le solveur peut changer/adapter son comportement à chaud, en combinant ses computation modules avec ceux des autres solveurs. Ils sont représentés par des blocs bleus dans la figure A.1.

Définition 1 ***(Computation Module)*** *Un computation module $\mathcal{O}m$ est une application définie par :*

$$\mathcal{C}m : \mathcal{I} \to \mathcal{O} \tag{A.1}$$

Dans (1), la nature de $\mathcal{D}$ et $\mathcal{I}$ dépend du type de computation module. Ils peuvent être soit une configuration, ou un ensemble de configurations, ou un ensemble de valeurs de différents types de données, etc.

Soit une méta-heuristique de recherche locale, basée sur un algorithme bien connu, comme par exemple *Tabu Search*. Prenons l'exemple d'un computation module retournant le voisinage d'une configuration donnée, pour une certaine métrique de voisinage. Ce computation module peut être défini par la fonction suivante:

$$\mathcal{C}m : D_1 \times D_2 \times \cdots \times D_n \rightarrow 2^{D_1 \times D_2 \times \cdots \times D_n} \tag{A.2}$$

où D_i représente la définition des domaines de chacune des variables de la configuration d'entrée.

A.3.2 Communication module

Les communication modules sont les composants des solveurs en charge de la réception des informations communiquées entre solveurs. Ils peuvent interagir avec les computation modules, en fonction de l'abstract solver. Les communication modules jouent le rôle de prise, permettant aux solveurs de se brancher et de recevoir des informations. Il sont représentés en rouge dans la figure A.1a.

Un communication module peut recevoir deux types d'informations, provenant toujours d'un solveur tiers : des données et des computation modules. En ce qui concerne les computation modules, leur communication peut se faire via la transmission d'identifiants permettant à chaque solveur de les instancier.

Pour faire la distinction entre les deux différents types de communication modules, nous appelons *data communication module* les communication modules responsables de la réception de données et *object communication module* ceux s'occupant de la réception et de l'instanciation de computation modules.

Définition 2 ***(Data communication module)*** *Un data communication module $\mathcal{C}h$ est un composant produisant une application définie comme suit :*

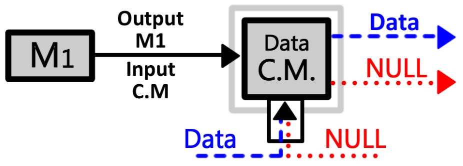
$$\mathcal{C}h : I \times \{D \cup \{NULL\}\} \rightarrow D \cup \{NULL\} \tag{A.3}$$

et retournant l'information $\mathcal{I}$ provenant d'un solveur tiers,quelque soit l'entrée $\mathcal{U}$.

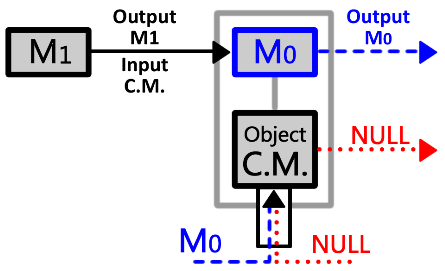
Définition 3 ***(Object communication module)*** *Si nous notons* $\mathbf{M}$ *l'espace de tous les computation modules de la définition 1, alors un object communication module $\mathcal{C}h$ est un composant produisant un computation module venant d'un solveur tiers défini ainsi :*

$$\mathcal{C}h : I \times \{\mathbf{M} \cup \{NULL\}\} \rightarrow O \cup \{NULL\} \tag{A.4}$$

Puisque les communication modules reçoivent des informations provenant d'autres solveurs sans pour autant avoir de contrôle sur celles-ci, il est nécessaire de définir l'information



(a) Data communication module



(b) Object communication module

Figure A.2: Mécanisme interne du communication module

NULL, signifiant l'absence d'information. La figure A.2 montre le mécanisme interne d'un communication module. Si un data communication module reçoit une information, celle-ci est automatiquement retournée (figure A.2a, lignes bleues). Si un object communication module reçoit un computation module, ce dernier est instancié et exécuté avec l'entrée du communication module, et le résultat est retourné (figure A.2b, lignes bleues). Dans les deux cas, si aucune information n'est reçue, le communication module retourne l'objet *NULL* (figure A.2, lignes rouges).

A.3.3 Abstract solver

L'abstract solver est le cœur du solveur. Il joint les computation modules et les communication modules de manière cohérente, tout en leur restant indépendant. Ceci signifie qu'il peut changer ou être modifié durant l'exécution, sans altérer l'algorithme général et en respectant la structure du solveur. À travers l'abstract solver, on peut décider également des informations à envoyer aux autres solveurs. Chaque fois que nous combinons certains composants en utilisant des opérateurs POSL, nous créons un *module*.

Définition 4 *Noté par la lettre $\mathcal{M}$, un* **module** *est:*

1. *un computation module; ou*

2. *un communication module; ou*

3. *$[OP\ \mathcal{M}]$, la composition d'un module $\mathcal{M}$ exécutée séquentiellement, en retournant une sortie, en dépendant de la nature de l'opérateur unaire* OP*; ou*

4. *$[\mathcal{M}_1\ OP\ \mathcal{M}_2]$, la composition de deux modules $\mathcal{M}_1$ et $\mathcal{M}_2$ exécutée séquentiellement, en retournant une sortie, en dépendant de la nature de l'opérateur binaire* OP.

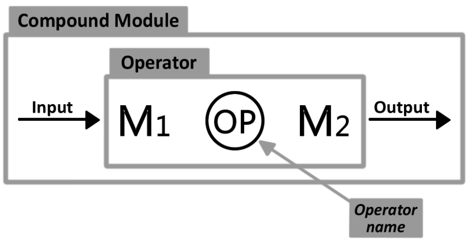


Figure A.3: Un compound module

5. *[$\mathcal{M}_1$ OP $\mathcal{M}_2$], la composition de deux modules $\mathcal{M}_1$ et $\mathcal{M}_2$ exécutée, en retournant une sortie, en dépendant de la nature de l'opérateur binaire* OP*. Ces deux opérateurs vont être exécutes en parallèle si et seulement si* OP *supporte le parallélisme, ou bien il lance une exception en cas contraire.*

Nous notons par **M** *l'espace des modules, et nous appelons compound modules à la composition de modules présentés en 3. 4., et/ou 5..*

Pour illustrer la définition 4, la figure A.3 montre graphiquement le concept de compound module.

Dans le cas particulier où un des compound modules impliqués est un communication module, chaque opérateur gère l'information *NULL* à sa manière.

Afin de grouper des modules, nous utiliserons la notation $|.|$ comme un groupe générique qui pourra être indifféremment interprété comme $[.]$ ou comme $[\![.]\!]_p$.

Ensuite, les opérateurs fournis par POSL sont présentés.

Sequential Execution Operator: L'opération $\left|\mathcal{M}_1 \ \textcircled{\mapsto}\ \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{seq}$ comme le résultat de l'exécution de $\mathcal{M}_1$ suivi de $\mathcal{M}_2$. C'est un exemple d'opérateur ne supportant pas une exécution en parallèle de ses compound modules impliqués, puisque l'entrée du second compound module est la sortie du premier.

Conditional Execution Operator: L'opération $\left|\mathcal{M}_1 \textcircled{?}_{<cond>} \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{cond}$ le résultat de l'exécution en séquentiel de $\mathcal{M}_1$ si $< cond >$ es **vrai** ou $\mathcal{M}_2$, autrement.

Cyclic Execution Operator: L'opération $|\circlearrowleft_{<cond>} \mathcal{M}|$ définit le compound module $\mathcal{M}_{cyc}$ en répétant séquentiellement l'exécution de $\mathcal{M}$ tant que $< cond >$ est **vrai**.

Random Choice Operator: L'opération $\left|M_1 \textcircled{\rho} \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{rho}$ qui exécute $\mathcal{M}_1$ en suivant une probabilité ρ, ou en exécutant $\mathcal{M}_2$ en suivant une probabilité $(1-\rho)$.

Not *NULL*: L'opération $\left|\mathcal{M}_1 \textcircled{\vee} \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{non}$ qui exécute $\mathcal{M}_1$ et retourne une sortie si elle n'est pas *NULL*, ou exécute $\mathcal{M}_2$ et retourne une sortie autrement.

Minimum Operator: Soient o_1 et o_2 les sorties de $\mathcal{M}_1$ et $\mathcal{M}_2$, respectivement. Nous assumons qu'il existe un ordre total dans $I_1 \cup I_2$ où l'objet *NULL* est la plus grand valeur. Alors, l'opération $\left|\mathcal{M}_1 \text{Ⓜ} \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{min}$ qui exécute $\mathcal{M}_1$ et $\mathcal{M}_2$, et retourne $\min\{o_1, o_2\}$.

Maximum Operator: Soient o_1 et o_2 les sorties de $\mathcal{M}_1$ et $\mathcal{M}_2$, respectivement. Nous assumons qu'il existe un ordre total dans $I_1 \cup I_2$ où l'objet *NULL* est la plus petite valeur. Alors, l'opération $\left|\mathcal{M}_1 \text{Ⓜ} \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{max}$ qui exécute $\mathcal{M}_1$ et $\mathcal{M}_2$, et retourne $\max\{o_1, o_2\}$.

Race Operator: L'opération $\left|\mathcal{M}_1 \textcircled{\downarrow} \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{race}$ qui exécute les deux modules $\mathcal{M}_1$ et $\mathcal{M}_2$, et retourne la sortie du module qui termine en premier

Les opérateurs $\textcircled{\rho}$, $\textcircled{\vee}$ et $\textcircled{m}$ sont très utiles en terme de partage d'informations entre solveurs, mais également en terme de partage de comportements. Si un des opérandes est un communication module alors l'opérateur peut recevoir le computation module d'un autre solveur, donnant la possibilité d'instancier ce module dans le solveur le réceptionnant. L'opérateur va soit instancier le module s'il n'est pas *NULL* et l'exécuter, soit exécuter le module donné par le second opérande.

Maintenant, nous présentons les opérateurs nous permettant d'envoyer de l'information vers d'autres solveurs. Deux types d'envois sont possibles : i) on exécute un module et on envoie sa sortie, ii) ou on envoie le module lui-même.

Sending Data Operator: L'opération $\left|(\!|\mathcal{M}|\!)^d\right|$ définit le compound module $\mathcal{M}_{sendD}$ qui exécute le module $\mathcal{M}$ puis envoie la sortie vers un communication module.

Sending Module Operator: L'opération $|(\!|\mathcal{M}|\!)^m|$ définit le compound module $\mathcal{M}_{sendM}$ qui exécute le module $\mathcal{M}$, puis envoie le module lui même vers un communication module.

Avec les opérateurs présentés jusqu'ici, nous sommes en mesure de concevoir les abstract solvers (ou algorithme) de résolution d'un problème de contraintes. Une fois un tel abstract solver définie, on peut changer les composants (computation modules et communication modules) auxquels elle fait appel, permettant ainsi d'implémenter différents solveurs à partir du même abstract solver mais composés de différents modules, du moment que ces derniers respectent la signature attendue, à savoir le types des entrées et sorties.

Un abstract solver est déclaré comme suit: après déclarer les noms de l'**abstract solver** (*name*), la première ligne définit la liste des computation modules abstraites ($\mathcal{L}^m$), la seconde ligne, la liste des communication modules abstraites ($\mathbf{M}$), puis l'algorithme du solver est définit comment le corps su solver (the root compound module $\mathbf{M}$), entre **begin** et **end**.

Un abstract solver peut être déclaré par l'expression régulière suivante:

abstract solver *name* **computation**: $\mathcal{L}^m$ (**communication**: $\mathcal{L}^c$)? **begin** **M** **end**

Par exemple, l'algorithme 1 montre l'abstract solver correspondant à la figure A.1b.

Algorithm 1: Pseudo-code POSL pour l'abstract solver de la figure A.1b

```
abstract solver as_01
computation : $I, V, S, A$
connection: $C.M.$
  begin
    $I \circledcirc{\mapsto} \left[ \circlearrowleft (\textsc{Itr} < K_1) \left[ V \circledcirc{\mapsto} S \circledcirc{\mapsto} \left[ C.M. \; \textcircled{m} \; (\!|A|\!)^d \right]\right] \right]$
  end
```

A.3.4 Créer les solveurs

Maintenant on peut créer les solveurs en instanciant les modules. Il est possible de faire ceci en spécifiant qu'un **solver** donné doit implémenter (en utilisant le mot clé **implements**) un abstract solver donné, suivi par la liste de computation puis communication modules. Ces modules doivent correspondre aux signatures exigées par l'abstract solver.

Algorithm 2: Une instanciation de l'abstract solver présenté dans l'algorithme 1

```
solver solver_01 implements as_01
computation : $I_{rand}, V_{1ch}, S_{best}, A_{AI}$
connection: $CM_{last}$
```

A.3.5 Connecter les solveurs : créer le solver set

La dernière étape est de connecter les solveurs entre eux. POSL fournit des outils pour créer des stratégies de communication très facilement. L'ensemble des solveurs connectés qui seront exécutés en parallèle pour résoudre un CSP s'appelle *solver set*.

Les communications sont établies en respectant les règles suivantes :

1. À chaque fois qu'un solveur envoie une information via les opérateurs $(\!|.|\!)^d$ ou $(\!|.|\!)^m$, il créé une *prise mâle de communication*

2. À chaque fois qu'un solveur contient un communication module, il crée une *prise femelle de communication*

3. Les solveurs peuvent être connectés entre eux en reliant *prises mâles* et *femelles*.

Avec l'opérateur $(\cdot)$, nous pouvons avoir accès aux computation modules envoyant une information et aux noms des communication modules d'un solveur. Par exemple : $Solver_1 \cdot \mathcal{M}_1$ fournit un accès au computation module $\mathcal{M}_1$ du $Solver_1$ si et seulement s'il est utilisé par l'opérateur $(\!|.|\!)^d$ (ou $(\!|.|\!)^m$), et $Solver_2 \cdot \mathcal{C}h_2$ fournit un accès au communication module $\mathcal{C}h_2$ de $Solver_2$.

Maintenant, nous définissons les opérateurs de communication que POSL fournit.

Définition 5 Connection One-to-One Operator *Soient*

1. $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *une liste de prises mâles, et*
2. $\mathcal{O} = [\mathcal{Z}_0 \cdot \mathcal{CM}_0, \mathcal{Z}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{Z}_{N-1} \cdot \mathcal{CM}_{N-1}]$ *une liste de prises femelles*

Alors, l'opération

$$\mathcal{J} \boxed{\rightarrow} \mathcal{O}$$

connecte chaque prise mâle $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ *avec la prise femelle correspondante* $\mathcal{Z}_i \cdot \mathcal{CM}_i \in \mathcal{O}$, $\forall\ 0 \leq i \leq N-1$ *(voir figure A.4a).*

Définition 6 Connection One-to-N Operator *Soient*

1. $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *une liste de prises mâles, et*
2. $\mathcal{O} = [\mathcal{Z}_0 \cdot \mathcal{CM}_0, \mathcal{Z}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{Z}_{M-1} \cdot \mathcal{CM}_{M-1}]$ *une liste de prises femelles*

Alors, l'opération

$$\mathcal{J} \boxed{\rightsquigarrow} \mathcal{O}$$

connecte chaque prise mâle $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ *avec chaque prise femelle* $\mathcal{Z}_j \cdot \mathcal{CM}_j \in \mathcal{O}$, $\forall\ 0 \leq i \leq N-1$ *et* $0 \leq j \leq M-1$ *(see Figure A.4b).*

POSL permet aussi de déclarer des solveurs non communicatifs pour les exécuter en parallèle, en déclarant seulement la liste des noms:

$$[\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{N-1}]$$

A.4 Les résultats

Le but principal de cette section est de sélectionner quelques instances de problèmes de référence, pour analyser et illustrer la versatilité de POSL pour étudier des stratégies de solution basées sur la recherche locale méta-heuristique avec communication. Grâce à POSL

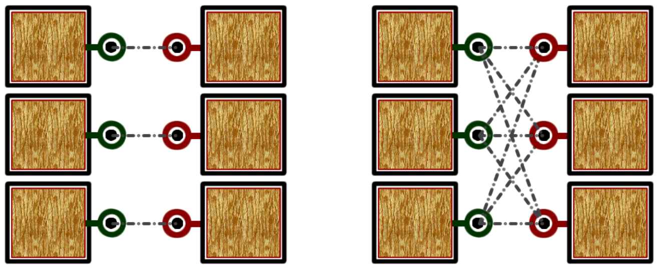

(a) Communication 1 à 1 (b) Communication 1 à N

Figure A.4: Représentation graphique des opérateurs de communication

nous pouvons analyser des résultats et formuler des conclusions sur le comportement de la stratégie de recherche, mais aussi sur la structure de l'espace de recherche du problème. Dans cette section, nous expliquons la structure des solveurs de POSL que nous avons générés pour les expériences, et les résultats.

Nous avons choisi l'une des méthodes de solutions les plus classiques pour des problèmes combinatoires: l'algorithme méta-heuristique de recherche locale. Ces algorithmes ont une structure commune: ils commencent par l'initialisation des structures de données. Ensuite, une configuration initiale s est générée. Après cela, une nouvelle configuration s' est sélectionnée dans le voisinage $V(s)$. Si s' est une solution pour le problème P, alors le processus s'arrête, et s' est renvoyée. Dans le cas contraire, les structures de données sont mises à jour, et s' est acceptée, ou non, pour l'itération suivante, en fonction de certains critères (par exemple, en pénalisant les caractéristiques des optimums locaux).

Les expériences ont été effectuées sur un processeur Intel ® Xeon ™ E5-2680 v2, 10×4 cœurs, 2.80GHz. Les résultats montrés dans cette section sont les moyennes de 30 runs pour chaque configuration. Dans les tableaux de résultats, les colonnes marquées **T** correspondent au temps de l'exécution en secondes et les colonnes marquées **It.** correspondent au nombre d'itérations. Toutes les expériences de cette section sont basées sur différentes stratégies en parallèle, avec 40 cœurs.

A.4.1 Social Golfers Problem

Le problème de *Social Golfers* (*SGP*) consiste à planifier $n = g \times p$ golfeurs en g groupes de p joueurs chaque semaine pendant w semaines, de telle manière que deux joueurs jouent dans le même groupe au plus une fois. Une instance de ce problème peut être représentée par le triplet $g - p - w$. Ce problème, et d'autres problèmes étroitement liés, trouvent de nombreuses applications pratiques telles que le codage, le cryptage et les problèmes couvrants. Sa structure nous a semblé intéressante car elle est similaire à d'autres problèmes, comme

Kirkman's Schoolgirl et *Steiner Triple System*, donc nous pouvons construire des modules efficaces pour résoudre un grand éventail de problèmes.

Nous avons utilisé une stratégie de communication cyclique pour résoudre ce problème, en échangeant la configuration courante entre deux solveurs avec des caractéristiques différentes. Les résultats montrent que cette stratégie marche très bien pour ce problème.

Algorithm 3: Solveur pour *SGP*

```
abstract solver as_eager                                   // ITR → nombre d'itérations
computation : $I, V, S_1, S_2, A$                          // SCI → nombre d'itérations avec le même coût
begin
    $I \; \textcircled{$\mapsto$} \; [\textcircled{$\circlearrowleft$}(\text{ITR} < K_1) \; V \; \textcircled{$\mapsto$} \; \left[S_1 \; \textcircled{?}_{\text{SCI}\%K_2} \; S_2\right] \; \textcircled{$\mapsto$} \; A \;]$
end
solver SOLVER$_{eager}$ implements as_eager
    computation : $I_{BP}, V_{BAS}, S_{first}, S_{rand}, A_{AI}$
```

L'algorithme 3 montre l'abstract solver utilisé pour résoudre me manière séquentielle le *SGP*. L'utilisation de deux modules de sélection (S_1 et S_2) est un simple chamanisme pour éviter les minimums locaux: il tente d'améliorer le coût un certain nombre de fois, en exécutant le computation module S_1. S'il n'y arrive pas, il exécute le computation module S_2. L'abstract solver a été instancié par les computation modules suivantes:

1. S_{BP} génère une configuration aléatoire s, en respectant la structure du problème, c'est-à-dire que la configuration est un ensemble de w permutations du vecteur $[1..n]$.

2. V_{BAS} définit le voisinage $V(s)$ permutant le joueur qui a contribué le plus au coût, avec d'autres joueurs dans la même semaine.

3. S_{rirst} sélectionne la première configuration $s' \in V(s)$ qui améliore le coût actuel, et retourne (s, s')

4. S_{rand} sélectionne une configuration aléatoire $s' \in V(s)$, et retourne (s, s')

5. A_{AI} retourne toujours la configuration sélectionnée (s').

Pour *SGP*, nous avons utilisé une stratégie de communication, où un solveur "compagnon", incapable de trouver une solution au final, mais capable de trouver des configurations avec un coût considérablement plus petit que celui trouvé par le solveur *standard* dans le même instant de temps, au début de la recherche. L'idée est d'échanger leurs configurations cycliquement, jusqu'à trouver une solution. Les algorithmes 4 et 5 montrent les solveurs utilisés pour cette stratégie, où $V_{BP}(p)$ est le computation module de voisinage pour le solveur "compagnon", qui cherche des configurations seulement en changeant des joueurs parmi p semaines. Le communication module instancié CM_{last}, prend en compte la dernière configuration reçue quand il est au moment de l'exécution.

Algorithm 4: Solveur standard pour *SGP*

```
abstract solver $as\_standard$
computation : $I, V, S_1, S_2, A$
communication : $C.M.$
begin
    $I \textcircled{\mapsto} [\textcircled{\circlearrowleft} (\textsc{Itr} < K_1)\ V \textcircled{\mapsto} \left[S_1 \textcircled{?}_{\textsc{Sci}\%K_2} S_2\right] \textcircled{\mapsto} \left[C.M. \textcircled{m} (|A|)^d\right]\ ]$
end
solver $\textsc{Solver}_{standard}$ implements as_standard
    computation : $I_{BP}, V_{BAS}, S_{first}, S_{rand}, A_{AI}$
    communication : $CM_{last}$
```

Algorithm 5: Solveur compagnon pour *SGP*

```
abstract solver $as\_compagnon$
computation : $I, V, S_1, S_2, A$
communication : $C.M.$
begin
    $I \textcircled{\mapsto} [\textcircled{\circlearrowleft} (\textsc{Itr} < K_1)\ V \textcircled{\mapsto} \left[S_1 \textcircled{?}_{\textsc{Sci}\%K_2} S_2\right] \textcircled{\mapsto} \left[C.M. \textcircled{\vee} (|A|)^d\right]\ ]$
end
solver $\textsc{Solver}_{compagnon}$ implements as_compagnon
    computation : $I_{BP}, V_{BP}(p), S_{first}, S_{rand}, A_{AI}$
    communication : $CM_{last}$
```

Nous avons dessiné aussi différentes stratégies de communication, en combinant des solveurs connectés et non-connectés, et en appliquant différents opérateurs de communication : one to one et one to N.

Instance	**Séquentielle**		**Parallèle**		**Coopérative**	
	T	It.	T	It.	T	It.
5–3–7	1.25	2,903	0.23	144	**0.10**	98
8–4–7	0.60	338	0.28	93	**0.14**	54
9–4–8	1.04	346	0.59	139	**0.36**	146

Table A.1: Résultats pour *SGP*

Comme nous nous y attendions, le tableau A.1 confirme le succès de l'approche parallèle sur le séquentiel. De plus, les expériences confirment que la stratégie de communication proposée pour cet benchmark est la correcte: en comparant par rapport aux runs en parallèle sans communication, il améliore les runtimes par un facteur de 1.98 (facteur moyen parmi les trois instances). Les résultats coopératifs de ce tableau on été obtenus en utilisant l'opérateur de communication one to one avec 100% de solveurs communicatifs (algorithme 6).

Algorithm 6: Stratégie 100% de communication *compagnon*

```
$[\textsc{Solver}_{compagnon} \cdot A] \boxed{\rightarrow} [\textsc{Solver}_{standard} \cdot C.M.]\ 20;$
$[\textsc{Solver}_{standard} \cdot A] \boxed{\rightarrow} [\textsc{Solver}_{compagnon} \cdot C.M.]\ 20;$
```

A.4.2 Costas Array Problem

Le problème *Costas Array* (*CAP*) consiste à trouver une matrice *Costas*, qui est une grille de $n \times n$ contenant n marques avec exactement une marque par ligne et par colonne et les $n(n-1)/2$ vecteurs reliant chaque couple de marques de cette grille doivent tous être différents. Ceci est un problème très complexe trouvant une application utile dans certains domaines comme le sonar et l'ingénierie de radar, et présente de nombreux problèmes mathématiques ouverts. Ce problème a aussi une caractéristique intéressante: même si son espace de recherche grandit factoriellement, à partir de l'ordre 17 le nombre de solutions diminue drastiquement.

Pour ce problème nous avons testé une stratégie de communication simple, où l'information à communiquer est la configuration courante. Pour construire les solveurs, nous avons réutilisé les computation modules de sélection (S_{first}) et de d'acceptation (A_{AI}) et le communication module utilisés dans la résolution de *SGP*. Les autres computation modules sont les suivants :

1. I_{perm}: génère une configuration aléatoire s, comme une permutation du vecteur $[1..n]$.
2. V_{AS}: définit le voisinage $V(s)$ permutant la variable qui a contribué le plus au coût, avec d'autres.

Pour résoudre *CAP* nous avons eu besoin d'utiliser un computation module de *reset* (T_{AS}) comme machinisme d'exploration. L'algorithme 7 montre le solveur utilisé pour résoudre ce problème séquentiellement. Les résultats des runs en séquentiel et en parallèle sans communication sont montrés dans le tableau A.2. Ils montrent le succès de l'approche en parallèle. Afin d'améliorer ces résultats, nous avons appliqué une stratégie simple de communication : communiquer la configuration courante au moment d'exécuter le critère d'acceptation. Les algorithmes 8 et 9 montrent les solveurs envoyeur et récepteur.

Algorithm 7: Solveur pour *CAP*

```
abstract solver as_hard
computation : $I, T, V, S, A$
begin
    $I \textcircled{\mapsto} [\textcircled{\circlearrowleft} (\textsc{Itr} < K_1)\ T \textcircled{\mapsto} [\textcircled{\circlearrowleft} (\textsc{Itr}\ \%\ K_2) \left[V \textcircled{\mapsto} S \textcircled{\mapsto} A\right]\ ]\ ]$
end
solver $\textsc{Solver}_1$ implements as_hard
    computation : $I_{perm}, T_{AS}, V_{AS}, S_{first}, A_{AI}$
```

L'un des buts principaux de cette étude a été d'explorer différentes stratégies de communication. Nous avons ensuite mis en place et testé différentes variantes de la stratégie exposée

Algorithm 8: Solveur envoyeur pour *CAP*

```
abstract solver $as\_hard\_sen$
computation : $I, T, V, S, A$
begin
    $I \; \textcircled{$\mapsto$} \; [\textcircled{$\circlearrowleft$} (\textsc{Itr} < K_1) \; T \; \textcircled{$\mapsto$} \; [\textcircled{$\circlearrowleft$} (\textsc{Itr} \,\%\, K_2) \left[V \; \textcircled{$\mapsto$} \; S \; \textcircled{$\mapsto$} \; (\!|A|\!)^d\right] \; ] \; ]$
end
solver $\textsc{Solver}_{sender}$ implements as_hard_sen
    computation : $I_{perm}, T_{AS}, V_{AS}, S_{first}, A_{AI}$
```

Algorithm 9: Solveur récepteur pour *CAP*

```
abstract solver $as\_hard\_rec$
computation : $I, T, V, S, A$
communication : $C.M.$
begin
    $I \; \textcircled{$\mapsto$} \; [\textcircled{$\circlearrowleft$} (\textsc{Itr} < K_1) \; T \; \textcircled{$\mapsto$} \; [\textcircled{$\circlearrowleft$} (\textsc{Itr} \,\%\, K_2) \left[V \; \textcircled{$\mapsto$} \; S \; \textcircled{$\mapsto$} \; \left[A \; \textcircled{$m$} \; C.M.\right]\right] \; ] \; ]$
end
solver $\textsc{Solver}_{receiver}$ implements as_hard_rec
    computation : $I_{perm}, T_{AS}, V_{AS}, S_{first}, A_{AI}$
    communication: $CM_{last}$
```

STRATÉGIE	T	It.	% success
Séquentielle	132.73	2,332,088	40.00
Parallèle	25.51	231,262	100.00
Coopérative	**10.83**	**79,551**	100.00

Table A.2: Résultats pour *CAP* 19

ci-dessus en combinant deux opérateurs de communication (one to one et one to N) et des pourcentages différents de solveurs communicants.

Comme prévu, la meilleure stratégie était basée sur 100% de communication avec l'opérateur one to N (algorithme 10), parce que cette stratégie permet de communiquer un lieu prometteur à l'intérieur de l'espace de recherche à un maximum de solveurs, en reforçant l'intensification.

Algorithm 10: Stratégie de communication one to N 100% pour *CAP*

```
$[\textsc{Solver}_{sender} \cdot A(20)] \boxed{\rightsquigarrow} [\textsc{Solver}_{receiverA} \cdot C.M.(20)]$ ;
```

A.4.3 N-Queens Problem

Le problème *N-Queens* (*NQP*) demande de placer N rennes sur un échiquier, de manière a ce qu'aucune d'elles ne puisse attaquer une autre en un seul mouvement. C'est un problème introduit en 1848 par le joueur d'échecs Max Bezzelas comme le problème *8-Queens*, et un an après il a été généralisé comme le problème *N-Queens* par Franz Nauck. Depuis sa création, plusieurs mathématiciens, Gauss inclut, ont travaillé sur ce problème. Il a beaucoup d'applications, par exemple, dans le stockage en parallèle de la mémoire, le contrôle du trafique, la prévention des deadlocks, les réseaux neurales, etc. [126]. Quelques études suggèrent que le nombre de solutions augmente exponentiellement en fonction du nombre de rennes (N), mais les méthodes de recherche locale ont montré des bons résultats avec ce problème [127]. C'est pour cela que nous avons testé quelques stratégies de communication en utilisant POSL, pour résoudre un problème relativement facile, mais un utilisant la communication.

Pour construire les solveurs, nous avons réutilisé presque tous les computation modules utilisés pour résoudre le problème *Costas Array* (S_{first}, S_{first}, A_{AI}, et aussi le communication module CM_{last}. Les solveurs utilisés pour les expériences sans communication sont présentés dans l'algorithme 11, ou l'abstract solver est instancié dans le solveurs $\textsc{Solver}_{selective}$ avec le computation module de voisinage $V_{PAS}(p)$, qui reçoit une configuration, et retourne un voisinage $V(s)$ en permutant la variable qui a contribué le plus au coût, avec un pourcentage p des autres.

Algorithm 11: Solveur simple pour *NQP*

```
abstract solver as_simple
computation : $I, V, S, A$
begin
    $I \textcircled{\mapsto} [\textcircled{\circlearrowleft} (\textsc{Itr} < K_1)\ V \textcircled{\mapsto} S \textcircled{\mapsto} A\ ]$
end
solver $\textsc{Solver}_{selective}$ implements as_simple
    computation : $I_{perm}, V_{PAS}(p), S_{first}, A_{AI}$
```

Le tableau A.3 présente les résultats des runs en séquentiel et en parallèle. Ces résultats montrent que l'amélioration de l'approche en parallèle par rapport à l'approche séquentielle en utilisant POSL diminue au fur et à mesure que l'ordre du problème augmente.

Pour appliquer l'approche coopérative a la résolution de ce problème, nous avons implémenté une stratégie de communication similaire à celle appliquée avec *SGP*, mais dans ce cas, avec des solveurs qui utilisent le même module de voisinage $V_{PAS}(p)$ mais avec une valeur différente de p et un module de sélection différent. Dans cette stratégie de communication la configuration courante est échangée cycliquement entre les solveurs différents. Un solveur *compagnon* utilise le computation module $V_{PAS}(p)$ avec une valeur plus petite pour p ainsi que le computation module S_{best}, donc capable de trouver des configurations prometteuses plus rapidement, mais avec une convergence lente. L'autre solveur est très similaire au solveur utilisé pour les expériences sans communication, mais dans cette stratégie de communication, les solveurs sont à la fois des envoyeurs et des récepteurs (voir l'algorithme 12).

Grâce à cette expérience nous avons été capables de trouver une stratégie de communication

Instance	**Sequential**		**Parallel**	
	T	It.	T	It.
250	0.29	8,898	0.19	4,139
500	0.35	4,203	0.24	2,675
1000	0.35	2,766	0.30	2,102
3000	1.50	2,191	1.33	2,168
6000	4.71	3,339	4.57	3,323

Table A.3: Résultats pour *NQP* (séquentielle et en parallèle sans communication)

Algorithm 12: Des solveurs cycliques pour *NQP*

```
**abstract solver** $as\_cyc$
**computation** : $I, V, S_1, S_2, A$
**communication** : $C.M.$
**begin**
    $I \circled{\mapsto} [\circled{\circlearrowleft} (\textsc{Itr} < K_1)\ V \circled{\mapsto} S \circled{\mapsto} \left[A \circled{?}_{\textsc{Itr}\%K_2} \left[(\!|A|\!)^d \circled{m}\ C.M.\right]\right] ]$
**end**
**solver** $\textsc{Solver}_{standard}$ **implements** as_cyc
    **computation** : $I_{perm}, V_{PAS}(2.5), S_{first}, A_{AI}$
    **communication**: $CM_{last}$
**solver** $\textsc{Solver}_{compagnon}$ **implements** as_cyc
    **computation** : $I_{perm}, V_{PAS}(1), S_{best}, A_{AI}$
    **communication**: $CM_{last}$
```

Instance	**Communication 1-1**		**Communication 1-n**		**I.R.**
	T	It.	T	It.	
250	**0.09**	1,169	0.10	1,224	2.00
500	**0.14**	864	0.15	977	1.65
1000	0.22	889	**0.21**	807	1.39
3000	1.25	1,602	**1.02**	1,613	1.17
6000	4.83	2,938	**4.24**	2,537	1.01

Table A.4: Résultats de la communication cyclique avec *NQP*

(algorithme 13) pour améliorer les temps d'exécution, mais seulement pour des instances petites du problème. Ce résultat confirme l'hypothèse que quand l'ordre du problème monte, le gain en utilisant la communication pendant la recherche de la solution diminue. Le tableau A.4 montre comment l'*improvement ratio* (colonne **I.R.**) diminue avec l'ordre N.

A.4.4 Golomb Ruler Problem

Le *Golomb Ruler Problem* (*GRP*) consiste à trouver un vecteur ordonné de n entiers non négatifs différents, appelés *marques*, $m_1 < \cdots < m_n$, tel que toutes les différences $m_i - m_j$, $(i > j)$ sont toutes différentes. Une instance de ce problème est définie par la paire (o, l) où o est l'ordre du problème, (le nombre de *marques*) et l est la longueur de la règle (la dernière *marque*). Nous supposons que la première *marque* est toujours 0. Lorsque nous appliquons POSL pour résoudre une instance de problème séquentiellement, nous pouvons remarquer qu'il effectue de nombreux *restars* avant de trouver une solution. Pour cette raison, nous avons choisi ce problème pour étudier une stratégie de communication intéressante: communiquer la configuration actuelle afin d'éviter son voisinage, c'est à dire, une configuration *tabu*.

Nous réutilisons les modules de sélection et d'acceptation des études antérieures (S_{first} etA_{AI}) pour concevoir les abstract solvers. Les nouvelles modules sont:

1. I_{sort}: renvoie une configuration aléatoire s en tant que vecteur d'entiers trié. La configuration est générée *loin* de l'ensemble des configurations *tabu* arrivées via communication entre solveurs.

2. V_{sort}: donné une configuration, retourne le voisinage en changeant une valeur tout en gardant l'ordre, à savoir, le remplacement de la valeur s_i par toutes les valeurs possibles $s'_i \in D_i$ en satisfaisant $s_{i-1} < s'_i < s_{i+1}$.

Nous avons également ajouté un module de reset T: il reçoit et renvoie une configuration. Le computation module utilisé pour l'instancier (T_{tabu}) insère la configuration reçue dans une liste *tabu* à l'intérieur du solveur et retourne la configuration d'entrée telle quelle. L'algorithme 14 présente le solveur utilisé pour envoyer des informations (solveur envoyeur).

Algorithm 14: Solveur envoyeur pour *GRP*

```
abstract solver as_golomb_sender
computation : $I, V, S, A, T$
begin
    $[\circlearrowleft (\textsc{Itr} < K_1)\ I \circledmapsto [\circlearrowleft (\textsc{Itr}\ \%\ K_2) \left[V \circledmapsto S \circledmapsto A\right] ] \circledmapsto (\!|T|\!)^d\ ]$
end
solver $\textsc{Solver}_{sender}$ implements as_golomb_sender
    computation : $I_{sort}, V_{sort}, S_{first}, A_{AI}, , T_{tabu}$
```

Le module T_{tabu} est exécuté lorsque le solveur est incapable de trouver une meilleure configuration autour de l'actuelle: elle est supposée être un minimum local, et elle est envoyée au solveur récepteur. L'algorithme 15 présente un solveur utilisé pour recevoir l'information. Le communication module CM_{set} reçoit plusieurs configurations qui sont reçues par le computation module I_{sort} comme entrées.

Algorithm 15: Solveur récepteur pour *GRP*

```
abstract solver $as\_golomb\_receiver$
computation : $I, V, S, A, T$
connection : $C.M.$
begin
    $[\circlearrowright (\textsc{Itr} < K_1) \left[C.M. \; \textcircled{\mapsto} \; I\right] \; \textcircled{\mapsto} \; [\circlearrowright (\textsc{Itr} \; \% \; K_2) \left[V \; \textcircled{\mapsto} \; S \; \textcircled{\mapsto} \; A\right] ] \; \textcircled{\mapsto} \; T \; ]$
end
solver $\textsc{Solver}_{receiver}$ implements as_golomb_receiver
    computation : $I_{sort}, V_{sort}, S_{first}, A_{AI}, , T_{tabu}$
    communication: $CM_{set}$
```

Le bénéfice de l'approche en parallèle avec POSL est aussi prouvé pour le *GRP* (voir le tableau A.5). Dans ce tableau, la colonne **R** représente le nombre de redémarrages exécutés. Cette expérience a été réalisée en utilisant des solveurs similaires à ceux présentés précédemment, mais sans communication modules.

Instance	**Séquentiel**				**Parallèle**		
	T	It.	R	% success	T	It.	R
8–34	0.66	10,745	53	100.00	0.43	349	1
10–55	67.89	446,913	297	88.00	4.92	20,504	13
11–72	117.49	382,617	127	30.00	85.02	155,251	51

Table A.5: Résultats non coopératifs pour *GRP*

Pour *GRP*, la stratégie de communication que nous avons adopté a été différente. L'idée de cette stratégie est de profiter des nombreux redémarrages indiqués dans le tableau A.5. Chaque fois qu'un solveur redémarre, la configuration actuelle est communiquée pour alerter les solveurs et éviter son voisinage. De cette façon, chaque fois qu'un solveur redémarre, il génère une nouvelle configuration assez loin de ces "zones pénalisées".

Sur la base de l'opérateur de connexion utilisé dans la stratégie de communication, ce solveur peut recevoir une ou plusieurs configurations. Ces configurations sont l'entrée du module de génération (I_{sort}). Ce module insère toutes les configurations reçues dans une liste *tabu*, puis il génère une nouvelle première configuration loin de toutes les configurations dans la liste *tabu*.

Instance	**Communication one to one**			**Communication one to N**		
	T	It.	R	T	It.	R
8–34	0.44	309	1	**0.43**	283	1
10–55	3.90	15,437	10	**3.16**	12,605	8
11–72	85.43	156,211	52	**60.35**	110,311	36

Table A.6: Résultats avec communication pour *GRP*.

Comme nous pouvons voir dans le tableau A.6 l'amélioration en temps d'exécution avec communication est plus visible quand on utilise l'opérateur de communication one to N (algorithme 16), parce-que à chaque nouvelle itération, le solveur récepteur a plus d'information afin de générer une nouvelle configuration loin des "zones pénalisées".

Algorithm 16: Stratégie de communication one to N pour *GRP*

```
$[\textsc{Solver}_{sender} \cdot R(20)] \boxed{\rightsquigarrow} [\textsc{Solver}_{receiver} \cdot C.M.(20)]$ ;
```

A.5 Conclusions

Dans cette thèse, nous avons présenté POSL, un système pour construire des solveurs parallèles coopératifs. Il propose une manière modulable pour créer des solveurs capables d'échanger n'importe quel type d'informations, comme par exemple leur comportement même, en partageant leurs computation modules. Avec POSL, de nombreux solveurs différents pourront être créés et lancés en parallèle, en utilisant une unique stratégie générique mais en instanciant différents computation modules et communication modules pour chaque solveur.

Il est possible d'implémenter différentes stratégies de communication, puisque POSL fournit une couche pour définir les canaux de communication connectant les solveurs entre eux.

Nous avons présenté aussi des résultats en utilisant POSL pour résoudre des instances des problèmes classiques CSP. Il a été possible d'implémenter différentes stratégies communicatives et non communicatives, grâce au langage basé sur des opérateurs fournis, pour combiner différents computation modules. POSL donne la possibilité de relier dynamiquement des solveurs, étant capable de définir des stratégies différentes en terme de pourcentage de solveurs communicatifs. Les résultats montrent la capacité de POSL à résoudre ces problèmes, en montrant en même temps que la communication peut jouer un rôle décisif dans le processus de recherche.

POSL a déjà une importante bibliothèque de computation modules et de communication modules prête à utiliser, sur la base d'une étude approfondie sur les algorithmes méta-heuristiques classiques pour la résolution de problèmes combinatoires. Dans un avenir proche, nous prévoyons de la faire grandir, afin d'augmenter les capacités de POSL.

En même temps, nous prévoyons d'enrichir le langage en proposant de nouveaux opérateurs. Il est nécessaire, par exemple, d'améliorer le langage de *définition du solveur*, pour permettre la construction plus rapide et plus facile des ensembles de nombreux nouveaux solveurs. En plus, nous aimerions élargir le langage des opérateurs de communication, afin de créer des stratégies de communication polyvalentes et plus complexes, utiles pour étudier le comportement des solveurs.