POSL: A Parallel-Oriented Solver Language

Thesis for the Degree of
Doctor of Computer Science

Alejandro Reyes Amaro

Doctoral School STIM

Academic advisors:
Eric Monfroy[1], Florian Richoux[2]

[1]Department of Informatics
Faculty of Science
University of Nantes
France

[2]Department of Informatics
Faculty of Science
University of Nantes
France

Submitted: dd/mm/2016

Assessment committee:

Prof. (1)

Institution (1)

Prof. (2)

Institution (2)

Prof. (3)

Institution (3)

ISBN ??

Part I

POSL: Parallel Oriented Solver Language

1

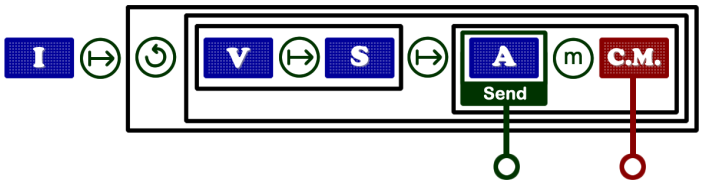
A Parallel-Oriented Language for Modeling Meta-Heuristic-Based Solvers

In this chapter POSL *is introduced as the main contribution, and a new way to solve CSPs. Its characteristics and advantages are summarized, and a general procedure to be followed is described, in order to build parallel solvers using* POSL*, followed by a detailed description of each of the single steps.*

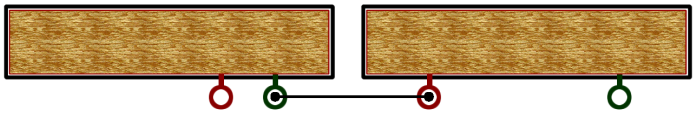
Contents

In this chapter we present the different steps to build communicating parallel solvers with POSL. First of all, the algorithm we have conceived to solve the target problem is decomposed it into small modules of computation, which are implemented as separated *functions*. We name them *computation modules* (see Figure 1.1a, blue shapes). At this point it is crucial to find a good decomposition of its algorithm, because it will have a significant impact in its future re-usage and variability. The next step is to decide which information is interesting to *receive* from other solvers. This information is encapsulated into another kind of component called *communication module*, allowing data transmission between solvers (see Figure 1.1a, red shapes). A third stage is to ensemble the modules through POSL's inner language (the interested reader is referred to Appendix [...]) to create independent solvers. The parallel-oriented language based on operators provided by POSL (see Figure 1.1b, green shapes) allows not only the information exchange, but also executing components in parallel. In this stage the information that is interesting to be shared with other solvers is sent using operators. After that we can connect them using *communication operators*. We call this final entity a *solvers set* (see Figure 1.1c).

(a) Creating POSL's modules



(b) Assembling modules using POSL's operators



(c) Connecting POSL solvers to create *communication strategies*

Figure 1.1: Solver construction process using POSL

In the following sections all these steps are explained in details, but first, I explain how to model the target benchmark using POSL.

1.1 Modeling the target benchmark

Target problems are modeled in POSL using the C++ programing language, respecting some rules of the object-oriented design. First of all, the benchmark must inherit from the class **Benchmark** provided by POSL. This class does not have any method to be

overridden or implemented, but receives in its constructor three objects, instances from classes that the user must create. Those classes must inherit from **SolutionCostStrategy**, **RelativeCostStrategy** and **ShowStrategy**, respectively. In these classes the most important functionalities of the benchmark model are defined.

SolutionCostStrategy: In this class the strategy to compute the *cost* of a configuration is implemented. POSL is based on improving step by step an initial configuration, taking into account a *cost function* provided by the user through the model (by implementing the function *solutionCost(dots)*). The kind of problems that POSL solves is the class of *Constraint Satisfaction Problems*, so this *cost function* must return an integer taking into account the problem constraints. Given a configuration s, the *cost function*, as a mandatory rule, must return 0 if and only if s is a solution of the problem, i.e., s fulfill all the problem constraints. An example of *cost function* is one that returns the number of violated constraints. However, the more expressive the function cost is, the better the performance of POSL leading to the solution.

The method to be implemented in this class is:

- `int solutionCost(std::vector<int> & c)` → Computes the cost of a given configuration (`c`).

RelativeCostStrategy: In this class the user implements the strategy to compute the *cost* of a given configuration with respect to another. If the cost of some configuration has been calculated, sometimes it is possible to store some information in order to compute the cost of another configuration, if the differences between them are known. If it is possible, the algorithms is defined in this class. If it is not possible, this class must have the same functionality of **SolutionCostStrategy**.

The methods to implement in this class are:

- `void initializeCostData(std::vector<int> & c)` → Initializes the information related to the cost (auxiliary data structures, the current configuration (`c`), the current cost, etc.)
- `void updateConfiguration(std::vector<int> & c)` → Updates the information related to the cost.
- `int relativeSolutionCost(std::vector<int> & c)` → Returns the relative cost of the configuration `c` with respect to the current configuration.
- `int currentCost()` → Property that returns the cost of the current configuration.
- `int costOnVariable(int variable_index)` → Returns a measure of the contribution of a variable to the total cost of a configuration.

- `int sickestVariable()` → Returns the variable contributing the most to the cost.

SolutionCostStrategy: This class represents the way a benchmark shows a configuration, in order to provide more information about the structure. For example, a configuration of the instance 3–3–2 of the *Social Golfers Problem* (see bellow for more details about this benchmark) can be written as follows:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 3, 4, 5, 6, 7, 8, 9, 1, 2]
```

This text is, nevertheless, very difficult to be read if the instance is larger. Therefore, it is recommended that the user implements this class in order to give more details and to make it easier to interpret the configuration. For example, for the same instance of the problem, a solution could be presented as follows:

```
Golfers: players-3, groups-3, weeks-2
6         8         7
1         3         5
4         9         2
--
7         2         3
4         8         1
5         6         9
--
```

The method to be implemented in this class is:

- `std::string showSolution(std::shared_ptr<Solution> s)` → Returns a string to be written in the standard output.

Once we have modeled the target benchmark, it can be solved using POSL. In the following sections we describe how to use this parallel-oriented language to solve *Constraint Satisfaction Problems*.

1.2 First stage: creating POSL's modules

There exist two types of basic modules in POSL: *computation modules* and *communication modules*. A *computation module* is a function which received an input, then executes an internal algorithm, and returns an output. A *communication module* is also a function receiving and returning information, but in contrast, the *communication module* can receive information from two different sources: through input parameters or from outside, i.e., by communicating with a module from another solver.

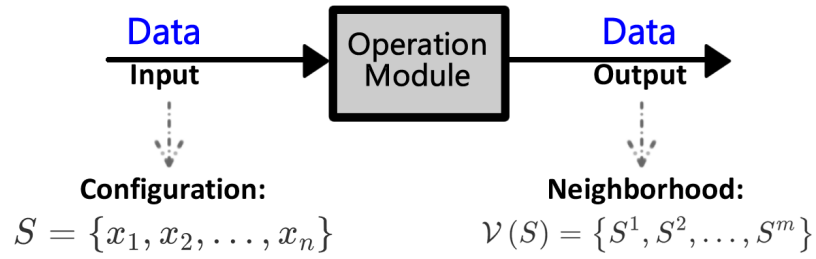


Figure 1.2: An example of a computation module computing a neighborhood

1.2.1 Computation Module

A *computation module* is the most basic and abstract way to define a piece of computation. It is a function which receives an instance of a POSL data type as input, then executes an internal algorithm, and returns an instance of a POSL data type as output. The input and output types will characterize the computation module signature. It can be dynamically replaced by (or combined with) other computation modules, since they can be shared among solvers working in parallel. They are joined through *abstract solvers*.

Definition 1 ***(Computation Module)*** *A computation module Cm is a mapping defined by:*

$$Cm : D \rightarrow I \tag{1.1}$$

where D and I can be either a set of configurations, a set of sets of configurations, a set of values of some data type, etc.

Consider a local search meta-heuristic solver. One of its *computation modules* can be the function returning the set of configurations composing the neighborhood of a given configuration:

$$Cm_{neighborhood} : D_1 \times D_2 \times \cdots \times D_n \rightarrow 2^{D_1 \times D_2 \times \cdots \times D_n}$$

where D_i represents the definition domains of each variable of the input configuration.

Figure 1.2 shows an example of *computation module*: which receives a configuration S and then computes the set $\mathcal{V}$ of its neighbor configurations $\{S^1, S^2, \ldots, S^m\}$.

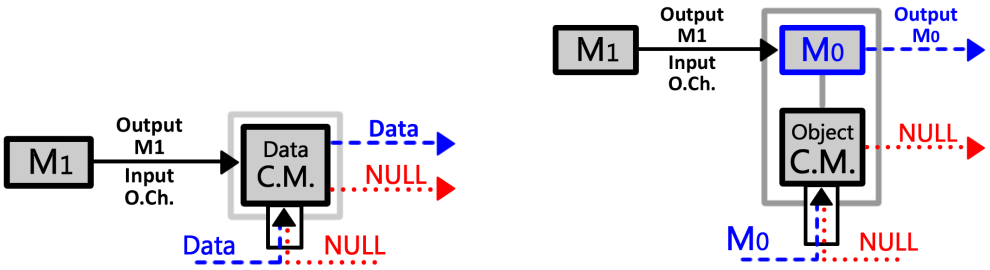
1.2.1.1 Creating new *computation modules*

To create new *computation modules* we use C++ programing language. POSL provides a hierarchy of data types to work with (See anexes) and some abstract classes to inherit from, depending on the type of *computation module* that the user wants to create. These abstract classes represent *abstract computation module* and define a type of action to be executed. In the following we present the most important ones:

- **AOM_FirstConfigurationGeneration** → Represents *computation modules* generating a first configuration. The user must implement the method *spcf_execute(**ComputationData**)* which returns a pointer to a **Solution**, that is, an object containing all the information concerning a partial solution (configuration, variable domains, etc.)
- **AOM_NeighborhoodFunction** → Represent *computation modules* creating a neighborhood of a given configuration. The user must implement the method *spcf_execute(**Solution**)* which returns a pointer to an object **Neighborhood**, containing a set of configurations which constitute the neighborhood of a given configuration, according to certain criteria. These configuration are efficiently stored.
- **AOM_SelectionFunction** → Represents *computation modules* selecting a configuration from a neighborhood. The user must implement the method *spcf_execute(**Neighborhood**)* which returns a pointer to an object **DecisionPair**, containing two solutions: the current and the selected one.
- **AOM_DecisionFunction** → Represents *computation modules* deciding which of the two solutions will be the current configuration for the next iteration. The user must implement the method *spcf_execute(**DecisionPair**)* which returns a pointer to an object **Solution**.

1.2.2 Communication modules

A *communication module* is also a function receiving and returning information, but in contrast, the *communication module* can also receive information by communicating with a module from another solver. A *communication module* is the component managing the information reception in the communication between solvers (we will talk about information transmission in the next section). They can interact with *computation modules* through operators (see Figure 1.3).



(a) Data *communication module* **(b)** Object *communication module*

Figure 1.3: Communication module

A *communication module* can receive two types of information from an external solver: data or *computation modules*. It is important to notice that by sending/receiving *computation modules*, we mean sending/receiving only required information to identify and being able to instantiate the *computation module*.

In order to distinguish from the two types of *communication modules*, we will call Data Communication Module to the *communication module* responsible for the data reception (Figure 1.3a), and Object Communication Module to the one responsible for the reception and instantiation of *computation modules* (Figure 1.3b).

Definition 2 ***(Data Communication Module)*** *A* Data Communication Module *Ch is a component that produces a mapping defined as follows:*

$$Ch : U \rightarrow I \tag{1.2}$$

It returns the information I coming from an external solver, no matter what the input U is.

Definition 3 ***(Object Communication Module)*** *If we denote by* $\mathbb{M}$ *the space of all the computation modules defined by Definition 1.1, then an* Object Communication Module *Ch is a component that produces a computation module coming from an external solver as follows:*

$$Ch : \mathbb{M} \rightarrow \mathbb{M} \tag{1.3}$$

Users can implement new computation and connection modules but POSL already contains many useful modules for solving a broad range of problems.

Due to the fact that *communication modules* receive information coming from outside without having control on them, it is necessary to define the *NULL* information, in order to denote the absence of information. If a Data Communication Module receives a piece of information, is returned automatically. If a Object Communication Module receives a *computation module*, it is instantiated and executed with the *communication module*'s input and its result is

returned. In both cases, if no available information exists (no communications performed), the *communication module* returns the *NULL* object.

1.3 Second stage: assembling POSL's modules

Modules mentioned above are defined respecting the signature of some predefined abstract module. For example, the module showed in Figure 1.2 is a *computation module* based on an abstract module that receives a configuration and returns a neighborhood. In that sense, an example of a concrete *computation module* (or just *computation module*) can be a function receiving a configuration, and returning a neighborhood constituted by N configurations which only differ from the input configuration in one entry.

In this stage an *abstract solver* is coded using POSL. It takes abstract modules as *parameters* and combines them through operators. Through the *abstract solver*, we can also decide which information to send to other solvers by using some operators to send the result of a computation module (see below). In the following we present a formal and more detailed specification of POSL's operators.

The *abstract solver* is the solver's backbone. It joins the *computation modules* and the *communication modules* coherently. It is independent from the *computation modules* and *communication modules* used in the solver. It means that they can be changed or modified during the execution, without altering the general algorithm, but still respecting the main structure. Each time we combine some of them using POSL's operators, we are creating a *compound module*. Here we formally define the concept of *module* and *compound module*.

Definition 4 *A* **module** *is (and it is denoted by the letter $\mathcal{M}$):*

a) a computation module or

b) a communication module or

c) $[\mathcal{M}_1\ OP\ \mathcal{M}_2]$, which is the composition of two modules $\mathcal{M}_1$ and $\mathcal{M}_2$ to be executed sequentially, returning an output depending on the nature of the operator OP*; or*

d) $[\![\mathcal{M}_1\ OP\ \mathcal{M}_2]\!]_p$, which is the composition of two modules $\mathcal{M}_1$ and $\mathcal{M}_2$ to be executed, returning an output depending on the nature of the operator OP*. These two modules will be executed in parallel if and only if* OP *supports parallelism, (i.e. some modules will be executed sequentially although they were grouped this way); or sequentially otherwise.*

We denote the space of the modules by $\mathbb{M}$ *and call compound modules to the composition of modules described in c) and d).*

For a better understanding of Definition 4, Figure 1.4 shows graphically the structure of a *compound module*.

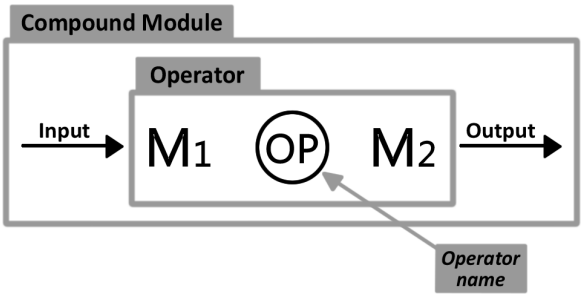


Figure 1.4: A *compound module*

As mentioned before, the *abstract solver* is independent from the *computation modules* and *communication modules* used in the solver. It means that one *abstract solver* can be used to construct many different solvers, by implementing it using different modules (see below the related concept of *abstract solver* instantiation). This is the reason why the *abstract solver* is defined only using abstract modules. Formally, we define an *abstract solver* as follows:

Definition 5 ***(Abstract Solver)*** *An* Abstract Solver *AS is a triple* $(\mathbf{M}, \mathcal{L}^m, \mathcal{L}^c)$*, where:* $\mathbf{M}$ *is a compound module (also called* root *compound module),* $\mathcal{L}^m$ *a list of abstract computation modules appearing in* $\mathcal{M}$*, and* $\mathcal{L}^c$ *a list of communication modules appearing in* $\mathcal{M}$*.*

The *root compound module* can be defined also as a free-context grammar as follows:

Definition 6 (root *compound module***'s grammar)** $G_{POSL} = (\mathbf{V}, \Sigma, \mathbf{S}, \mathbf{R})$*, where:*

a) $\mathbf{V} = \{CM, OP\}$ *is the set of variables,*

b) $\Sigma = \left\{\alpha, \beta, be, [,], [\![,]\!]_p, (,), \{,\}, (\!|,|\!)^m, |\!)^o, \textcircled{\mapsto}, \textcircled{?}, \circlearrowleft, \textcircled{\rho}, \textcircled{\vee}, \textcircled{\wedge}, \textcircled{M}, \textcircled{m}, \textcircled{\downarrow}, \textcircled{\cup}, \textcircled{\cap}\right\}$ *is the set of terminals,*

c) $\mathbf{S} = \{CM\}$ *is the set of start variables,*

d) and $\mathbf{R} =$

$$CM \longmapsto \alpha \mid \beta \mid (\!|CM|\!)^o \mid (\!|CM|\!)^m \mid [OP] \mid [\![OP]\!]_p$$
$$OP \longmapsto CM \textcircled{\mapsto} CM \mid CM \textcircled{?} CM \mid CM \textcircled{\rho} CM \mid CM \textcircled{\vee} CM \mid CM \textcircled{\wedge} CM$$
$$OP \longmapsto CM \textcircled{M} CM \mid CM \textcircled{m} CM \mid CM \textcircled{\downarrow} CM \mid CM \textcircled{\cup} CM \mid CM \textcircled{\cap} CM$$
$$OP \longmapsto CM \circlearrowleft (be) CM$$

is a set of rules

In the following I explain some of the concepts in Definition 6:

- The variables CM and OP are two very important entities in the language, as it can be seen in the grammar. We name them *compound module* and *operator*, respectively.
- The terminals α and β represent a *computation module* and a *communication module*, respectively.
- The terminal *be* is a boolean expression.
- The terminals $[\ ], [\![\ ]\!]_p$ are symbols for grouping and defining the way the involved *compound modules*are executed. Depending on the nature of the operator, this can be either sequentially or in parallel:
  - *a)* [OP]: The involved operator is executed sequentially.
  - *b)* $[\![\text{OP}]\!]_p$: The involved operator is executed in parallel if and only if OP supports parallelism. Otherwise, an exception is thrown.
- The terminals (and) are symbols for grouping the boolean expression in some operators.
- The terminals $(\!|.|\!)^m, (\!|.|\!)^o$, are operators to send information to other solvers (explained bellow).
- The rest of terminals are POSL operators.

In the following we define POSL operators. In order to group modules, like in Definition $4(c)$) and $4(d)$), we will use |.| as generic grouper. In order to help the reader to easely understand how to use the operators, I use an example of a solver that I build step by step, while presenting the definitions.

POSL creates solvers based on local search meta-heuristics algorithms. These algorithms have a common structure: 1. They start by initializing some data structures (e.g., a *tabu list* for *Tabu Search* [34], a *temperature* for *Simulated Annealing* [32], etc.). 2. An initial configuration s is generated. 3. A new configuration s' is selected from the neighborhood $\mathcal{V}(s)$. 4. If s' is a solution for the problem P, then the process stops, and s' is returned. If not, the data structures are updated, and s' is accepted or not for the next iteration, depending on a certain criterion. An example of such data structure is the penalizing features of local optima defined by Boussaïd et al [31] in their algorithm *Guided Local Search*.

Abstract computation modules composing *local search meta–heuristics* are:

$Abstract\ Computation\ module - 1$	I: Generating a configuration s
$Abstract\ Computation\ module - 2$	V: Defining the neighborhood $\mathcal{V}(s)$

Abstract Computation module − 3	S: Selecting $s' \in \mathcal{V}(s)$
Abstract Computation module − 4	A: Evaluating an acceptance criterion for s'

The list of modules to be used in the examples have been presented. Now I present the POSL operators.

Definition 7 (Operator Sequential Execution) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be modules, where $\mathcal{I}_1 \subseteq \mathcal{D}_2$. *Then the operation* $\left|\mathcal{M}_1 \textcircled{\mapsto} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{seq}$ *as the result of executing* $\mathcal{M}_1$ *followed by executing* $\mathcal{M}_2$*:*

$$\mathcal{M}_{seq} : \mathcal{D}_1 \rightarrow \mathcal{I}_2$$

This is an example of an operator that does not support the execution of its involved *compound modules* in parallel, because the input of the second *compound module* is the output of the first one.

Coming back to the example, I can use defined *abstract computation modules* to create a *compound module* that perform only one iteration of a local search, using the operator **Sequential Execution**. I create a *compound module* to execute sequentially I and V (see Figure 1.5a), then I create an other *compound module* to execute sequentially the *compound module* already created and S (see Figure 1.5b), and finally this *compound module* and the *computation module* A are executed sequentially (see Figure 1.5c). The *compound module* presented in Figure 1.5c can be coded as follows:

$$\Big[\Big[\Big[I \textcircled{\mapsto} V\Big] \textcircled{\mapsto} S\Big] \textcircled{\mapsto} A\Big]$$

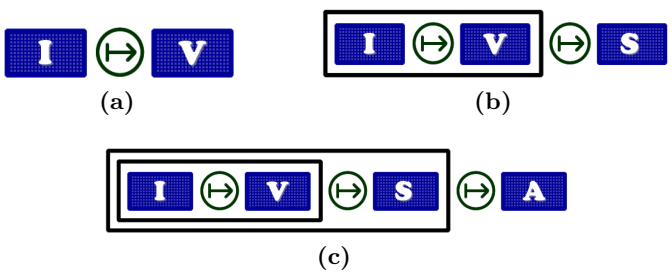


Figure 1.5: Using **sequential execution** operator

The following operator is very useful to execute modules sequentially creating bifurcations, subject to some boolean condition:

Definition 8 (Operator Conditional Execution) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Then the operation* $\left|\mathcal{M}_1 \textcircled{?}_{<cond>} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{cond}$ *as result of the sequential execution of* $\mathcal{M}_1$ *if* $< cond >$ *is* **true** *or* $\mathcal{M}_2$, *otherwise:*

$$\mathcal{M}_{cond} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

This operator can be used in the example if I want to execute two different *selection computation modules* (S_1 and S_2) depending on certain criterion (see Figure 1.6):

$$\left[\left[\left[I \textcircled{\mapsto} V\right] \textcircled{\mapsto} \left[S_1 \textcircled{?} S_2\right]\right] \textcircled{\mapsto} A\right]$$

In examples I remove the clause $< cond >$ for simplification.

Figure 1.6: Using **conditional execution** operator

We can execute modules sequentially creating also cycles.

Definition 9 (Operator Cyclic Execution) *Let* $\mathcal{M} : \mathcal{D} \rightarrow \mathcal{I}$ *be a module, where* $\mathcal{I} \subseteq \mathcal{D}$. *Then, the operation* $|\circlearrowleft_{<cond>} \mathcal{M}|$ *defines the compound module* $\mathcal{M}_{cyc}$ *as result of the sequential execution of* $\mathcal{M}$ *repeated while* $< cond >$ *remains* **true**:

$$\mathcal{M}_{cyc} : \mathcal{D} \rightarrow \mathcal{I}$$

Using this operator I can model a local search algorithm, by executing the *abstract computation module* I and then the other *computation modules* (V, S and A) cyclically, until finding a solution (i.e, a configuration with cost equal to zero) (see Figure 1.7):

$$\left[I \textcircled{\mapsto} \left[\circlearrowleft \left[\left[V \textcircled{\mapsto} S\right] \textcircled{\mapsto} A\right]\right]\right]$$

In the examples, I remove the clause $< cond >$ for simplification.

Definition 10 (Operator Random Choice) *Let*

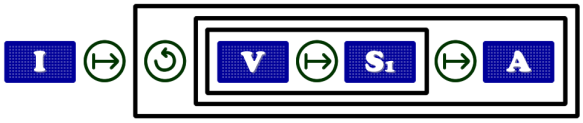


Figure 1.7: Using **cyclic execution** operator

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be modules, where $\mathcal{D}_1 \subset \mathcal{D}_2$ *and a real value* ρ*. Then the operation* $\left|M_1 (\rho) \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{rho}$ *that executes and returns the output of* $\mathcal{M}_1$ *with probability* ρ*, or executes and returns the output of* $\mathcal{M}_2$ *with probability* $(1 - \rho)$*:*

$$\mathcal{M}_{rho} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

In the example I can create a *compound module* to execute two *abstract computation modules* A_1 and A_2 following certain probability ρ using the operator **random execution** as follows (see Figure 1.8):

$$\left[I \mapsto \left[\circlearrowleft \left[\left[V \mapsto S\right] \mapsto \left[A_1 \ (\rho) \ A_2\right]\right]\right]\right]$$

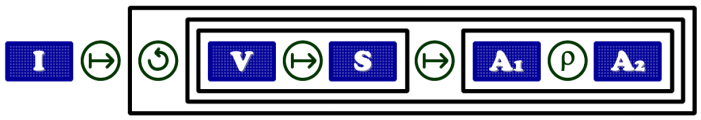


Figure 1.8: Using **random execution** operator

The following operator is very useful if the user needs to use a *communication module* inside an *abstract solver*. As explained before, if a *communication module* does not receive any information from another solver, it returns *NULL*. This may cause the undesired termination of the solver if this case is not considered correctly. Next, I introduce the operator **Operator Not** *NULL* **Execution** and illustrate how to use it in practice with an example.

Definition 11 (Operator Not *NULL* **Execution)** *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Then, the operation* $\left|\mathcal{M}_1 (\vee) \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{non}$ *that executes* $\mathcal{M}_1$ *and returns its output if it is not NULL, or executes* $\mathcal{M}_2$ *and returns its output otherwise:*

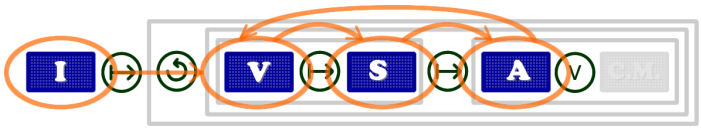
$$\mathcal{M}_{non} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Let us make consider a slightly more complex example: When applying the acceptance criterion, suppose that we want to receive a configuration from other solver to combine the *computation module* A with a *communication module*:

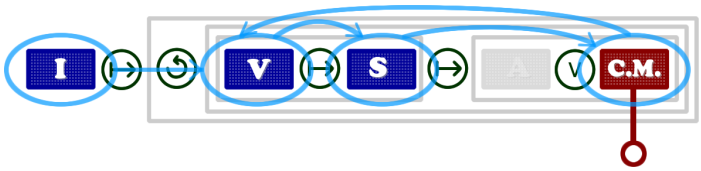
$\boxed{Communication\ module - 1:}$ $C.M.$: Receiving a configuration.

Figure 1.9 shows how to combine a *communication module* with the *computation module* A through the operator $\bigcirc\!\!\!\!\vee$. Here, the *computation module* A will be executed as long as the *communication module* remains *NULL*, i.e., there is no information coming from outside. This behavior is represented in Figure 1.9a by the orange lines. If some data has been received through the *communication module*, the later is executed instead of the module A, represented in Figure 1.9b by blue lines. The code can be written as follows:

$$\left[I \bigcirc\!\!\!\!\mapsto \left[\circlearrowleft \left[\left[V \bigcirc\!\!\!\!\mapsto S\right] \bigcirc\!\!\!\!\mapsto \left[A \bigcirc\!\!\!\!\vee C.M.\right]\right]\right]\right]$$



(a) The solver executes the computation module **A** if no information is received through the connection module



(b) The solver uses the information coming from an external solver

Figure 1.9: Two different behaviors within the same solver

This is *short-circuit* operator. It means that if the first argument (module) does not return *NULL*, the second will not be executed. POSL provides another operator with the same functionality but not *short-circuit*:

Definition 12 (Operator *BOTH* Execution) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Then the operation* $\left|\mathcal{M}_1 \bigcirc\!\!\!\!\wedge \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{both}$ *that executes both* $\mathcal{M}_1$ *and* $\mathcal{M}_2$, *then returns the output of* $\mathcal{M}_1$ *if it is not NULL, or the output of* $\mathcal{M}_2$ *otherwise:*

$$\mathcal{M}_{both} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

In the following definitions, the concepts of *cooperative parallelism* and *competitive parallelism* are implicitly included. We say that cooperative parallelism exists when two or more processes are running separately, they are independent, and the general result will be some combination of the results of all the involved processes (e.g. Definitions 13 and 14). On the other hand, competitive parallelism arise when the general result is the result of the process ending first (e.g. Definition 15).

Definition 13 (Operator Minimum) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Let also* o_1 *and* o_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$, *respectively. Assume that there exists some order criteria between them. Then the operation* $\left|\mathcal{M}_1 \textcircled{m} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{min}$ *that executes* $\mathcal{M}_1$ *and returns* $\min\{o_1, o_2\}$*:*

$$\mathcal{M}_{min} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Similarly we define the operator **Maximum**:

Definition 14 (Operator Maximum) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Let also* o_1 *and* o_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$, *respectively. Assume that there exists some order criteria between them. Then the operation* $\left|\mathcal{M}_1 \textcircled{M} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{max}$ *that executes* $\mathcal{M}_1$ *and returns* $\max\{o_1, o_2\}$*:*

$$\mathcal{M}_{max} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Comming back to the previews example, the **minimum operator** can be applied to obtain a more interesting behavior in the solver: When applying the acceptance criteria, suppose that we want to receive a configuration from other solver, to compare it with ours and select the one with the lowest cost. We can do that by applying the operator (m) to combine the *computation module* A with a *communication module* $C.M.$ (see Figure1.10):

$$\left[I \textcircled{\mapsto} \left[\circlearrowleft \left[\left[V \textcircled{\mapsto} S\right] \textcircled{\mapsto} \left[\!\left[A \textcircled{m} C.M.\right]\!\right]_p\right]\right]\right]$$

Notice that in this example, I can use the grouper $[\![.]\!]_p$ since the **minimum operator** supports parallelism.

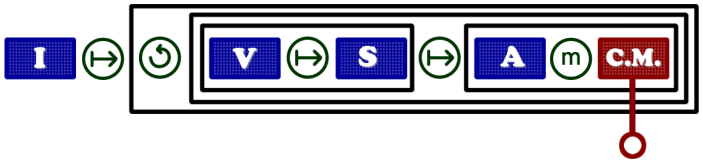


Figure 1.10: Using **minimum** operator

Definition 15 (Operator Race) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$ *and* $\mathcal{I}_1 \subset \mathcal{I}_2$. *Then the operation* $\left|\mathcal{M}_1 \textcircled{\downarrow} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{race}$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$, *and returns the output of the module ending first:*

$$\mathcal{M}_{race} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Sometimes nighborhood functions are slow depending on the configuration. In that case two neighborhood *computation modules* can be executed and take into account the output of the module ending first (see Figure1.11):

$$\left[I \textcircled{\mapsto} \left[\circlearrowleft \left[\left[\left[\!\left[V_1 \textcircled{\downarrow} V_2 \right]\!\right]_p \textcircled{\mapsto} S \right] \textcircled{\mapsto} \left[\!\left[A \textcircled{m} C.M. \right]\!\right]_p \right] \right] \right]$$

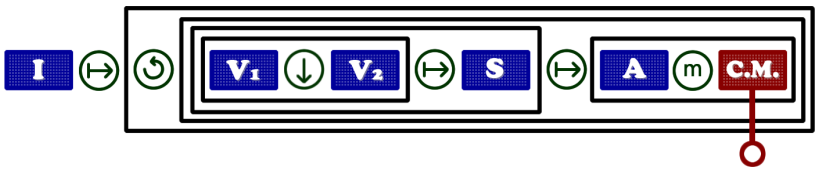


Figure 1.11: Using **race** operator

Some others operators can be useful when dealing with *sets*.

Definition 16 (Operator Union) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Let also* V_1 *and* V_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$, *respectively. Then the operation* $\left|\mathcal{M}_1 \textcircled{\cup} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_\cup$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$, *and returns* $V_1 \cup V_2$:

$$\mathcal{M}_\cup : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Similarly we define the operators **Intersection** and **Subtraction**:

Definition 17 (Operator Intersection) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Let also* V_1 *and* V_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$, *respectively. Then the operation* $\left|\mathcal{M}_1 \textcircled{\cap} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_\cap$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$, *and returns* $V_1 \cap V_2$:

$$\mathcal{M}_\cap : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Definition 18 (Operator Subtraction) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Let also* V_1 *and* V_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$, *respectively. Then the operation* $\left|\mathcal{M}_1 \textcircled{-} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_-$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$, *and returns* $V_1 - V_2$:

$$\mathcal{M}_- : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Now, I define the operators which allows to send information to other solvers. Two types of information can be sent: i) the output of the *computation module* and send its output, or ii) the *computation module* itself. . This utility is very useful in terms of sharing behaviors between solvers.

Definition 19 (Sending Data Operator) *Let* $\mathcal{M} : \mathcal{D} \rightarrow \mathcal{I}$ *be a module. Then the operation* $|(\!|\mathcal{M}|\!)^o|$ *defines the compound module* $\mathcal{M}_{sendD}$ *that executes the module* $\mathcal{M}$ *and sends its output outside:*

$$\mathcal{M}_{sendD} : \mathcal{D} \rightarrow \mathcal{I}$$

Similarly we define the operator **Send Module**:

Definition 20 (Sending Module Operator) *Let $\mathcal{M} : \mathcal{D} \to \mathcal{I}$ be a module. Then the operation $|(\!|\mathcal{M}|\!)^m|$ defines the compound module $\mathcal{M}_{sendM}$ that executes the module $\mathcal{M}$, then returns its output and sends the module itself outside:*

$$\mathcal{M}_{sendM} : \mathcal{D} \to \mathcal{I}$$

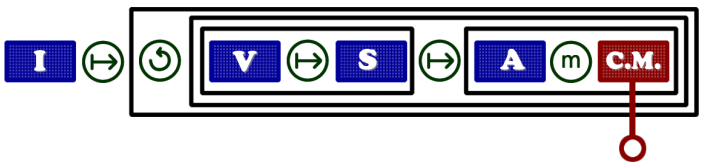
In the following example, I use one of the *compound modules* already presented in the previews examples, using a *communication module* to receive a configuration (see Figure 1.12a):

$$\left[I \textcircled{\mapsto} \left[\circlearrowleft \left[\left[V \textcircled{\mapsto} S \right] \textcircled{\mapsto} \left[\!\left[A \textcircled{m} C.M. \right]\!\right]_p \right] \right] \right]$$

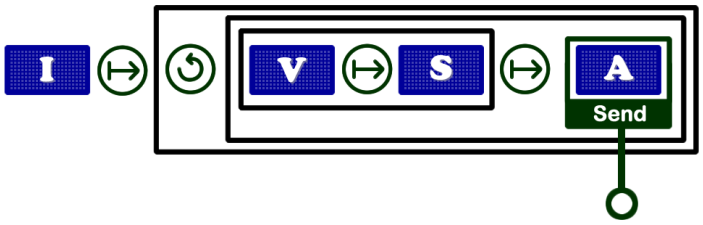
I also build another, as its complement: sending the accepted configuration to outside, using the **sending data operator** (see Figure1.12b):

$$\left[I \textcircled{\mapsto} \left[\circlearrowleft \left[\left[V \textcircled{\mapsto} S \right] \textcircled{\mapsto} (\!|A|\!)^o \right] \right] \right]$$

In the Section 1.5 I explain how to connect solvers to each other.



(a)



(b)

Figure 1.12: Sender and receiver behaviors

Once all desired abstract modules are linked together with operators, we obtain the *root compound module*, an important part of an *abstract solver*. To implement a concrete solver from an *abstract solver*, one must instantiate each abstract module with a concrete one respecting the required signature. From the same *abstract solver*, one can implement many different concrete solvers simply by instantiating abstract modules with different concrete modules.

An *abstract solver* is defined as follows: after declaring the **abstract solver**'s name, the first line defines the list of abstract *computation modules*, the second one the list of abstract *communication modules*, then the algorithm of the solver is defined as the solver's body (the root *compound module*), between **begin** and **end**.

An *abstract solver* can be declared through the simple regular expression:

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

where:

- *name* is the identifier of the *abstract solver*,
- L^m is the list of abstract *computation modules*,
- L^c is the list of abstract *communication modules*, and
- $\mathcal{M}$ is the root *compound module*.

For instance, Algorithm 1 illustrates the abstract solver corresponding to Figure 1.1b.

Algorithm 1: POSL pseudo-code for the *abstract solver* presented in Figure 1.1b

```
abstract solver as_01
computation : I, V, S, A
connection: C.M.
    begin
        I ⟼
        [↻ (Itr % K_1)
            [V ⟼ S ⟼ [C.M. (m) (|A|)^o]]
        ]
    end
```

1.4 Third stage: creating POSL solvers

With *computation* and *communication modules* composing an *abstract solver*, one can create solvers by instantiating *modules*. This is simply done by specifying that a given **solver** must **implements** a given *abstract solver*, followed by the list of *computation* then *communication modules*. These modules must match signatures required by the *abstract solver*.

> In the following example, I describe some concrete *computation modules* that can be used to implement the *abstract solver* declared in Algorithm 1:

$\boxed{Computation\ module-1}$ I_{rand} generates a random configuration s

$\boxed{Computation\ module-2}$ V_{1ch} defines the neighborhood $\mathcal{V}(s)$ changing only one element

$\boxed{Computation\ module-3}$ S_{best} selects the best configuration $s' \in \mathcal{V}(s)$ improving the current cost.

$\boxed{Computation\ module-4}$ A_{alw} evaluates an acceptance criterion for s'. We have chosen the classical module, selecting the configuration with the lowest global cost, *i.e.*, the one which is likely the closest to a solution.

I use also the following concrete *communication module*:

$\boxed{Communication\ module-1}$ CM_{last} returns the last configuration arrived, if at the time of its execution, there is more than one configuration waiting to be received.

These modules are used and explained in details in the Chapter 2 of this document. Algorithm 2 implements Algorithm 1 by instantiating its modules.

Algorithm 2: An instantiation of the *abstract solver* presented in Algorithm 1

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

1.5 Forth stage: connecting the solvers

We call *solver set* to the pool of (concrete) solvers that we plan to use in parallel to solve a problem. Once we have our solvers set, the last stage is to connect the solvers to each other. Up to this point, solvers are disconnected, but they are ready to establish the communication. POSL provides a platform to the user such that cooperative strategies can be easily defined.

In the following we present two important concepts necessary to formalize the *communication operators*.

Definition 21 (Communication Jack) *Let $\mathcal{S}$ be a solver. Then the operation $\mathcal{S} \cdot \mathcal{M}$ opens an outgoing connection from the solver $\mathcal{S}$, sending to the outside either a) the output of $\mathcal{M}$, if it is affected by a sending data operator as presented in Definition 19, or b) $\mathcal{M}$ itself, if it is affected by a sending module operator as presented in Definition 20.*

Definition 22 (Communication Outlet) *Let $\mathcal{S}$ be a solver. Then, the operation $\mathcal{S} \cdot \mathcal{CM}$ opens an ingoing connection to the solver $\mathcal{S}$, receiving from the outside either a) the output of some computation module, if $\mathcal{CM}$ is a data communication module, or b) a computation module, if $\mathcal{CM}$ is an object communication module.*

The communication is established by following the following rules guideline:

a) Each time a solver sends any kind of information by using a *sending* operator, it creates a *communication jack*.

b) Each time a solver defines a *communication module*, it creates a *communication outlet*.

c) Solvers can be connected to each other by linking *communication jacks* to *communication outlets*.

Following, we define the *connection operators* that POSL provides.

Definition 23 (Connection Operator One-to-One) *Let*

a) $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *be the list of communication jacks, and*

b) $\mathcal{O} = [\mathcal{Z}_0 \cdot \mathcal{CM}_0, \mathcal{Z}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{Z}_{N-1} \cdot \mathcal{CM}_{N-1}]$ *be the list of communication outlets*

Then the operation

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

connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ with the corresponding communication outlet $\mathcal{Z}_i \cdot \mathcal{CM}_i \in \mathcal{O}$, $\forall 0 \leq i \leq N-1$ (see Figure 1.13a).

Definition 24 (Connection Operator One-to-N) *Let*

a) $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *be the list of communication jacks, and*

b) $\mathcal{O} = [\mathcal{Z}_0 \cdot \mathcal{CM}_0, \mathcal{Z}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{Z}_{M-1} \cdot \mathcal{CM}_{M-1}]$ *be the list of communication outlets*

Then the operation

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

connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ with every communication outlet $\mathcal{Z}_j \cdot \mathcal{CM}_j \in \mathcal{O}$, $\forall 0 \leq i \leq N-1$ and $0 \leq j \leq M-1$ (see Figure 1.13b).

Definition 25 (Connection Operator Ring) *Let*

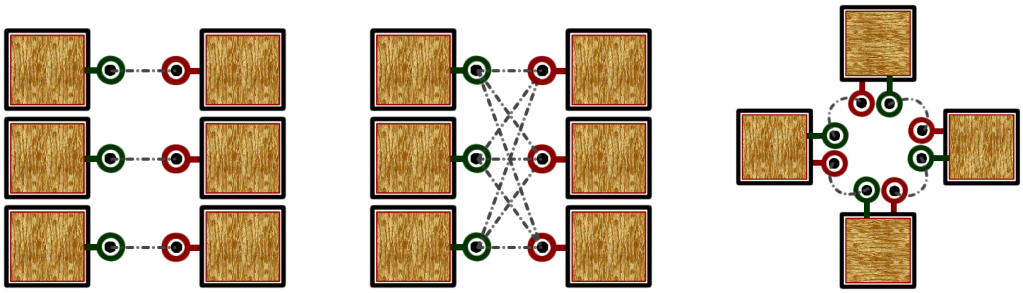
a) $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *be the list of communication jacks, and*

b) $\mathcal{O} = [\mathcal{S}_0 \cdot \mathcal{CM}_0, \mathcal{S}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{CM}_{N-1}]$ *be the list of communication outlets*

Then the operation

$$\mathcal{J} \circledast \mathcal{O}$$

connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ *with the corresponding communication outlet* $\mathcal{Z}_{(i+1)\%N} \cdot \mathcal{CM}_{(i+1)\%N} \in \mathcal{O}$, $\forall 0 \leq i \leq N-1$ *(see Figure 1.13c).*



(a) Communication 1 to 1 **(b)** Communication 1 to N **(c)** Cyclic communication

Figure 1.13: Graphic representation of communication operators

POSL also allows to declare non-communicating solvers to be executed in parallel, declaring only the list of solver names:

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

When we apply a connection operator $\circledast$ between a *communication jacks* list $\mathcal{J}$ and a *communication outlets* list $\mathcal{O}$, internally we are assigning an *abstract computation unit* (typically a thread) to each solver that we declare in each list. This assignment receives the name of *Solver Scheduling*. Before running the *solver set*, this *abstract unit of computation* is just an integer $\tau \in [0..N]$ identifying uniquely each of the solvers. When the *solver set* is launched, the solver with the identifier τ runs into the computation unit τ. This identifier assignation remains independent of the real availability of resources of computation. It just takes into account the user declaration. This means that, if the user declares 30 solvers (15 senders and 15 receivers) and the *solver set* is launched using 20 cores, only the first 20 solvers will be executed, and in consequence, there will be 10 solvers sending information to nowhere. Users should take this into account when declaring the *solver set*.

The connection process depends on the applied connection operator. In each case the goal is to assign, to the sending operator ($(\!|.|\!)^o$ or $(\!|.|\!)^m$) inside the *abstract solver*, the identifier of the solver (or solvers, depending on the connection operator) where the information will be

sent. Algorithm 3 presents the connection process.

Algorithm 3: Scheduling and connection main algorithm

```
input  : 𝒥 list of communication jacks,
         𝒪 list of communication outlets
1 while no available jacks or outlets do
2 |   S_jack ← GetNext(𝒥)
3 |   R_oulet ← GetNext(𝒪)
4 |   S ← GetSolverFromConnector(S_jack)
5 |   R ← GetSolverFromConnector(R_oulet)
6 |   Schedule(S)
7 |   R_id ← Schedule(R)
8 |   Connect(root(S),S_jack, R_id)
9 end
```

In Algorithm 3:

- `GetNext(...)` returns the next available solver-jack (or solver-outlet) in the list, depending on the connection operator, e.g., for the connection operator One-to-N, each *communication jack* in $\mathcal{J}$ must be connected with each *communication outlet* in $\mathcal{O}$.
- `GetSolverFromConnector(...)` returns the solver name given a connector declaration.
- `Schedule(...)` schedules a solver and returns its identifier.
- `Root(...)` returns the *root compound module* of a solver.
- `Connect(...)` searches the *computation module* S_{jack} recursively inside the *root compound module* of S and places the identifier R_{id} into its list of destination solvers.

Let us suppose that we have declared two solvers S and Z, both implementing the *abstract solver* in Algorithm 1, so they can be either sender or receiver. The following code connects them using the operator **1 to N**:

$$[S \cdot A] \overset{\rightsquigarrow}{\bigcirc} [Z \cdot C.M.]$$

If the operator **1 to N** is used with only with one solver in each list, the operation is equivalent to applying the operator **1 to 1**. However, to obtain a communication strategy like the one showed in Figure 1.13b, six solvers (three senders and three receivers) have to be declared to be able to apply the following operation:

$$[S_1 \cdot A, S_2 \cdot A, S_3 \cdot A] \overset{\rightsquigarrow}{\bigcirc} [Z_1 \cdot C.M., Z_2 \cdot C.M., Z_3 \cdot C.M.]$$

POSL provides a mechanism to make this easier, through *namespace expansions*.

1.5.1 Solver namespace expansion

One of the goals of POSL is to provide a way to declare sets of solvers to be executed in parallel fast and easily. For that reason, POSL provides two forms of namespace expansion, in order to create sets of solvers using already declared ones:

Solver name expansion - Uses an integer K to denote how many times the solver name S will appear in the declaration. $[\dots S_i \cdot \mathcal{M}(K), \dots]$ expands as $\left[\dots S_i \cdot \mathcal{M}, S_i^2 \cdot \mathcal{M}, \dots S_i^K \cdot \mathcal{M} \dots\right]$
and all new solvers $S_i^j, j \in [2..K]$ are created using the same solver declaration of solver S_i.

Connection declaration expansion - Uses an integer K to denote how many times the connection will be repeated in the declaration. Let a) $[\mathcal{S}_1 \cdot \mathcal{M}_1, \dots, \mathcal{S}_N \cdot \mathcal{M}_N]$ and b) $[\mathcal{R}_1 \cdot \mathcal{CM}_1, \dots, \mathcal{R}_M \cdot \mathcal{CM}_M]$ be the list of *communication jacks* and *communication outlets*, respectively, and c) $\textcircled{\text{op}}$ a connection operator. Then

$$[\mathcal{S}_1 \cdot \mathcal{M}_1, \dots, \mathcal{S}_N \cdot \mathcal{M}_N] \ \textcircled{op} \ [\mathcal{R}_1 \cdot \mathcal{CM}_1, \dots, \mathcal{R}_M \cdot \mathcal{CM}_M]\, K$$

expands as

$$\begin{gathered}
[\mathcal{S}_1 \cdot \mathcal{M}_1, \dots, \mathcal{S}_N \cdot \mathcal{M}_N] \ \textcircled{op} \ [\mathcal{R}_1 \cdot \mathcal{CM}_1, \dots, \mathcal{R}_N \cdot \mathcal{CM}_N] \\
\left[\mathcal{S}_1^2 \cdot \mathcal{M}_1, \dots, \mathcal{S}_N^2 \cdot \mathcal{M}_N\right] \ \textcircled{op} \ \left[\mathcal{R}_1^2 \cdot \mathcal{CM}_1, \dots, \mathcal{R}_N^2 \cdot \mathcal{CM}_N\right] \\
\dots \\
\left[\mathcal{S}_1^K \cdot \mathcal{M}_1, \dots, \mathcal{S}_N^K \cdot \mathcal{M}_N\right] \ \textcircled{op} \ \left[\mathcal{R}_1^K \cdot \mathcal{CM}_1, \dots, \mathcal{R}_N^K \cdot \mathcal{CM}_N\right]
\end{gathered}$$

and all new solvers $S_i^k, i \in [1..N]$ and $R_j^k, j \in [1..M]$, $k \in [2..K]$, are created using the same solver declaration of solvers S_i and R_j, respectively.

> Now, suppose that I have created solvers S and Z mentioned in the previews example. As a communication strategy, I want to connect them through the operator **1 to N**, using S as sender and Z as receiver. Then, using **namespace expansions**, I need to declare how many solvers I want to connect. Algorithm 4 shows the desired communication strategy. Notice in this example that the connection operation is affected also by the number 2 at the end of the line, as **connection declaration expansion**. In that sense, and supposing that 12 units of computation are available, a *solver set* working on parallel following the topology described in Figure 1.14 can be obtained.

Algorithm 4: A communication strategy

1 [S · *A* (3)] ⇝ [Z · *C.M.* (3)] 2 ;

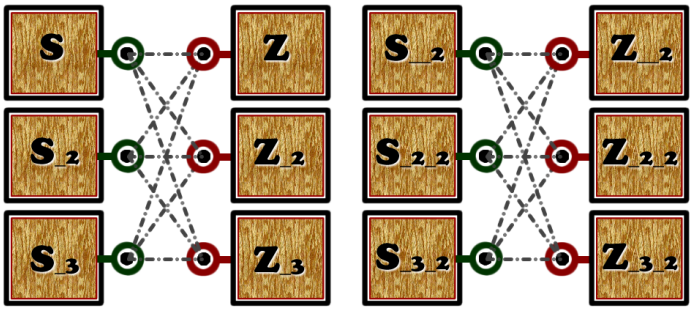


Figure 1.14: An example of connection strategy for 12 units of computation

1.6 Summarize

In this Chapter POSL have been formally presented, as a Parallel–Oriented Solver Language to build meta-heuristic-based solver to solve *Constraint Satisfaction Problems*. This language provides a set of *computation modules* useful to solve a wide range of problems. It is also possible to create new ones if needed, using a low-level framework in C++ programming language. POSL also provides a set of *communication modules*, essential features to share information between solvers.

One of the advantages of POSL is to create *abstract solvers* using a operator-based language, that remains independent of the used *computation* and *communication modules*. That is why it is possible to create many different solvers using the same solution strategy (the *abstract solver*) only instantiating it with different modules (*computation* and *communication modules*). It is also possible to create different communication strategies using the *connection operators* that POSL provides.

In the next Chapter, a detailed study of various communicating and non-communicating strategies, using some *Constraint Satisfaction Problems* as benchmarks. In this study, the efficacy of POSL to study easily and fast these strategies, is showed.

Part II

Study and evaluation of POSL

2

Experiments design and results

In this chapter I expose all details about the process of evaluation of POSL*, i.e., all experiments I perform. For each benchmark, I explain also used strategies in the evaluation process. I explain the used environments were we run the experiments (description of my desktop machine, Curiosiphi server). I describe all the experiments and I expose a complete analysis of the obtained result.*

In this chapter I illustrate and analyze the versatility of POSL studying different ways to solve constraint problems based on local search meta-heuristics. I have chosen the *Social Golfers Problem*, the *N-Queens Problem*, the *Costas Array Problem* and the *Golomb Ruler Problem* as benchmarks since they are two challenging yet differently structured problems. In this chapter I present formally each benchmark, I explain the structure of POSL's solvers that I generated for experiments and present a detailed analysis of obtained results.

The experiments[i] were performed on an Intel® Xeon™ E5-2680 v2, 10×4 cores, 2.80GHz. Showed results are the means of 30 runs for each setup, presented in columns labeled **T**, corresponding to the run-time in seconds, and **It.** corresponding to the number of iterations; and their respective standard deviations (**T(sd)** and **It.(sd)**). In some tables, the column labeled **% success** indicates the percentage of solvers finding a solution before reaching a time–out (5 minutes).

The experiments in this section are multi-walk runs using the same solver main structure (except differents w.r.t. communication operations). Parallel experiments use 40 cores for all problem instances. It is important to point out that POSL is not designed to obtain the best results in terms of performance, but to give the possibility of rapidly prototyping and studying different cooperative or non cooperative search strategies.

All benchmarks were coded using the POSL low-level framework in C++.

First results using POSL to solve constraint problems were published in [126] were we used POSL to solve the *Social Golfers Problem* and study some communication strategies. It was the first version of POSL, therefore it was able to solve only relatively easy instances. However, the efficacy of the communication was showed using this tool.

With the next and more optimized version of POSL, I decide to start to perform more detailed studies using the benchmark mentioned before and some others.

2.1 Solving the *Social Golfers Problem*

In this section I present the performed study using *Social Golfers Problem* (*SGP*) as a benchmark.

[i] POSL source code is available on GitHub:https://github.com/alejandro-reyesamaro/POSL

2.1.1 Problem definition

The *Social Golfers Problem* (*SGP*) consists in scheduling $g \times p$ golfers into g groups of p players every week for w weeks, such that two players play in the same group at most once. An instance of this problem can be represented by the triple $g - p - w$. This problem, and other closely related problems, arise in many practical applications such as encoding, encryption, and covering problems [121]. Its structure is very attractive, because it is very similar to other problems, like *Kirkman's Schoolgirl Problem* and the *Steiner Triple System*, so efficient modules to solve a broad range of problems can be built.

The cost function for this benchmark was implemented make an efficient use of the stored information about the cost of the previews configuration. Using integers to work with bit-flags, a table to store the information about the partners of each player in each week can be filled in $O(p^2 \cdot g \cdot w)$. So if a configuration has $n = (p \cdot g \cdot w)$ elements, this table can be filled in $O(p \cdot n)$. This table is filled from scratch only one time in the search process (I explain in the next section why). Then, every cost of a new configuration, is calculated based on this information and the performed changes between the new configuration an the stored one. This relative cost is calculated in $O(c \cdot g)$, where c is the number of performed changed in the new configuration with respect to the stored one.

2.1.2 Experiment design

Here, I give the abstract solver designed for this problem as well as concrete computation modules composing the different solvers I have tested:

a) Generation module:

I: Generates a random configuration s, respecting the structure of the problem, *i.e.*, the configuration is a set of w permutations of the vector $[1..n]$.

b) Neighborhood modules:

V_{Std}: Defines the neighborhood $\mathcal{V}(s)$ swapping players among groups.

V_{AS}: Defines the neighborhood $\mathcal{V}(s)$ swapping the most culprit player with other players from the same week. It is based on the *Adaptive Search* algorithm.

c) Selection modules:

S_{First}: Selects the first configuration $s' \in \mathcal{V}(s)$ improving the current cost.

S_{Best}: Selects the best configuration $s' \in \mathcal{V}(s)$ improving the current cost.

S_{Rand}: Selects a random configuration $s' \in \mathcal{V}(s)$.

d) Acceptance module:

A: Evaluates an acceptance criteria for s'. We have chosen the classical module selecting the configuration with the lowest global cost, *i.e.*, the one which is likely the closest to a solution.

A very first experiment was performed to select the best neighborhood function to solve the problem, comparing a basic solver using V_{Std}; a new solver using V_{AS}; and a combination of V_{Std} and V_{AS} by applying the operators (ρ), already introduced in the previous chapter. Algorithms 5, 6 and 7 present the *abstract solver* for each case, respectively.

Algorithm 5: Standard *abstract solver* for *SGP*

```
1 abstract solver as_union                          // Itr → number of iterations
2 computation : I, V, S, A
3 begin
4     [↺ (Itr < K1)
5         I (↦) [↺ (Itr % K2) [V (↦) S (↦) A] ]
6     ]
7 end
```

Algorithm 6: *Abstract solver* combining neighborhood functions using operator *RHO*

```
1 abstract solver as_union                          // Itr → number of iterations
2 computation : I, V1, V2, S, A
3 begin
4     [↺ (Itr < K1)
5         I (↦) [↺ (Itr % K2) [[V1 (ρ) V2] (↦) S (↦) A] ]
6     ]
7 end
```

Algorithm 7: *Abstract solver* combining neighborhood functions using operator *Union*

```
1 abstract solver as_union                          // Itr → number of iterations
2 computation : I, V1, V2, S, A
3 begin
4     [↺ (Itr < K1)
5         I (↦) [↺ (Itr % K2) [[V1 (∪) V2] (↦) S (↦) A] ]
6     ]
7 end
```

Solvers mentioned above were too slow to solve instances of the problem with more than 3 weeks, so another solver implementing the *abstract solver* described in Algorithm 8 have been

created, using V_{AS} and combining S_{First} and S_{Rand}: it tries a number of times to improve the cost, and if it is not possible, it picks a random neighbor for the next iteration. We also compared the S_{First} and S_{Best} selection modules.

Algorithm 8: *Abstract solver* for *SGP* to scape from local minima

```
1 abstract solver as_eager                                  // Itr → number of iterations
2 computation : I, V, S1, S2, A
3 begin
4     [↻ (Itr < K1)
5         I ↦ [↻ (Itr % K2) [V ↦ [S1 ?_{Sci<K3} S2] ↦ A] ]
6     ]
7 end
```

After that, the best solver to be communicating solvers to compare their performance with the non communicating strategies was chosen. The shared information is the current configuration. Algorithms 9 and 10 show that the communication is performed while applying the acceptance criterion of the new configuration for the next iteration. Here, solvers receive a configuration from an outer solver, and match it with their current configuration. Then solvers select the configuration with the lowest global cost. We design different communication strategies. Either we execute a full connected solvers set, or a tuned combination of connected and unconnected solvers. Between connected solvers, we applied two different connections operations: connecting each sender solver with one receiver solver (*1 to 1*), or connecting each sender solver with all receiver solvers (*1 to N*).

Algorithm 9: Communicating *abstract solver* for *SGP* (sender)

```
1 abstract solver as_eager_sender                           // Itr → number of iterations
2 computation : I, V, S1, S2, A                  // Sci → number of iterations with the same cost
3 begin
4     [↻ (Itr < K1)
5         I ↦ [↻ (Itr % K2) [V ↦ [S1 ?_{Sci<K3} S2] ↦ (|A|)^o] ]
6     ]
7 end
```

In all Algorithms ins this section, three parameter can be found: 1. K_1: the maximum number of *restarts*, 2. K_2: the maximum number of iterations in each *restart*, and K_3: the maximum number of iterations with the same current cost. 3.

After the selection of the proper modules to study the different communication strategies, I proceeded to tune these parameter. Only a few runs were necessaries to conclude that the mechanism of using the *computation module* S_{rand} to scape from local minima was enough. For that reason, since the solver never perform restarts, the parameter K_1 was irrelevant. So the reader can assume $K_1 = 1$ for every experiment.

Algorithm 10: Communicating *abstract solver* for *SGP* (receiver)

```
1  abstract solver as_eager_receiver                    // Itr → number of iterations
2  computation : I, V, S1, S2, A          // Sci → number of iterations with the same cost
3  communication : C.M.
4  begin
5      [↻ (Itr < K1)
6          I (↦)
7          [↻ (Itr % K2)
8              V (↦) [S1 (?)Sci<K3 S2] (↦) [A (m) C.M.]
9          ]
10     ]
11 end
```

With the certainty of the solvers do not performs restarts during the search process, I select the same value for $K_2 = 5000$ in order to be able to use the same *abstract solver* for all instances.

Finally, in the tuning process of K_3, I notice only slightly differences between using the values 5, 10, and 15. So I decided to use $K_3 = 5$.

2.1.3 Analysis of results

Table 2.1 showed results of launching *solver set*s to solve each instance of the problem sequentially. Not surprisingly, the means of sequential runtimes and iterations (Table 2.1) are bigger than those means of parallel runs, with or without communication (all other tables).

Instance	T	T(sd)	It.	It.(sd)	% success
5–3–7	8.31	7.64	17,347	15,673	100.00
8–4–7	16.92	15.15	7,829	7,019	100.00
9–4–8	79.60	64.07	20,779	16,537	94.28
11–7–5	3.37	2.16	664	380	100.00

Table 2.1: *Social Golfers*: a single sequential solver

In a first stage of the experiments I use the operator-based language provided by POSL to build and test many different non communicating strategies. The goal is to select the best concrete modules to run tests performing communication. In particular, I have tested two kind of computation modules: the one computing the neighborhood of a given configuration and the one choosing the current configuration for the next solver iteration.

I focused on choosing the right neighborhood function. In the case of the *Social Golfers Problem*, this experiment was launched using a basic abstract solver showed in Algorithm 5.

Abstract solvers	T	T(sd)	It.	It.(sd)
Adaptive Search (AS)	**1.06**	0.79	352	268
Std (ρ) AS	41.53	26.00	147	72
Std $(\cup)$ AS	59.65	55.01	198	110
Standard (Std)	87.90	41.96	146	58

Table 2.2: *Social Golfers*: Instance 10–10–3 in parallel

Instance	O.M. Best Improvement				O.M. First Improvement			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	4.99	4.43	4,421	3,938	**1.32**	0.68	1322	676
8–4–7	5.10	1.77	954	334	**1.82**	0.84	445	191
9–4–8	12.37	5.40	1,342	591	**6.43**	4.60	873	591
11–7–5	5.19	1.67	351	114	**2.22**	0.69	273	58

Table 2.3: *Social Golfers*: comparing selection functions

Solvers implemented from this abstract solver was too slow to solve instances beyond three weeks: they were very often trapped into local minima. This is the reason why we perform this first experiment with the instance 10–10–3 whereas next experiments scale above 3 weeks. This was not a problem though, since the goal of this first experiment was only to find the right concrete neighborhood module.

Results in Table 2.2 are not surprising. The neighborhood neighborhood module V_{AS} is based on the *Adaptive Search* algorithm, which has shown very good results [1]. It selects the most culprit variable (i.e., a player), that is, the variable to most responsible for constraints violation. Then, it permutes this variable value with the value of each other variable, in all groups and all weeks. Each permutation gives a neighbor of the current configuration. V_{Std} uses no additional information, so it performs every possible swap between two players in different groups, every week. It means that this neighborhood is $g \times p$ times bigger than the previous one, with g the number of groups and p the number of players per group. It allows for more organized search because the set of neighbors is pseudo-deterministic, i.e., the construction criteria is always the same but the order of the configuration is random. On the other hand, *Adaptive Search* neighborhood function takes random decisions more frequently, and the order of the configurations is random as well. We also tested abstract solvers with different combinations of these modules, using the (ρ) and the $(\cup)$ operators. The (ρ) operator executes its first or second parameter depending on a given probability ρ, and the $(\cup)$ operator returns the union of its parameters output. All these combinations spent more time searching the best configuration among the neighborhood, although with a lower number of iterations than V_{AS}. The V_{AS} neighborhood function being clearly faster, we have chosen it for our experiments, even if it shown a more spread standard deviation: 0.75 for AS versus 0.62 for Std, considering the ratio $\frac{T(sd)}{T}$.

With the selected neighborhood function, I focused on choosing the best *selection* function.

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	1.19	0.64	1,156	608	1.11	0.49	1,067	484
8–4–7	**1.30**	0.72	**317**	161	1.46	0.57	347	128
9–4–8	4.38	2.72	**597**	347	5.51	3.06	736	389
11–7–5	1.76	0.41	214	44	**1.62**	0.34	**202**	30

Table 2.4: *Social Golfers*: test with 100% of communication

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	1.04	0.45	1,019	456	1.04	0.53	1,031	530
8–4–7	1.40	0.57	337	122	1.43	0.76	353	167
9–4–8	4.64	2.17	637	279	5.75	3.06	776	389
11–7–5	1.81	0.40	220	33	1.82	0.39	222	39

Table 2.5: *Social Golfers*: test with 50 % of communication

I compared two different concrete modules used within the abstract solver in Algorithm 8, which combines selection modules (S_{First} or S_{Best}) with S_{Rand}, to avoid being trapped into local minima: it tries to improve the cost in a limited number of iterations. If it is not possible, it selects a random neighbor for the next iteration. The first module was S_{Best} that selects the best configuration inside the neighborhood. It not only spent more time searching a better configuration, but also is more sensitive to become trapped into local minima. The second module was S_{First} which selects the first configuration inside the neighborhood improving the current cost. Using this module, solvers favor exploration over intensification and of course spend clearly less time computing the neighborhood. Table 2.3 presents results of this experiment, showing that an exploration-oriented strategy is better for the *SGP*. If we compare results of Tables 2.1 and 2.3 with respect to the standard deviation, we can some gains in robustness with parallelism. The spread in the running times and iterations for the instance 9–4–8 (the hardest one) is 10% lower (0.80 sequentially versus 0.71 in parallel), and for the others, it is around 40% lower (0.91, 0.89 and 0.64 sequentially versus 0.51, 0.45 and 0.31 in parallel, for 5–3–7, 8–4–7 and 11–7–5 respectively, with the same ratio $\frac{T(sd)}{T}$).

Then we ran experiments to study POSL's behavior solving target problems in communicating scenarios. Some compositions of solvers set were taken into account: i. the structure of the

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	**0.90**	0.51	**881**	492	1.19	0.67	1,170	655
8–4–7	1.39	0.43	341	94	1.46	0.43	352	96
9–4–8	**4.33**	1.92	599	248	4.53	2.01	625	251
11–7–5	1.99	0.54	242	51	1.63	0.35	224	28

Table 2.6: *Social Golfers*: test with 25% of communication

communication (with/without communication or a mix), and ii. the used communication operator.

Each time a POSL meta-solver is launched, many independent search solvers are executed. We call "good" configuration a configuration with the lowest cost within the current configuration neighborhood and with a cost strictly lesser than the current one. Once a good configuration is found in a sender solver, it is transmitted to the receiver one. At this moment, if the information is accepted, there are some solvers searching in the same subset of the search space, and the search process becomes more exploitation–oriented. This can be problematic if this process makes solvers converging too often towards local minima. In that case, we waste more than one solver trapped into a local minima: we waste all solvers that have been attracted to this part of the search space because of communications. I avoid this phenomenon through a simple (but effective) play: if a solver is not able to find a better configuration inside the neighborhood (executing S_{First}), it selects a random one at the next iteration (executing S_{Rand}). This strategy, using communication between solvers, produces some gain in terms of runtime (Table 2.3 with respect to Tables 2.4, 2.5 and 2.6. The percentage of the receiver solvers that were able to find the solution before the others did, was significant (see Appendix A). That shows that the communication played an important role during the search, despite inter–process communication's overheads (reception, information interpretation, making decisions, etc). Having many solvers searching in different places of the search space, the probability that one of them reaches a promising place is higher. Then, when a solver finds a good configuration, it can be communicated, and receiving the help of one or more solvers in order to find the solution. For this problem we have reduced the spread in the running times and iterations of the results for the two last instances (9–4–8 and 11–7–5) applying the communication strategy (0.71 without communication versus 0.44 with communication, for 9–4–8, and 0.31 without communication versus 0.20 with communication for 11–7–5).

Other two strategies were analyzed in the resolution of this problem, with no success, both based on the sub-division of the work by weeks, i.e., solvers trying to improve a configuration only working with one or some weeks. To this end two strategies were designed:

A **Circular strategy:** In this strategy, K solvers try to improve a configuration during a during a number of iteration, only working on one week. When no improvement is obtained, the current configuration is communicated to the next solver (circularly), which tries to do the same working on the next week (see Figure 2.1a).

This strategy does not show better results than previews strategies. The reason is because, although the communication in POSL is asynchronous, most of the times solvers were trapped waiting for a configuration coming from its neighbor solver.

B **Dichotomy strategy:** In this strategy solvers are divided by levels. Solvers in level 1, only work on one week, solvers on level 2, only work on 2 consecutive weeks, and so on, until the solver that works on all (except the first one) weeks. Solvers in level 1 improve a configuration during some number of iteration, then this configuration is sent to the corresponding solver. A solver in level 2 do the same, but working on weeks k to $k+1$. It means that it receives configurations from the solver working on week k and from the solver working on week $k+1$, and sends its configuration to the corresponding solver working on weeks k to $k+3$; and so on. The solver in the level works on all (except the first one) weeks and receive configuration from the solver working on weeks 2 to $w/2$ and from the solver working on weeks $w/2+1$ to w (see Figure2.1b). We tested this strategy with all possible levels.

The goal of this strategy was testing if focused searches rapidly communicated can help at the beginning of the search. However, The failure of this strategy is in the fact that most of the time the sent information arrives to late to the receiver solver.

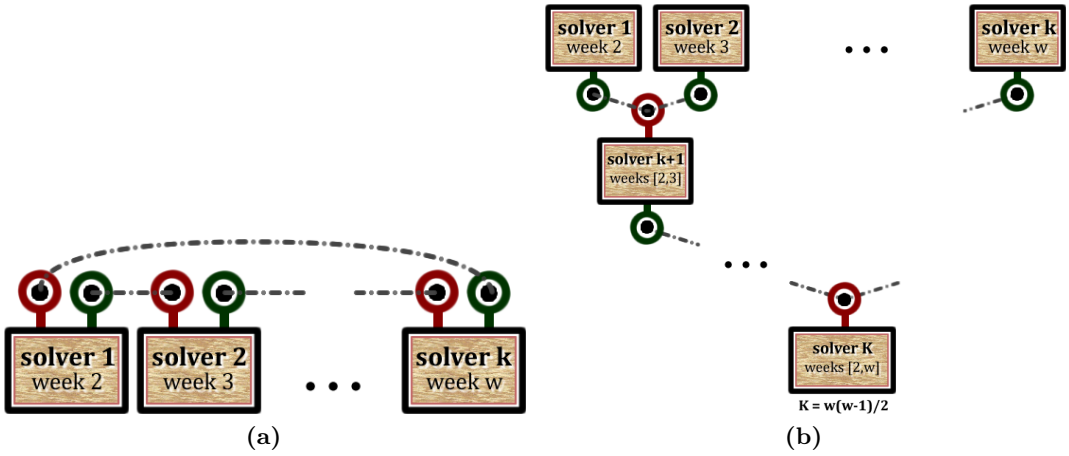


Figure 2.1: Unsuccessful communication strategies to solve *SGP*

2.2 Solving the *Costas Array Problem*

In this section I present the performed study using *Costas Array Problem* (*CAP*) as a benchmark.

2.2.1 Problem definition

The *Costas Array Problem* (*CAP*) consists in finding a *costas array*, which is an $n \times n$ grid containing n marks such that there is exactly one mark per row and per column and the $n(n-1)/2$ vectors joining each couple of marks are all different. This is a very complex problem that finds useful application in some fields like sonar and radar engineering. It also presents an interesting characteristic: although the search space grows factorially, from order 17 the number of solutions drastically decreases [122].

The cost function for this benchmark was implemented in C++ based on the current implementation of *Adaptive Search*[ii].

2.2.2 Experiment design

To handle this problem, I have reused some modules used for the *Social Golfers Problem* and *N-Queens Problem*: the *Neighborhood computation module* used for *N-Queens*, and the *Selection* and *Acceptance computation modules* used for both. The new modules are:

a) Generation module:

I: Generates a random configuration s, as a permutation of the vector $[1..n]$.

b) Neighborhood module:

V_{AS}: Defines the neighborhood $V(s)$ swapping the variable which contributes the most to the cost with other.

First attempts to solve this problems were using the same strategy (*abstract solver*) used to solve the *Social Golfers Problem*, without success: POSL was not able to solve instances larger than $n = 8$ in a reasonable amount of time (seconds). After many unsuccessful attempts to find the rights parameter of *maximum number of restarts*, *maximum number of iterations*, and *maximum number of iterations with the same cost*, I decided to implement the mechanism used by Daniel Díaz in the current implementation of *Adaptive Search* to escape from local minima: I have added a *Reset* module (R).

The basic solver I use to solve this problem is presented in Algorithm 11, and I take it as a base to build all the different communication strategies. Basically, it is a classical local search iteration, where instead of performing restarts, it performs resets. After a deep analysis of this implementation and results of some runs, I decided to use $K_1 = 24000$ (maximum

[ii] It is based on the code from Daniel Díaz available at https://sourceforge.net/projects/adaptivesearch/

number of iterations) big enough to solve the chosen instance $n = 17$; and $K_2 = 3$ (the number of iteration before performing the next *reset*).

Algorithm 11: Reset-based *abstract solver* for *CAP*

```
1 abstract solver as_hard                          // Itr → number of iterations
2 computation : I, R, V, S, A
3 begin
4     I (↦)
5     [↺ (Itr < K1)
6         R (↦) [↺ (Itr % K2) [V (↦) S (↦) A] ]
7     ]
8 end
```

The *abstract solver* for the sender solver is presented in Algorithm 12. Like for the *Social Golfers Problem*, we design different communication strategies combining different percentages of communicating solvers and our two communication operators (*1 to 1* and *1 to N*). However for this problem, we study the behavior of the communication performed at two different moments: while applying the acceptance criteria (Algorithm 13), and while performing a

reset (Algorithms 13, 14 and 15).

Algorithm 12: Reset-based *abstract solver* for *CAP* (sender)

```
1  abstract solver as_hard_sender                    // Itr → number of iterations
2  computation : I, R, V, S, A
3  begin
4      I (↦)
5      [↺ (Itr < K1)
6          R (↦) [↺ (Itr % K2) [V (↦) S (↦) (|A|)^o] ]
7      ]
8  end
```

Algorithm 13: Reset-based *abstract solver* for *CAP* (receiver, variant A)

```
1  abstract solver as_hard_receiver_a                // Itr → number of iterations
2  computation : I, R, V, S, A
3  communication : C.M.
4  begin
5      I (↦)
6      [↺ (Itr < K1)
7          R (↦) [↺ (Itr % K2) [V (↦) S (↦) [A (m) C.M.]] ]
8      ]
9  end
```

Algorithm 14: Reset-based *abstract solver* for *CAP* (receiver, variant B)

```
1  abstract solver as_hard_receiver_b                // Itr → number of iterations
2  computation : I, R, V, S, A          // Sci → number of iterations with the same cost
3  communication : C.M.
4  begin
5      I (↦)
6      [↺ (Itr < K1)
7          [R (?)_{Sci<K3} [R (m) C.M.]] (↦) [↺ (Itr % K2) [V (↦) S (↦) A] ]
8      ]
9  end
```

Algorithm 15: Reset-based *abstract solver* for *CAP* (receiver, variant C)

```
1  abstract solver as_hard_receiver_c                // Itr → number of iterations
2  computation : I, R, V, S, A
3  communication : C.M.
4  begin
5      I (↦)
6      [↺ (Itr < K1)
7          [R (m) C.M.] (↦) [↺ (Itr % K2) [V (↦) S (↦) A] ]
8      ]
9  end
```

2.2.3 Analysis of results

We present in Table 2.7 results of launching *solver sets* to solve each instance of *Costas Array Problem* sequentially. Runtimes and iteration means showed in this Table are bigger than those presented in Table 2.8, confirming once again the success of the parallel approach.

STRATEGY	T	T(ds)	It.	It.(sd)	% success
Sequential (1 core)	2.12	0.87	44,453	18,113	42.00
Parallel (40 cores)	0.73	0.46	9,556	6,439	100.00

Table 2.7: *Costas Array* 17: no communication

I chose directly the neighborhood module (V_{AS}), the selection module (S_{First}) and the acceptance module A, to create the solvers. I ran experiments to study parallel communicating strategies taken into account the structure of the communication, and the communication operator used, but in this problem, I perform the communication at two different times: at the time of applying the acceptance criteria, and at the time of performing the *reset*.

Table 2.8 shows that the *abstract solver A* (receiving the configuration at the time of applying the acceptance criteria) is more effective. The reason is that the others, interfere with the proper performance of the *reset*, that is a very important step in the algorithm. This step can be performed on three different ways:

a) Trying to shift left/right all sub-vectors starting or ending by the variable which contributes the most to the cost, and selecting the configuration with the lowest cost.

b) Trying to add a constant (circularly) to each element in the configuration.

c) Trying to shift left from the beginning to some culprit variable (i.e., a variable contributing to the cost).

Then, one of these 3 generated configuration has the sabe probability of being selected, to be the result of the *reset* step. In that sense, some different *resets* can be performed for the same configuration. Here is when the communication play its important role: receiver and

	100% COMM				50% COMM			
STRATEGY	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
Str A: 1 to 1	**0.41**	0.30	4,973	3,763	0.55	0.43	8,179	7,479
Str A: 1 to N	0.43	0.31	5,697	4,557	0.57	0.46	8,420	7,564
Str B: 1 to 1	0.48	0.41	6,546	5,562	0.51	0.49	8,004	7,998
Str B: 1 to N	0.45	0.46	5,701	6,295	0.48	0.51	7,245	8,379
Str C: 1 to 1	0.48	0.43	6,954	6,706	0.58	0.43	8,329	6,593
Str C: 1 to N	0.49	0.38	6,457	5,875	0.58	0.50	8,077	8,319

Table 2.8: *Costas Array* 17: with communication

sender solvers apply different *reset* in the same configuration, and results showed the efficacy of this communication strategy.

Table 2.8 shows also high values of standard deviation. This is not surprising, due to the highly random nature of the neighborhood function and the selecting criterion, as well as the execution of many resets during the search process.

2.3 Solving the *Golomb Ruler Problem*

In this section I present the performed study using *Golomb Ruler Problem* (*GRP*) as a benchmark.

2.3.1 Problem definition

The *Golomb Ruler Problem* (*GRP*) problem consists in finding an ordered vector of n distinct non-negative integers, called *marks*, $m_1 < \cdots < m_n$, such that all differences $m_i - m_j$ $(i > j)$ are all different. An instance of this problem is the pair (o, l) where o is the order of the problem, (i.e., the number of *marks*) and l is the length of the ruler (i.e., the last *mark*). We assume that the first *mark* is always 0. This problem has been applied to radio astronomy, x-ray crystallography, circuit layout and geographical mapping [125]. When I apply POSL to solve an instance of this problem sequentially, I can notice that it performs many *restarts* before finding a solution. For that reason I have chosen this problem to study a new communication strategy.

The cost function is implemented based on the storage of a counter for each measure present in the rule (configuration). I also store all distances where a variable is participating. This information is usefull to compute the more culprit variable (the variable that interfiers less in the representd measures), in case of the user wants to apply algorithms like *Adaptive Search*. This cost is calculated in $O\left(o^2 + l\right)$.

2.3.2 Experiment design

I use *Golomb Ruler Problem* instances to study a different communication strategy. This time I communicate the current configuration, to avoid its neighborhood, i.e., a *tabu* configuration. I have reused some modules used in the resolution of *Social Golfers* and *Costas Array*

problems to design the solvers: the *Selection* and *Acceptance* modules. The new modules are:

a) Generation module:

I: Generates a random configuration s, respecting the structure of the problem, i.e., the configuration is an ordered vector of integers. This module takes into account a set of *tabu* configurations arrived from the same solver, and also via solver-communication through a *communication module* $C.M.$ that receives a set of configurations. This module constructs the new configuration far enough from the *tabu* configurations.

b) Neighborhood module:

V: Defines the neighborhood $\mathcal{V}(s)$ by changing one value while keeping the order, i.e., replacing the value s_i by all possible values $s'_i \in D_i$ that satisfy $s_{i-1} < s'_i < s_{i+1}$.

I also add a module to insert a configuration into a *tabu* list inside the solver. In Algorithm 16 the *abstract solver* used to send information (sender *abstract solver*) is presented. When the module T is executed, the solver is unable to find a better configuration around the current one, so it is assumed to be a local minimum, and it is sent to the receiver solver. Algorithm 17 presents an *abstract solver* used to receive information (receiver *abstract solver*). Based on the connection operator used in the communication strategy, this solver might receives one or many configurations. These configurations are the input of the generation module (I), and this module inserts all received configurations into a *tabu* list, and then generates a new first configuration, far from all configurations in the *tabu* list.

Algorithm 16: *Abstract solver* for *GRP* (sender)

```
1 abstract solver as_golomb_sender                    // Itr → number of iterations
2 computation : I, V, S, A, T
3 begin
4     [↺ (Itr < K1)
5         I ↦ [↺ (Itr % K2) [V ↦ S ↦ A] ] ↦ (|T|)^o
6     ]
7 end
```

Algorithm 17: *Abstract solver* for *GRP* (receiver)

```
1 abstract solver as_golomb_receiver                  // Itr → number of iterations
2 computation : I, V, S, A, T
3 connection : C.M.
4 begin
5     [↺ (Itr < K1)
6         [C.M. ↦ I] ↦ [↺ (Itr % K2) [V ↦ S ↦ A] ] ↦ (|T|)^o
7     ]
8 end
```

In this communcation strategy there are some parameters to be tuned. The first ones are: 1. K_1, the number of restarts, and 2. K_2, the number of iterations by restart. Both are instance dependent, so, after many experimental runs, I choose them as follows:

- *Golomb Ruler* 8–34: $K_1 = 300$ and $K_2 = 200$
- *Golomb Ruler* 10–55: $K_1 = 1000$ and $K_2 = 1500$
- *Golomb Ruler* 11-72: $K_1 = 1000$ and $K_2 = 3000$

The other parameters are related to the behavior of the *tabu list*:

The idea of this strategy (*abstract solver*) follows the following steps:

Step 1

The *computation module* generates an initial configuration tacking into account a set of configurations into a *tabu list*. The configuration arriving to this *tabu list* come from the same solver (Step 3) or from outside (other solvers) depending on the strategy (non-communicating or communicating).

This module applies some other modules provided by POSL to solve the *Sub-Sum Problem* in order to generates *valid* configurations for *Golomb Ruler Problem*. A valid configuration s for *Golomb Ruler Problem* is a configuration that fulfills the following constraints:

- $s = (a_1, \ldots, a_o)$ where $a_i < a_j, \forall i < j$, and
- all $d_i = a_{i+1} - a_i$ are all different, for all $d_i, i \in [1...o-1]$

The *Sub-sum Problem* is defined as follows: Given a set E of integers, with $|E| = N$, finding a sub set e of n elements that sums exactly z. In that sense, a solution $S_{sub-sum} = \{s_1, \ldots, s_{o-1}\}$ of the *Sub-sum problem* with $E = \left\{1, \ldots, l - \frac{(o-2)(o-1)}{2}\right\}$, $n = o - 1$ and $z = l$, can be traduced to a *valid* configuration C_{grp} for *Golomb Ruler Problem* as follows:

$$C_{grp} = \{c_1, c_1 + s_1, \ldots, c_{o-1} + s_{o-1}\}$$

where $c_1 = 0$.

In the selection module applied inside the module I the selection step of the search process selects a configuration from the neighborhood *far* from the *tabu* configurations, with respect to certain vectorial norm and an epsilon. In other words, a configuration C is selected if and only if:

a) the cost of the configuration C is lower than the current cost, and

b) $||C - C_t||_p > \varepsilon$, for all *tabu* configuration C_t

where p and ε are parameters.

I experimented with 3 different values for p. Each value defines a different type of norm of a vector $x = \{x_1, \ldots, x_n\}$:

- $p = 1$: $||x||_1 = \sum_{i=0}^{n} |x_i|$
- $p = 2$: $||x||_2 = \sqrt{\sum_{i=0}^{n} |x_i|^2}$
- $p = \infty$: $||x||_\infty = \max(x)$

After many experimental runs with these values I choose $p = \infty$ and $\varepsilon = 4$ for the communication strategy study. I also made experiments trying to find the right size for the *tabu* list and the conclusion was that the right sizes were 15 for non-communicating strategies and 40 for communicating strategies, taking into account that in the latter, I work with 20 receivers solvers.

Step 2

After generating the first configuration, the next step is to apply a local search to improve it. In this step I use the neighborhood *computation module* V, that creates neighborhood $\mathcal{V}(s)$ by changing one value while keeping the order in the configuration, and the other modules (selection and acceptance). The local search is executed a number K_2 of times, or until a solution is obtained.

Step 3

If no improvement is reached, the current configuration is classified as a *potential local minimum* and inserted into the *tabu* list. Then, the process returns to the Step 1.

2.3.3 Analysis of results

The benefit of the parallel approach is also proved for the *Golomb Ruler Problem* (see Table 2.9 with respect to 2.10, 2.11, 2.12 and 2.13). But the main goal of choosing this benchmark was to study a different communication strategy, since for solving this problem, POSL needs to perform some restarts. In this communication strategy, solvers do not communicate the current configuration to have more solvers searching in its neighborhood, but a configuration that we assume is a local minimum to be avoided. We consider that the current configuration is a local minimum since the solver (after a given number of iteration) is not able to find a better configuration in its neighborhood, so it will communicate this configuration just before performing the restart.

The first experiment compares the runs of non communitaing solvers not using a *tabu* list with non communicating solvers using a *tabu* list. The results in Tables 2.10 and 2.11 demonstrate that using a *tabu* list can help the search process. Without communication, the improvement is not substantial (8% for 8–34, 7% for 10–55 and 5% for 11–72). The reason is because only one configuration is inserted in the *tabu* list after each restart. When we use *1 to 1* communication, after the restart k, the receiving solver has twice the number of configurations in the *tabu* list (one *tabu* configuration from itself and the received one after each restart). Table 2.12 shows that this strategy is not sufficient for some instances, but when we use *1 to N* communication, the number of *tabu* configurations after the restart k, in the receiving solver is considerably higher, e.g., after the restart k a receiving solver has $k(N+1)$ configurations in his *tabu* list (one *tabu* configuration from itself and N received from the other solvers, each restart). Hence, these solvers can generate configurations far enough from many potentially local minima. This phenomenon is more visible when the problem order increases. Table 2.13 shows that the improvement for the higher case (11-72) is about 32% with respect to non communicating solvers without using a *tabu* list (Table 2.10), and about 29% with respect to non communicating solvers using a *tabu* list (Table 2.11).

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)	% success
8–34	0.79	0.66	13,306	11,154	66	55.74	100.00
8–34 (t)	0.66	0.63	10,745	10,259	53	51.35	100.00
10–55	66.44	49.56	451,419	336,858	301	224.56	80.00
10–55 (t)	67.89	50.02	446,913	328,912	297	219.30	88.00
11–72	160.34	96.11	431,623	272,910	143	90.91	26.67
11–72 (t)	117.49	85.62	382,617	275,747	127	91.85	30.00

Table 2.9: *Golomb Ruler*: a single sequential solver

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)
8–34	0.47	34.82	436	330.10	2	1.63
10–55	5.31	38.63	22,577	16,488	15	11.00
11–72	89.76	55.85	164,763	102,931	54	34.32

Table 2.10: *Golomb Ruler*: parallel, without tabu list.

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)
8–34	0.43	0.37	349	334	1	1.64
10–55	4.92	4.68	20,504	19,742	13	13.07
11–72	85.02	67.22	155,251	121,928	51	40.64

Table 2.11: *Golomb Ruler*: parallel, with tabu list.

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)
8–34	0.44	0.31	309	233	1	1.23
10–55	3.90	3.22	15,437	12,788	10	8.52
11–72	85.43	52.60	156,211	97,329	52	32.43

Table 2.12: *Golomb Ruler*: parallel, communication 1 to 1.

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)
8–34	0.43	0.29	283	225	1	1.03
10–55	3.16	2.82	12,605	11,405	8	7.61
11–72	60.35	43.95	110,311	81,295	36	27.06

Table 2.13: *Golomb Ruler*: parallel, communication 1 to n.

2.4 Summarizing

In this Chapter I have chosen various *Constraint Satisfaction Problems* as benchmarks to 1. evaluate the POSL behavior solving these kind of problems, and 2. to study different solution strategies, specially communication strategies.To this end, I have chosen benchmarks with different characteristics, to help me having a wide view of the POSL behavior.

In the solution process of *Social Golfers Problem*, it was showed the success of an exploitation-oriented communication strategy, in which the current configuration is communicated to ask other solvers for help, concentrating the effort in a more promising area. I was able also to study some other unsuccessful strategies, to show that strategies based on the sub-division of the effort by weeks, is not a good idea.

N-Queens Problem....

The *Costas Array Problem* is a very complicated constrained problem, and very sensitive to the methods to solve it. For that reason I used some ideas from already existent algorithms. However, thanks to some studies of different communication strategies, based on the communication of the current communication at different times times (places) in the algorithm, it was possible to find a communication strategy to improve the performance.

During the solution process of the *Golomb Ruler Problem*, POSL needs to perform many restarts. For that reason this problem was chosen to study a different (and innovative up tu my knowledge) communication strategy, in which the communicated information is a potential local minimum to be avoided. This new communication strategy showed to be effective to solve these kind of problems.

In all cases, thanks to the operator-based language provided by POSL it was possible to test many different strategies (communicating and non-communicating) fast and easily. Whereas

creating solvers implementing different solution strategies can be complex and tedious, POSL gives the possibility to make communicating and non-communicating solver prototypes and to evaluate them with few efforts. In this Chapter was possible to show that a good selection and management of inter-solvers communication can largely help to the search process, working with complex constrained problems.

3

Bibliography

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

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

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

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

[9] Ignasi Abío and Peter J Stuckey. Encoding Linear Constraints into SAT. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 75–91. Springer, 2014.

[10] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. *AIIDE*, pages 216–217, 2008.

[11] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–49, 2012.

[12] 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.

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

[14] 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.

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

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

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

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

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

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

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] Edmund K. Burke, Jingpeng Li, and Rong Qu. A hybrid model of integer programming and variable neighbourhood search for highly-constrained nurse rostering problems. *European Journal of Operational Research*, 203(2):484–493, 2010.

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

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

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

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

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

[58] 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.

[59] 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.

[60] 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.

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

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

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

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

[65] 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.

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

[69] 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.

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

[71] Christophe Lecoutre. XML Representation of Constraint Networks. Format XCSP 2.1. *Constraint Networks: Techniques and Algorithms*, pages 541–545, 2009.

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

[73] 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.

[74] 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.

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

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

[77] Javier Diaz, Camelia Muñoz-Caro, and Alfonso Niño. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

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

[79] 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.

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

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

[82] 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)*.

[83] 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.

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

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

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

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

[88] 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.

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

[90] Daisuke Ishii, Kazuki Yoshizoe, and Toyotaro Suzumura. Scalable Parallel Numerical CSP Solver. In *Principles and Practice of Constraint Programming*, pages 398–406, 2014.

[91] 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.

[92] 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.

[93] 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.

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

[95] 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.

[96] 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.

[97] 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.

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

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

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

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

[102] 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.

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

[104] 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.

[105] 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.

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

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

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

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

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

[111] 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.

[112] 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.

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

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

[115] 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.

[116] 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.

[117] 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.

[118] 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.

[119] Emmanuel Paradis. R for Beginners. Technical report, Institut des Sciences de l'Evolution, Université Montpellier II, 2005.

[120] Scott Rickard. Open Problems in Costas Arrays. In *IMA International Conference on Mathematics in Signal Processing at The Royal Agricultural College*, Cirencester, UK., 2006.

[121] Frédéric Lardeux, Éric Monfroy, Broderick Crawford, and Ricardo Soto. Set Constraint Model and Automated Encoding into SAT: Application to the Social Golfer Problem. *Annals of Operations Research*, 235(1):423–452, 2014.

[122] Konstantinos Drakakis. A review of Costas arrays. *Journal of Applied Mathematics*, 2006:32 pages, 2006.

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

[124] 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.

[125] Stephen W. Soliday, Abdollah. Homaifar, and Gary L. Lebby. Genetic algorithm approach to the search for Golomb Rulers. In *International Conference on Genetic Algorithms*, volume 1, pages 528–535, Pittsburg, 1995.

[126] Alejandro Reyes-amaro, Éric Monfroy, and Florian Richoux. POSL: A Parallel-Oriented metaheuristic-based Solver Language. In *Recent developments of metaheuristics*, to appear. Springer.

[127] Alejandro Reyes-Amaro, Éric Monfroy, and Florian Richoux. A Parallel-Oriented Language for Modeling Constraint-Based Solvers. In *Proceedings of the 11th edition of the Metaheuristics International Conference (MIC 2015)*. Springer, 2015.

Part III

Appendix

A

Results of experiments with *Social Golfers Problem*

In this Chapter we presents results values of experiments with Social Golfers Problem.

In the following tables: ...

Std: Simple Swap		AS: Adaptive Search swap		A000 / AS (rho) SS		A000 / AS U SS	
Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.
62,160	108	980	274	65,470	244	64,750	192
60,050	102	1,100	338	90,400	219	65,350	215
60,900	104	400	126	22,760	99	55,930	225
164,690	245	2,510	814	33,230	103	102,410	300
57,950	115	2,380	866	20,860	90	112,620	298
137,180	221	280	96	24,470	104	63,870	191
139,970	233	320	103	23,710	119	3,670	84
60,600	114	250	98	25,650	112	11,580	105
73,670	120	2,760	986	27,540	94	63,830	200
60,480	104	1,860	654	65,820	217	9,670	90
64,240	124	690	285	28,980	98	58,400	219
155,360	227	500	190	31,630	114	162,710	436
56,860	105	560	187	69,000	232	182,930	397
58,050	109	980	347	75,730	239	4,720	96
148,500	236	320	118	80,710	240	130,950	333
65,760	106	650	200	74,160	201	124,350	318
56,050	111	240	106	95,450	357	9,260	98
149,560	211	1,080	311	17,390	87	4,500	89
65,510	101	1,420	483	16,010	75	3,500	88
69,260	108	300	98	18,940	86	9,790	106
50,830	102	240	86	33,360	106	132,650	325
57,180	107	320	100	68,100	221	123,940	361
54,430	100	270	98	23,190	114	8,410	109
153,100	243	2,130	660	32,040	109	9,220	87
59,940	110	1,070	332	16,390	93	58,940	220
72,780	114	1,310	337	21,700	109	11,360	82
154,310	229	2,180	754	20,190	99	10,700	91
55,590	112	960	316	23,570	93	121,510	305
63,300	112	1,900	598	21,640	110	8,780	84
148,800	241	1,880	607	77,870	229	59,280	204
87,902	**146**	**1,061**	**352**	**41,532**	**147**	**59,653**	**198**
41,961	**58**	**791**	**268**	**26,001**	**72**	**55,019**	**110**

Table SGP 1: Comparing neighborhood functions. Social Golfers 10-10-3 (40 cores)

Caption: **mean**
Standard Deviation

Best Improvement		First Improvement		First Improvement (sequential)	
Time	Iter.	Time	Iter.	Time	Iter.
3,020	2,714	1,370	1,308	12,950	27,257
3,860	3,433	1,380	1,308	8,210	17,417
880	751	2,850	2,849	11,310	23,687
15,040	13,458	1,010	958	15,110	31,674
6,470	5,454	2,160	2,193	8,470	17,890
2,160	1,878	450	444	2,090	4,464
4,350	3,756	1,060	1,087	3,390	7,244
3,080	2,908	2,680	2,677	16,140	34,172
4,770	4,272	1,770	1,729	2,780	5,925
3,170	2,842	480	504	17,940	37,708
15,150	13,470	1,710	1,769	9,440	19,294
1,910	1,690	780	784	360	774
7,350	6,544	850	871	4,510	9,558
3,190	2,895	440	482	13,740	28,960
1,450	1,256	2,290	2,329	36,550	74,451
1,680	1,541	1,640	1,549	3,780	8,027
2,430	2,127	540	547	5,160	10,899
4,640	4,021	420	439	1,920	3,377
3,570	3,266	1,920	1,874	21,140	42,813
19,450	17,174	2,020	1,979	9,470	19,832
1,340	1,222	910	931	5,900	12,503
5,100	4,559	1,510	1,517	1,290	2,754
3,680	2,987	1,770	1,715	4,810	10,186
3,680	2,987	1,970	2,013	2,430	4,997
1,960	1,740	1,150	1,090	940	1,996
1,970	1,712	630	664	10,130	21,330
4,740	4,166	960	956	2,060	4,370
8,780	7,878	600	622	7,940	16,710
2,770	2,449	1,390	1,330	4,060	8,656
8,340	7,475	1,120	1,130	5,380	11,473
4,999	**4,421**	**1,328**	**1,322**	**8,313**	**17,347**
4,430	**3,938**	**682**	**676**	**7,640**	**15,673**

Table SGP 2: Comparing selection functions. Social Golfers 5-3-7 (40 cores, and sequential)

Caption: **mean**
Standard Deviation

100%/ com. 1-1		100% / com. 1-N		50% / com. 1-1		50% / com. 1-n	
Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.
1,080	1,083	520	506	870	848	690	735
1,600	1,562	860	845	1,060	1,087	1,020	965
740	723	580	575	580	580	840	844
860	856	820	763	870	841	710	715
520	609	2,130	2,139	480	484	570	557
2,340	2,352	1,270	1,151	1,000	953	1,340	1,248
1,720	1,650	1,030	979	2,460	2,458	310	282
870	834	820	791	610	579	980	989
1,100	1,068	520	454	1,610	1,559	210	209
1,280	1,317	730	682	1,710	1,744	1,150	1,112
100	116	1,220	1,234	1,280	1,300	900	921
90	116	1,790	1,649	1,580	1,527	380	387
1,730	1,670	540	517	1,100	1,090	1,620	1,549
740	680	1,160	1,056	1,260	1,195	1,600	1,613
1,380	1,291	860	761	780	746	1,000	1,025
1,390	1,271	1,230	1,199	340	342	1,120	1,078
2,670	2,488	1,790	1,727	1,210	1,156	2,360	2,364
1,160	1,163	1,990	1,976	730	741	1,020	961
1,410	1,381	1,880	1,793	1,140	1,042	960	961
1,730	1,657	810	780	720	737	1,340	1,233
1,860	1,727	1,820	1,584	1,240	1,189	740	738
2,000	1,946	1,290	1,231	620	541	1,160	1,171
590	601	1,180	1,080	1,300	1,205	280	277
1,980	1,828	490	458	750	702	1,810	1,742
770	776	1,370	1,278	830	826	2,000	2,037
590	604	890	858	1,400	1,302	650	618
1,590	1,493	1,080	1,095	250	227	2,020	2,050
320	289	690	678	1,470	1,419	680	676
1,050	961	480	464	1,000	1,036	1,040	1,002
580	568	1,740	1,700	1,140	1,127	880	870
1,195	1,156	1,119	1,067	1,046	1,019	1,046	1,031
646	608	499	484	459	456	533	530

Table SGP 3: Comparing communication strategies. Social Golfers 5-3-7 (40 cores)

Caption: **mean**

Standard Deviation

sender solver

receiver solver

receiver solver not tacking into account the received information

non-communicating solver

25% / com. 1-1		25% / com. 1-n	
Time	Iter.	Time	Iter.
1,820	1,745	510	518
980	905	1,330	1,266
610	547	990	994
2,030	1,884	890	844
410	419	1,350	1,264
330	321	380	363
1,710	1,675	1,430	1,305
390	382	820	807
1,560	1,451	1,090	1,079
790	774	2,180	2,072
520	450	730	727
470	481	1,100	1,163
1,350	1,370	810	849
550	563	830	819
820	776	2,040	2,005
580	587	910	871
280	312	990	926
450	444	1,440	1,388
1,310	1,282	690	643
570	542	2,740	2,622
1,210	1,251	960	904
1,370	1,330	450	441
1,200	1,187	680	708
400	393	150	169
260	272	2,820	2,805
870	850	1,550	1,527
1,610	1,616	610	682
470	483	1,380	1,372
1,260	1,195	2,380	2,385
930	932	1,600	1,594
904	881	1,194	1,170
514	492	675	655

Table SGP 4: Comparing communication strategies. Social Golfers 5-3-7 (40 cores)

Caption: **mean**

Standard Deviation

sender solver

receiver solver

receiver solver not tacking into account the received information

non-communicating solver

B001 / Best Impr.		B001 / First Impr.		First Improvement (sequential)	
Time	Iter.	Time	Iter.	Time	Iter.
6,300	1,192	1,720	424	12,760	5,994
5,220	991	1,620	385	1,560	739
3,130	573	2,990	718	22,930	10,742
5,450	1,023	1,390	361	6,120	2,888
2,690	502	4,700	1,104	3,330	1,560
5,810	1,100	1,450	373	5,470	2,631
3,250	623	1,170	300	7,700	3,443
3,880	724	1,780	417	10,140	4,541
6,270	1,163	1,220	308	11,960	5,574
2,880	541	2,320	551	1,280	640
1,520	279	1,660	406	11,450	5,302
4,180	783	2,100	518	48,190	22,314
5,460	980	880	220	18,640	8,735
8,270	1,564	2,190	537	11,670	5,413
4,240	797	1,470	361	16,190	6,423
4,650	836	1,530	387	770	419
8,230	1,548	1,580	380	2,780	1,333
4,790	885	940	255	37,270	17,453
4,920	907	4,110	962	30,880	14,246
7,850	1,455	2,280	537	14,470	6,841
6,340	1,197	1,600	390	24,590	11,291
3,260	605	1,450	343	22,820	10,632
3,470	653	2,090	498	20,230	9,409
7,380	1,384	950	261	58,420	27,138
5,170	973	1,630	405	3,470	1,637
3,200	594	2,020	491	16,220	7,521
5,500	1,046	1,660	401	25,120	11,640
7,920	1,466	1,190	313	500	264
5,570	1,035	1,330	324	49,260	22,696
6,340	1,189	1,720	419	11,470	5,409
5,105	**954**	**1,825**	**445**	**16,922**	**7,829**
1,776	**334**	**840**	**191**	**15,151**	**7,019**

Table SGP 5: Comparing selection functions. Social Golfers 8-4-7 (40 cores, and sequential)

Caption: **mean**

Standard Deviation

100%/ com. 1-1		100% / com. 1-N		50% / com. 1-1		50% / com. 1-n	
Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.
1,250	287	1,500	358	890	229	2,060	472
1,390	327	1,450	362	1,900	416	2,210	520
1,320	321	1,990	455	1,930	449	770	201
2,220	522	1,010	253	740	206	1,070	276
850	213	2,130	466	760	206	1,800	418
2,200	499	1,010	247	640	151	860	214
1,040	271	820	178	960	239	840	224
2,860	667	1,040	258	870	229	4,120	936
2,710	630	1,370	309	1,680	415	1,550	365
320	97	2,250	532	1,040	251	1,230	308
410	102	1,930	431	180	61	300	112
1,140	283	950	209	2,040	485	1,380	357
870	215	1,580	356	1,860	417	730	203
960	254	1,210	277	370	127	990	249
710	184	2,620	620	1,040	250	1,230	318
2,150	468	1,470	339	2,520	556	1,230	318
800	207	1,160	304	1,050	279	2,230	531
2,720	642	1,720	441	1,510	375	1,770	424
490	152	2,470	587	1,600	378	830	223
460	134	1,770	393	1,290	343	2,450	581
1,340	335	970	234	1,810	425	820	206
1,220	292	490	148	1,380	382	2,360	516
2,040	496	740	197	2,160	507	930	248
680	175	1,050	256	2,060	449	700	187
1,590	382	2,210	502	1,790	417	1,080	281
1,790	435	2,440	562	1,960	421	1,160	300
1,030	257	930	220	1,890	468	1,350	336
660	160	1,370	336	1,140	289	1,330	321
1,040	283	920	221	1,720	400	1,360	334
820	208	1,420	349	1,210	297	2,420	596
1,303	317	1,466	347	1,400	337	1,439	353
724	161	572	128	579	122	768	167

Table SGP 6: Comparing communication strategies. Social Golfers 8-4-7 (40 cores)

Caption: **mean**

Standard Deviation

sender solver

receiver solver

receiver solver not tacking into account the received information

non-communicating solver

25% / com. 1-1		25% / com. 1-n	
Time	Iter.	Time	Iter.
800	209	1,490	337
1,570	396	790	194
890	221	960	247
1,130	308	1,360	324
1,470	359	1,820	419
1,590	380	1,410	349
2,160	496	1,510	356
2,080	508	2,640	621
880	222	1,660	368
1,770	397	1,410	346
1,070	275	1,050	278
1,230	323	1,120	292
1,220	323	1,830	427
1,610	384	1,810	419
1,090	270	1,620	388
1,910	430	1,220	323
780	215	1,680	436
870	223	1,410	331
1,090	282	1,270	316
1,580	396	1,340	324
1,490	368	1,810	427
2,160	500	1,330	310
1,500	329	1,500	356
1,590	384	1,660	395
1,580	384	760	186
720	188	1,070	247
1,830	435	1,730	417
1,880	433	1,560	385
1,600	374	710	191
820	203	2,450	565
1,399	341	1,466	352
435	94	434	96

Table SGP 7: Comparing communication strategies. Social Golfers 8-4-7 (40 cores)

Caption: **mean**

Standard Deviation

sender solver

receiver solver

receiver solver not tacking into account the received information

non-communicating solver

B001 / Best Impr.		B001 / First Impr.		First Improvement (sequential)	
Time	Iter.	Time	Iter.	Time	Iter.
6,700	722	5,150	697	87,900	23,451
10,500	1,136	9,710	1,312	32,970	8,761
12,320	1,342	1,630	261	42,030	11,055
8,250	891	7,580	923	199,170	53,107
16,480	1,805	2,930	428	200,570	53,327
11,080	1,193	5,100	693	99,330	26,472
10,400	1,124	7,660	1,004	153,110	40,620
12,670	1,385	2,810	420	28,660	7,641
7,630	817	6,440	903	65,230	17,365
20,590	2,250	3,990	558	82,080	21,887
13,150	1,401	4,270	597	8,060	2,168
11,360	1,227	5,410	729	10,520	2,818
3,090	327	7,310	996	109,290	28,731
7,210	773	3,700	534	122,420	32,141
3,390	351	15,430	2,037	11,800	3,068
19,630	2,136	1,280	223	57,280	14,529
19,290	2,105	21,620	2,817	30,700	8,036
8,090	874	2,470	381	76,630	20,459
18,030	1,964	15,710	2,076	55,600	14,763
15,600	1,712	2,130	314	78,660	20,925
19,950	2,181	7,210	987	20,690	5,507
14,200	1,538	7,360	985	131,740	34,774
23,010	2,502	10,660	1,403	10,300	2,712
19,090	2,043	3,250	453	145,370	38,315
16,470	1,781	4,260	619	4,570	1,266
9,870	1,075	10,550	1,403	122,070	31,495
6,620	713	3,680	528	41,970	10,294
11,440	1,242	4,330	598	256,760	62,613
6,700	725	6,000	826	55,230	13,420
8,410	917	3,540	488	47,320	11,650
12,374	**1,342**	**6,439**	**873**	**79,601**	**20,779**
5,400	**591**	**4,607**	**591**	**64,072**	**16,537**

Table SGP 8: Comparing selection functions. Social Golfers 9-4-8 (40 cores, and sequential)

Caption: **mean**

Standard Deviation

100%/ com. 1-1		100% / com. 1-N		50% / com. 1-1		50% / com. 1-n	
Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.
2,050	292	6,020	809	4,590	640	8,980	1,143
4,200	594	4,010	573	6,570	888	14,360	1,909
1,880	281	4,770	641	5,530	774	7,770	999
3,530	516	5,560	755	5,180	714	3,800	529
1,000	175	5,250	686	3,360	477	1,950	282
5,960	793	6,180	832	2,750	401	5,870	759
3,730	532	15,050	2,001	4,050	565	5,570	755
2,610	362	5,660	752	8,980	1,195	2,200	286
3,990	529	8,910	1,150	8,920	1,161	7,180	970
4,760	633	8,610	1,098	2,180	299	5,420	745
3,460	462	1,620	256	6,570	925	1,580	255
9,290	1,187	4,480	608	8,040	1,083	8,860	1,185
2,820	387	5,300	709	2,970	415	6,110	846
10,050	1,357	1,030	166	3,380	469	5,590	787
11,170	1,465	2,690	393	4,390	550	3,300	442
4,200	565	5,970	803	3,130	421	4,090	555
2,340	324	3,100	418	2,840	405	3,930	572
4,140	561	10,100	1,311	3,550	500	5,550	757
4,920	669	3,700	507	2,450	369	4,540	631
2,810	412	800	147	6,140	847	9,990	1,301
7,010	957	7,250	952	1,660	240	11,020	1,427
10,160	1,313	3,790	507	3,840	538	8,060	1,067
3,380	465	6,020	765	3,350	478	4,590	647
1,960	315	4,230	583	5,430	733	3,100	438
4,750	667	6,370	873	3,090	456	2,050	316
2,760	365	2,360	330	2,760	408	3,200	488
2,270	349	4,890	684	5,830	764	4,210	593
6,480	856	3,050	403	7,520	1,004	6,390	862
2,220	331	11,060	1,395	8,510	1,103	3,040	424
1,510	202	7,560	975	1,920	290	10,290	1,324
4,380	597	5,513	736	4,649	637	5,753	776
2,720	347	3,066	389	2,173	279	3,067	389

Table SGP 6: Comparing communication strategies. Social Golfers 8-4-7 (40 cores)

Caption: **mean**

Standard Deviation

sender solver

receiver solver

receiver solver not tacking into account the received information

non-communicating solver

25% / com. 1-1		25% / com. 1-n	
Time	Iter.	Time	Iter.
3,280	472	7,040	923
6,380	876	7,440	994
6,570	882	1,920	315
3,150	405	2,180	348
1,990	293	3,940	510
4,010	545	5,510	754
5,130	658	5,130	689
5,460	750	3,210	465
6,360	856	4,050	565
4,030	564	3,110	434
8,970	1,235	5,160	714
1,660	270	2,580	384
1,710	252	2,360	359
3,530	503	2,450	372
4,130	588	4,370	602
3,950	592	3,660	530
1,630	272	7,130	991
5,660	775	2,790	403
7,110	914	9,110	1,178
5,500	709	2,790	393
3,870	541	2,550	390
2,700	366	4,560	591
3,290	450	6,390	873
6,430	851	2,340	345
7,860	1,083	5,560	766
3,330	486	4,750	651
2,840	412	6,480	843
3,220	459	7,160	950
3,700	512	7,700	1,021
2,520	389	2,630	394
4,332	599	4,535	625
1,920	248	2,019	251

Table SGP 7: Comparing communication strategies. Social Golfers 8-4-7 (40 cores)

Caption:

mean

Standard Deviation

sender solver

receiver solver

receiver solver not tacking into account the received information

non-communicating solver

B001 / Best Impr.		B001 / First Impr.		First Improvement (sequential)	
Time	Iter.	Time	Iter.	Time	Iter.
8,650	565	1,340	179	1,850	320
1,910	129	2,600	302	5,040	964
6,440	447	2,490	280	2,780	556
2,340	160	2,510	297	1,940	406
2,370	160	3,460	350	1,440	324
4,170	280	1,770	235	4,640	929
6,770	469	3,720	402	6,470	1,201
7,720	531	3,730	402	3,190	627
4,270	283	2,520	325	2,650	586
6,490	440	1,860	228	2,440	501
2,920	194	1,300	202	9,190	1,694
6,510	446	2,380	286	3,980	757
6,510	446	1,390	216	1,210	313
4,540	301	1,860	245	3,790	739
6,320	433	3,630	387	5,810	1,065
6,260	429	2,410	278	2,070	438
5,170	339	1,950	265	2,890	599
6,370	438	1,780	233	1,650	392
6,580	427	1,660	255	1,600	356
5,290	361	2,260	275	5,670	1,070
6,480	446	2,690	323	2,750	558
4,520	315	2,300	276	1,980	402
4,670	317	2,280	276	4,060	815
3,570	241	1,750	217	8,560	1,563
3,010	193	1,800	245	2,970	614
6,670	457	2,280	281	4,120	810
4,840	327	2,270	281	520	172
4,080	272	1,500	196	520	172
5,330	348	1,600	265	720	196
4,990	338	1,530	197	3,920	768
5,192	**351**	**2,221**	**273**	**3,347**	**664**
1,671	**114**	**692**	**58**	**2,167**	**380**

Table SGP 11: Comparing selection functions. Social Golfers 11-7-5 (40 cores, and sequential)

Caption: **mean**

Standard Deviation

100%/ – – >		100% / # – >		50% / – – >		50% / # – >	
Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.
1,330	180	1,790	216	1,010	154	1,390	151
2,240	223	1,530	207	1,960	220	1,810	238
1,590	215	1,200	177	1,540	220	2,030	249
2,090	216	1,580	217	2,640	276	2,260	253
1,790	215	1,340	150	1,970	230	2,030	276
1,590	216	1,530	166	1,640	209	2,080	248
1,710	187	1,080	146	2,160	195	1,540	213
1,010	141	2,570	286	1,640	183	1,510	181
1,790	189	1,580	205	1,640	219	2,280	258
2,370	221	1,530	196	2,290	257	1,390	176
1,990	220	1,300	198	1,800	228	1,300	217
1,990	220	1,540	203	1,800	218	1,640	179
1,320	180	1,640	170	2,670	298	1,340	199
1,890	249	1,330	194	1,640	206	1,400	200
1,130	143	2,140	251	1,620	234	2,170	210
1,250	150	1,720	189	2,160	232	1,940	259
1,390	210	1,190	160	2,130	279	2,040	187
1,210	159	1,910	203	1,900	237	1,900	206
1,240	149	2,130	221	1,630	216	1,600	201
1,830	204	1,900	243	1,840	197	1,280	153
1,610	227	1,790	223	1,540	206	2,550	275
1,710	226	1,430	173	2,290	248	2,500	301
2,420	272	1,360	188	1,200	198	1,830	230
1,820	240	1,360	185	1,410	187	1,360	186
1,830	240	1,990	219	1,910	254	1,570	224
2,220	270	1,400	208	1,680	211	2,660	288
2,280	266	1,520	195	1,570	193	1,920	255
1,680	192	2,270	253	1,050	145	1,890	196
1,810	233	1,770	212	1,890	223	1,430	202
2,730	352	1,340	191	2,250	234	2,020	252
1,762	214	1,625	202	1,816	220	1,822	222
420	45	348	31	401	33	395	39

Table SGP 12: Comparing communication strategies. Social Golfers 11-7-5 (40 cores)

Caption: **mean**

Standard Deviation

sender solver

receiver solver

receiver solver not tacking into account the received information

non-communicating solver

25% / – – >		25% / # – >	
Time	Iter.	Time	Iter.
2,330	268	2,180	250
2,590	300	2,150	258
2,600	300	1,760	237
2,600	300	1,760	260
1,420	211	2,510	277
2,680	280	2,470	271
1,830	221	2,420	286
2,190	261	2,400	286
2,220	268	1,840	220
770	139	2,120	221
1,280	170	1,710	218
2,020	210	2,040	244
1,520	207	2,050	244
2,030	263	2,150	258
2,550	269	1,760	241
1,140	180	1,390	197
2,530	280	1,540	196
2,520	319	1,730	220
1,530	201	2,200	246
2,580	286	1,930	261
1,560	161	1,090	179
1,510	215	1,800	195
1,980	270	1,200	179
2,130	271	1,510	209
1,640	211	1,810	229
2,150	265	1,680	233
1,260	150	1,830	250
1,470	204	1,500	218
2,730	320	2,070	225
2,440	274	2,080	237
1,993	242	1,630	224

Table SGP 12: Comparing communication strategies. Social Golfers 11-7-5 (40 cores)

Caption:

mean

Standard Deviation

sender solver

receiver solver

receiver solver not tacking into account the received information

non-communicating solver

B

Results of experiments with *Costas Array Problem*

In this Chapter we presents results values of experiments with Costas Array Problem.

In the following tables: ...

Sequential		No Communication	
Time	Iter.	Time	Iter.
3,410	70,278	1,810	19,861
3,140	67,274	250	4,672
2,070	43,561	120	1,234
340	7,381	440	5,205
3,010	61,855	1,570	16,639
3,150	64,966	340	4,840
2,030	43,528	450	4,831
1,280	26,578	120	2,306
2,280	47,955	1,020	10,638
1,340	27,925	200	2,539
2,240	46,846	1,160	14,293
2,610	53,341	910	16,289
740	15,529	550	6,652
2,050	43,846	700	12,914
1,600	33,460	280	2,884
2,640	55,891	1,160	11,834
1,980	41,479	850	14,665
1,440	29,980	390	4,630
3,080	63,730	890	9,382
3,090	65,338	560	9,262
1,100	22,774	1,580	16,999
		1,620	29,008
		310	5,462
		280	2,974
		180	3,367
		380	3,631
		270	3,064
		1,000	10,571
		1,280	14,581
		230	4,099
		100	1,752
		560	6,574
		1,150	12,022
		720	10,345
		1,210	13,006
		280	3,023
		110	1,195
		990	10,448
		910	9,787
		1,020	18,244
		920	10,564
		730	10,939
		780	12,232
		1,430	24,616
		900	15,970
2,125	44,453	727	9,557
871	18,113	469	6,439

Table CAP 1: No communication (1 and 40 cores) Costas Array 17

St_a / 100% com . 1-1		St_a / 100% com. 1-n		St_a / 50% com. 1-1		St_a / 50% com. 1-n	
Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.
860	9,481	1,000	13,702	330	3,964	760	12,946
810	7,954	170	1,648	550	10,750	140	1,963
830	7,870	350	4,172	840	15,835	120	1,384
240	2,947	350	3,617	280	3,019	780	9,553
180	2,252	210	2,593	960	11,142	720	12,013
300	3,106	200	1,976	1,030	15,841	20	172
280	3,978	160	1,617	340	5,251	570	7,855
190	2,392	370	4,336	330	4,153	110	1,259
200	1,868	1,040	16,596	260	3,313	290	3,496
740	6,670	150	1,891	290	4,834	1,270	22,888
380	6,395	230	2,566	10	119	1,020	20,027
200	2,320	800	9,874	170	3,730	1,430	16,756
700	9,247	920	9,599	370	8,019	460	6,115
170	1,700	130	1,486	1,640	23,068	290	5,632
1,030	10,403	780	8,361	1,060	20,573	40	571
760	13,045	800	10,440	360	5,329	1,700	23,350
760	7,660	340	4,647	410	5,146	430	8,873
500	6,751	1,040	13,030	580	7,987	420	5,695
160	1,360	380	4,243	810	9,737	990	11,596
140	1,528	360	3,763	90	1,112	250	3,478
120	1,198	500	6,985	220	2,644	210	2,500
380	4,804	110	1,312	530	10,198	300	3,691
220	1,981	100	1,066	1,310	13,846	490	7,138
180	2,833	150	2,560	70	1,348	760	7,840
190	1,928	560	5,119	570	6,703	440	7,016
980	8,974	590	9,214	550	6,703	1,540	30,901
1,130	10,750	650	7,919	20	427	1,640	19,264
260	2,572	160	1,922	300	3,779	1,380	27,048
380	3,324	260	2,638	180	2,364	180	3,737
120	1,465	80	1,035	750	8,929	720	8,287
40	689	540	11,224	630	9,292	1,430	24,226
790	13,384	760	13,072	50	681	660	6,959
80	844	370	5,209	600	7,396	160	2,191
610	5,650	110	2,298	1,330	15,478	470	9,328
470	4,748	120	1,182	780	14,206	40	402
100	1,036	420	4,527	790	12,373	350	4,260
50	562	100	1,069	100	1,171	490	10,333
480	7,780	80	1,054	340	4,189	960	11,074
510	4,883	690	7,756	260	3,493	60	616
630	11,797	80	871	620	8,236	30	328
610	6,125	1,100	18,772	510	5,246	380	5,173
260	2,734	80	872	660	6,893	220	2,638
270	3,322	690	7,912	20	241	480	6,091
240	2,500	910	14,065	1,330	21,556	960	12,664
210	2,041	420	7,740	1,920	39,955	320	5,176
416	4,819	431	5,723	559	8,228	588	8,767
296	3,589	317	4,744	443	7,556	477	7,774

Table CAP 2: Strategy A (40 cores) Costas Array 17

Caption: **mean**

Standard Deviation

sender solver

receiver solver

receiver solver not tacking into account the received information

non-communicating solver

St_b / 100% com. 1-1		St_b / 100% com. 1-n		St_b / 50% com. 1-1		St_b / 50% com. 1-n	
Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.
10	322	2,420	32,411	990	13,492	220	4,852
320	5,560	1,690	17,034	130	1,654	2,190	27,028
790	9,226	650	6,397	960	19,321	810	13,510
40	541	1,200	22,472	570	12,136	80	814
1,280	20,420	640	6,389	230	4,858	680	12,859
1,240	12,358	610	5,670	740	8,590	630	6,412
850	10,123	920	9,427	410	5,498	120	1,390
280	4,840	250	3,058	320	4,429	440	4,309
880	10,507	960	16,715	300	4,039	490	5,614
430	5,125	940	9,898	320	4,123	450	9,082
60	625	100	1,243	1,180	15,841	110	1,252
340	6,875	870	17,608	420	7,598	2,330	42,391
60	583	180	1,885	50	616	140	1,780
980	11,539	790	7,738	1,620	18,894	120	1,557
90	949	470	5,982	1,890	26,893	500	5,965
230	2,281	130	1,366	130	1,924	1,310	27,804
830	12,990	120	1,780	1,280	13,420	500	7,017
550	6,721	170	1,932	160	1,861	430	5,332
330	6,537	40	432	10	211	290	3,736
320	3,317	320	3,547	110	1,369	80	859
760	11,194	200	3,073	120	1,573	160	3,142
420	4,405	80	1,360	100	872	40	507
160	1,955	20	130	60	691	100	1,219
90	1,105	430	7,915	1,050	14,983	580	6,859
200	3,649	30	373	1,670	34,654	100	1,486
300	3,982	210	2,869	490	6,250	170	2,440
160	1,748	50	544	1,560	26,260	200	2,719
790	8,998	470	5,890	10	187	40	382
280	3,715	370	3,932	130	1,246	430	6,370
120	1,318	170	2,164	300	4,066	630	7,480
1,370	14,143	90	1,546	290	3,562	360	4,621
390	4,216	1,160	14,086	1,030	13,999	1,500	17,857
630	7,579	120	1,447	500	10,426	1,240	16,702
170	1,990	120	1,186	300	3,451	660	9,184
90	1,012	300	5,012	110	1,441	160	1,642
1,590	16,567	300	3,133	970	20,263	200	2,668
320	3,976	20	280	40	988	170	2,218
1,310	17,230	470	6,574	320	6,691	260	2,918
120	1,567	260	2,423	480	10,150	620	13,171
190	3,301	90	1,048	640	6,592	1,030	19,681
210	2,143	360	3,436	20	451	90	1,036
470	9,954	600	7,060	90	1,609	1,100	13,264
360	5,307	290	3,217	480	8,611	440	4,471
160	1,691	380	5,341	270	3,601	280	3,415
1,020	17,737	280	2,581	940	15,943	210	2,746
480	6,265	452	5,769	529	8,118	504	7,372
417	5,321	477	6,578	506	8,192	524	8,544

Table CAP 3: Strategy B (40 cores) Costas Array 17

Caption: **mean**

Standard Deviation

sender solver

receiver solver

receiver solver not tacking into account the received information

non-communicating solver

St_c / 100% com. 1-1		St_c / 100% com. 1-n		St_c / 50% com. 1-1		St_c / 50% com. 1-n	
Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.
40	340	960	18,532	1,990	24,955	70	793
200	1,936	980	10,132	40	551	360	5,321
180	1,966	290	3,196	720	15,226	220	2,374
1,170	11,605	540	5,428	730	7,378	210	2,416
520	10,654	1,300	26,122	1,200	14,257	540	7,306
390	5,275	180	2,005	720	9,265	40	583
380	4,012	140	2,065	870	10,972	410	4,288
660	6,797	170	1,849	270	5,617	870	12,076
340	4,321	180	2,233	190	2,620	580	11,218
240	2,527	400	4,198	1,010	19,234	720	8,242
320	3,799	680	7,546	1,280	15,586	1,600	30,455
310	3,484	780	7,377	330	4,636	1,600	15,751
400	4,138	890	11,849	870	11,847	180	2,164
1,100	21,040	870	9,637	300	4,129	2,080	25,051
1,110	19,507	110	1,406	80	1,066	1,350	18,460
240	3,493	120	1,315	410	5,167	1,380	13,921
1,250	25,261	250	2,574	1,570	29,595	440	4,351
320	6,529	660	9,044	860	9,987	380	4,663
360	4,054	200	2,683	80	1,765	370	5,008
340	3,547	630	12,604	390	3,992	230	4,882
1,070	10,996	630	6,409	310	3,754	320	3,643
320	3,574	570	5,563	960	10,719	130	2,618
340	7,062	800	7,642	970	11,053	400	4,690
360	4,459	500	10,312	910	17,042	230	2,917
1,100	19,301	500	8,659	1,150	15,991	400	4,807
160	1,672	590	6,070	320	3,959	110	1,393
1,410	16,363	130	1,423	120	2,460	260	3,580
150	1,615	1,820	30,643	520	8,194	1,280	25,900
300	3,160	830	16,030	520	5,842	400	5,084
1,310	15,066	720	8,838	590	12,658	300	4,077
1,360	25,852	740	8,437	220	3,193	290	2,944
590	12,271	100	1,027	100	1,184	200	2,211
230	2,788	600	7,387	10	241	210	2,983
10	313	580	6,493	700	8,263	220	2,239
390	8,149	170	1,957	390	5,312	310	3,288
390	6,727	150	1,771	720	11,386	770	7,825
370	7,654	340	4,627	160	2,080	1,790	22,204
20	250	570	6,751	640	12,271	180	1,978
120	1,518	500	5,932	870	10,204	1,710	36,013
100	1,201	20	265	530	5,959	470	4,819
170	2,237	280	3,910	390	8,155	660	12,346
230	2,767	250	2,785	160	1,985	910	10,369
490	6,028	190	2,368	810	10,024	920	11,116
480	8,626	180	3,091	180	3,850	440	4,474
920	9,349	450	5,035	280	3,375	420	5,179
495	7,184	501	6,783	588	8,378	599	8,178
400	6,632	359	6,218	432	6,437	525	8,289

Table CAP 4: Strategy C (40 cores) Costas Array 17

Caption: **mean**

Standard Deviation

sender solver

receiver solver

receiver solver not tacking into account the received information

non-communicating solver

C

Results of experiments with *Golomb Ruler Problem*

In this Chapter we presents results values of experiments with Golomb Ruler Problem.

In the following tables: ...

Using Tabu List (no communication)		
time	iterations	restarts
110	743	3
100	768	3
50	518	2
20	88	0
20	202	1
30	233	1
20	197	0
10	112	0
20	86	0
40	229	1
10	88	0
20	88	0
20	89	0
100	970	4
30	173	0
40	401	2
20	169	0
30	173	0
30	196	0
170	1543	7
90	576	2
60	602	3
60	570	2
60	547	2
10	86	0
10	138	0
20	142	0
10	87	0
50	487	2
20	167	0
43	**349**	**1**
38	**334**	**2**

Without Tabu List (no communication)		
time	iterations	restarts
70	685	3
50	432	2
50	400	1
90	801	4
50	371	1
60	603	3
110	1118	5
30	286	1
40	378	1
30	315	1
30	259	1
50	431	2
20	204	1
130	1290	6
100	961	4
60	604	3
20	263	1
20	194	0
10	112	0
30	290	1
20	166	0
40	343	1
10	114	0
140	1200	5
30	259	1
20	203	1
40	260	1
10	88	0
20	202	1
40	259	1
47	**436**	**2**
35	**330**	**2**

Table GRP 1: No communication (40 cores) Golomb Ruler 8-34

Caption: **mean** **Standard Deviation**

Eps = 4
Norm = 8
tabu size = 15

Using Tabu List (communication 1-1)		
time	iterations	restarts
40	431	2
30	199	0
30	140	0
30	316	1
80	569	2
60	340	1
40	202	1
30	140	0
30	113	0
160	1202	6
90	570	2
60	430	2
30	262	1
50	395	1
30	143	0
40	337	1
30	263	1
40	312	1
20	193	0
20	120	0
30	286	1
30	112	0
20	174	0
10	89	0
80	573	2
20	60	0
30	289	1
70	460	2
10	4	0
90	543	2
44	309	1
31	233	1

Using Tabu List (communication 1-n)		
time	iterations	restarts
20	87	0
10	30	0
20	113	0
80	543	2
70	403	2
10	57	0
70	460	2
40	203	1
10	121	0
10	62	0
80	594	2
40	203	1
30	203	1
10	87	0
70	342	1
90	630	3
40	368	1
10	3	0
80	632	3
60	488	2
20	203	1
10	30	0
70	492	2
40	170	0
80	570	2
40	261	1
100	768	3
10	60	0
50	289	1
10	3	0
43	283	1
30	225	1

Table GRP 2: Communication (40 cores) Golomb Ruler 8-34

Eps = 4
Norm = 8
tabu size = 40

Caption:

mean

Standard Deviation

sender solver

receiver solver

Sequential (without tabu list)		
time	iterations	restarts
80	1232	6
950	15915	79
400	6803	34
600	10316	51
880	14802	74
2050	33658	168
2580	43742	218
200	3313	16
690	10887	54
1640	26832	134
1040	17171	85
210	3486	17
240	4082	20
100	1717	8
130	2483	12
1040	18059	90
1720	28799	143
1070	18138	90
1290	21230	106
40	740	3
460	7604	38
100	1741	8
940	15030	75
110	1887	9
1760	28857	144
420	7172	35
1440	25339	126
780	13258	66
220	3857	19
670	11030	55
795	**13,306**	**66**
668	**11,154**	**56**

Sequential (with tabu list)		
time	iterations	restarts
1090	17140	85
10	233	1
240	3860	19
2190	36030	180
2350	37203	186
700	11366	56
150	2634	13
740	11461	57
870	13795	68
80	1260	6
50	994	4
450	7202	36
970	15796	78
510	8142	40
100	1547	7
1230	20487	102
500	8287	41
560	9311	46
860	14089	70
70	1345	6
510	8343	41
450	7342	36
80	1401	7
260	4340	21
1960	31257	156
260	4086	20
1340	21514	107
250	3942	19
30	546	2
1070	17399	86
664	**10,745**	**53**
639	**10,260**	**51**

Table GRP 3: Sequential (1 core) Golomb Ruler 8-34

Caption: **mean**

Standard Deviation

Eps = 4
Norm = 8
tabu size = 15

Using Tabu List (no communication)		
time	iterations	restarts
4,740	20,069	13
13,990	56,986	37
7,980	33,958	22
6,810	28,162	18
9,300	37,336	24
21,410	89,951	59
3,340	13,506	9
3,190	13,458	8
1,230	4,449	2
2,320	9,792	6
5,520	23,529	15
4,330	17,268	11
3,170	12,892	8
1,050	3,734	2
8,090	34,830	23
5,310	21,741	14
2,950	12,597	8
620	2,892	1
11,370	48,732	32
2,790	11,731	7
2,400	9,733	6
2,110	8,268	5
11,010	47,398	31
1,560	6,140	4
3,270	13,937	9
640	2,355	1
990	3,698	2
480	1,768	1
3,120	12,896	8
2,600	11,322	7
4,923	**20,504**	**13**
4,687	**19,743**	**13**

Without Tabu List (no communication)		
time	iterations	restarts
5,360	21,072	14
1,700	7,295	4
6,590	27,135	18
900	3,466	2
5,710	24,439	16
4,000	16,161	10
4,760	21,600	14
1,480	5,834	3
15,100	63,697	42
12,840	54,234	36
4,270	19,252	12
7,360	32,230	21
7,330	32,695	21
2,370	10,163	6
6,890	29,354	19
1,080	4,570	3
10,340	44,978	29
13,190	56,969	37
8,370	36,197	24
3,260	14,755	9
970	4,450	2
920	4,247	2
4,250	17,070	11
3,630	15,300	10
8,690	36,498	24
2,000	7,965	5
3,760	15,501	10
7,390	29,296	19
3,370	14,731	9
1,430	6,169	4
5,310	**22,577**	**15**
3,863	**16,488**	**11**

Table GRP 4: No communication (40 cores) Golomb Ruler 10-55

Caption: **mean**

Standard Deviation

Eps = 4
Norm = 8
tabu size = 15

Using Tabu List (communication 1-1)		
time	iterations	restarts
1,370	5,957	3
440	1,613	1
6,440	26,337	17
590	2,402	1
310	1,258	0
1,790	7,891	5
2,880	11,075	7
1,700	6,537	4
260	797	0
550	2,409	1
5,750	21,993	14
6,800	26,270	17
9,100	35,963	23
1,070	4,018	2
3,010	11,320	7
4,590	17,455	11
10,120	40,753	27
3,620	14,283	9
3,080	12,199	8
430	1,674	1
9,580	39,655	26
3,060	11,673	7
7,210	28,998	19
1,310	5,008	3
6,620	27,298	18
8,850	35,597	23
2,480	10,118	6
9,170	32,662	21
1,460	5,952	3
3,460	13,935	9
3,903	15,437	10
3,225	12,789	9

Using Tabu List (communication 1-n)		
time	iterations	restarts
8,180	35,159	23
7,920	33,205	22
1,350	5,190	3
3,030	11,119	7
3,460	15,232	10
630	2,326	1
9,300	36,107	24
1,680	6,760	4
170	757	0
5,520	21,400	14
210	758	0
1,100	4,314	2
8,150	32,458	21
1,510	5,813	3
460	1,897	1
2,230	8,359	5
3,650	14,376	9
1,240	4,771	3
580	2,432	1
460	1,743	1
5,560	22,256	14
1,470	5,451	3
960	3,858	2
1,280	5,100	3
4,800	20,616	13
3,760	14,826	9
1,620	6,434	4
1,370	5,199	3
7,910	29,852	19
5,290	20,388	13
3,162	12,605	8
2,824	11,405	8

Table GRP 5: Communication (40 cores) Golomb Ruler 10-55

Eps = 4
Norm = 8
tabu size = 40

Caption:

mean

Standard Deviation

sender solver

receiver solver

Sequential (without tabu list)		
time	iterations	restarts
11,540	85,707	57
106,120	759,399	506
58,220	411,362	274
49,070	342,733	228
20,200	146,027	97
47,620	327,598	218
55,600	372,135	248
90,620	610,539	407
7,650	52,896	35
145,150	966,493	644
33,210	224,551	149
96,990	662,726	441
47,300	318,721	212
15,960	108,936	72
18,560	130,505	87
7,480	50,949	33
68,200	471,494	314
92,600	620,422	413
22,950	156,637	104
45,450	306,110	204
34,570	188,025	125
112,110	768,339	512
142,360	965,061	643
142,300	960,895	640
78,720	528,396	352
56,810	389,701	259
12,120	82,335	54
58,860	387,600	258
208,790	1,417,178	944
106,150	729,103	486
66,443	**451,419**	**301**
49,569	**336,859**	**225**

Sequential (with tabu list)		
time	iterations	restarts
17,720	117,993	78
146,880	993,766	662
63,660	428,626	285
10,090	67,926	45
75,560	514,704	343
53,700	372,893	248
147,140	1,011,234	674
57,570	400,503	267
44,530	295,743	197
40,970	277,320	184
410	2,791	1
40,840	271,494	180
57,140	382,793	255
38,570	256,086	170
134,890	894,727	596
91,410	608,716	405
57,340	389,626	259
105,230	728,128	485
114,190	790,798	527
110,290	758,168	505
23,310	157,930	105
14,320	88,766	59
151,240	1,007,429	671
88,750	600,499	400
39,030	257,327	171
51,060	221,101	147
37,810	198,660	132
24,270	156,268	104
5,610	32,591	21
193,370	1,122,786	748
67,897	**446,913**	**297**
50,024	**328,912**	**219**

Table GRP 6: Sequential (1 core) Golomb Ruler 10-55

Caption: **mean**

Standard Deviation

Eps = 4
Norm = 8
tabu size = 15

Using Tabu List (no communication)		
time	iterations	restarts
32170	58451	19
86650	156935	52
161460	293271	97
220280	398176	132
74050	136241	45
66420	119174	39
179500	327108	109
117350	212795	70
103960	192214	64
9800	17999	5
25290	45418	15
108060	201626	67
119020	219628	73
18550	34456	11
210190	384418	128
53520	100043	33
127630	230691	76
52620	96214	32
214640	383587	127
28080	50999	16
79130	146484	48
55850	103242	34
91340	167080	55
7270	12421	4
184280	338482	112
1620	2999	0
12520	22971	7
3640	6928	2
34220	64652	21
71590	132829	44
85,023	**155,251**	**51**
67,223	**121,929**	**41**

Without Tabu List (no communication)		
time	iterations	restarts
273700	508450	169
30790	56778	18
103280	192006	64
53230	95905	31
41540	78212	26
58890	104483	34
90190	167372	55
133910	248677	82
82350	152068	50
150320	273317	91
1430	2785	0
95300	173065	57
173490	311376	103
51820	95176	31
21590	40349	13
40000	73144	24
123830	232758	77
82010	149688	49
60880	113789	37
98000	181972	60
56980	104485	34
129980	236088	78
98100	180725	60
78750	146276	48
166690	302689	100
82660	151445	50
92160	164589	54
142440	263995	87
54130	98081	32
24350	43139	14
89,760	**164,763**	**54**
55,859	**102,931**	**34**

Table GRP 7: No communication (40 cores) Golomb Ruler 11-72

Caption: **mean**

Standard Deviation

Eps = 4
Norm = 8
tabu size = 15

Using Tabu List (communication 1-1)		
time	iterations	restarts
58150	104687	34
211410	387838	129
5580	10135	3
81170	151964	50
44770	80806	26
92430	163137	54
62460	111108	37
48680	87420	29
50410	89278	29
192780	356580	118
84510	158490	52
174050	321314	107
62130	110587	36
116400	209895	69
69700	127352	42
78790	143793	47
53510	100957	33
42740	76143	25
73150	131271	43
38160	72321	24
12310	22243	7
96690	179268	59
190280	350380	116
78940	143379	47
98090	180525	60
159820	295856	98
96030	175445	58
78610	141931	47
24700	44064	14
86470	158172	52
85,431	156,211	52
52,610	97,330	32

Using Tabu List (communication 1-n)		
time	iterations	restarts
170270	312109	104
62810	111833	37
14140	26681	8
53730	97756	32
16230	28758	9
24030	42519	14
39870	74069	24
59800	107992	35
16660	30007	10
79380	146171	48
95520	175044	58
118760	217352	72
21140	37044	12
19580	34345	11
124340	224683	74
85260	152680	50
10690	19041	6
75000	136039	45
26580	46451	15
154340	287165	95
7170	12523	4
44110	81521	27
90380	168320	56
32580	59889	19
47430	83487	27
114890	215689	71
47620	85957	28
13730	24413	8
77170	144113	48
67510	125686	41
60,357	110,311	36
43,951	81,296	27

Table GRP 8: Communication (40 cores) Golomb Ruler 11-72

Eps = 4
Norm = 8
tabu size = 40

Caption:

mean

Standard Deviation

sender solver

receiver solver

Sequential (without tabu list)		
time	iterations	restarts
279920	536214	178
123870	246321	82
158860	486633	162
59290	187837	62
298830	914400	304
216040	673087	224
98300	301330	100
47670	107164	35
160,348	**431,623**	**143**
96,116	**272,911**	**91**

Sequential (with tabu list)		
time	iterations	restarts
26420	80481	26
126880	414007	138
140600	462828	154
68180	214270	71
272440	859895	286
10300	34398	11
124410	417131	139
72180	243698	81
216000	716845	238
117,490	**382,617**	**127**
85,624	**275,748**	**92**

Table GRP 9: Sequential (1 core) Golomb Ruler 11-72

Caption: **mean**

Standard Deviation

Eps = 4
Norm = 8
tabu size = 15