POSL: A Parallel-Oriented metaheuristic-based Solver Language

Alejandro REYES AMARO, Eric MONFROY, and Florian RICHOUX

Abstract For a couple of years, all processors in modern machines are multi-core. Massively parallel architectures, so far reserved for super-computers, become now available to a broad public through hardware like the Xeon Phi or GPU cards. This architecture strategy has been commonly adopted by processor manufacturers, allowing them to stick with Moore's law. However, this new architecture implies new ways to design and implement algorithms to exploit its full potential. This is in particular true for constraint-based solvers dealing with combinatorial optimization problems. Here we propose a Parallel-Oriented Solver Language (POSL, pronounced "puzzle"), a new framework to build interconnected meta-heuristic based solvers working in parallel. The novelty of this approach lies in looking at solver as a set of components with specific goals, written in a parallel-oriented language based on operators. A major feature in POSL is the possibility to share not only information, but also behaviors, allowing solver modifications during runtime. Our framework has been designed to easily build constraint-based solvers and reduce the developing effort in the context of parallel architecture. POSL's main advantage is to allow solver designers to quickly test different heuristics and parallel communication strategies to solve combinatorial optimization problems, usually time-consuming and very complex technically, requiring a lot of engineering.

Alejandro REYES AMARO
LINA Inria–TASC, Université de Nantes, 2 rue de la Houssinière, Nantes, e-mail: alejandro.reyes@univ-nantes.fr

Eric MONFROY
LINA Inria–TASC, Université de Nantes, 2 rue de la Houssinière, Nantes e-mail: eric.monfroy@univ-nantes.fr

Florian RICHOUX
LINA Inria–TASC, Université de Nantes, 2 rue de la Houssinière, Nantes e-mail: florian.richoux@univ-nantes.fr

1 Introduction

Combinatorial Optimization has strong applications in several fields, including machine learning, artificial intelligence, and software engineering. In some cases, the main goal is only to find a solution, like for *Constraint Satisfaction Problems (CSP)*. A solution will be an assignment of variables satisfying the constraints set. In other words: finding one feasible solution.

*CSP*s find a lot of applications in the industry, implying the development of many methods to solve them. Meta–heuristics techniques have shown themselves to be effective for solving *CSP*s, but in most industrial cases the search space is huge enough to be intractable. However, recent advances in computer architecture are leading us toward massively *multi/many–core* computers, opening a new way to find solutions for these problems in a more feasible manner, reducing search time. Adaptive Search [5] is an efficient methods showing very good performances scaling to hundreds or even thousands of cores, using a multi-walk local search method. For this algorithm, an implementation of a cooperative multi-walks strategy has been published in [9]. These works have shown the efficiency of multi-walk strategy, that is why we have oriented POSL towards this parallel scheme.

In the last years, a lot of efforts have been made in parallel constraint programming. In this field, the inter-process communication for solver cooperation is one of the most critical issues. [11] presents a paradigm that enables the user to properly separate strategies combining solver applications in order to find the desired result, from the way the search space is explored. *Meta–S* is an implementation of a theoretical framework proposed in [6], which allows to tackle problems, through the cooperation of arbitrary domain–specific constraint solvers. POSL provides a mechanism of creating solver–independent communication strategies, making easy the study of solving processes and results. Creating solvers implementing different solution strategies can be complex and tedious. In that sense POSL gives the possibility of prototyping communicating solvers with few efforts.

In Constraint Programming, many researches focus on fitting and improving existing algorithms for specific problems. However, it requires a deep study to find the right algorithm for the right problem. HYPERION [3] is a Java framework for meta– and hyper–heuristics built with the principle of interoperability, generality by providing generic templates for a variety of local search and evolutionary computation algorithms and efficiency, allowing rapid prototyping with the possibility of reusing source code. POSL aims to offer the same advantages, but provides also a mechanism to define communication protocols between solvers.

In this chapter we present POSL, a framework for easily building many and different cooperating solvers based on coupling four fundamental and independent components: *operation module*s, *open channel*s, the *computation strategy* and *communication channels* or *subscription*s. Recently, the hybridization approach leads to very good results in constraint satisfaction [14]. *ParadisEO* is a framework to design parallel and distributed hybrid meta-heuristics showing very good results [4]. It includes a broad range of reusable features to easily design evolutionary algorithms and local search methods. Our framework POSL focuses only in local search

methods, but is designed to execute in parallel sets of different solvers, with and/or without communication, since the solver's components can be combined by using operators.

POSL provides, through a simple operator-based language, a way to create a *computation strategy*, combining already defined *components* (*operation module*s and *open channel*s). A similar idea was proposed in [7] without communication, introducing an evolutionary approach that uses a simple composition operator to automatically discover new local search heuristics for SAT and to visualize them as combinations of a set of building blocks. Another interesting idea is proposed in TEMPLAR, a framework to generate algorithms changing predefined components using hyper-heuristics methods [13]. In the last phase of the coding process with POSL, solvers can be connected each others, depending on the structure of their *open channel*s, and this way, they can share not only information, but also their behavior, by sharing their *operation module*s. This approach makes the solvers able to evolve during the execution.

Before ending this chapter with a brief conclusion and future works, we present some results obtained by using POSL to solve some instances of the *Social Golfers Problem*.

2 POSL parallel solvers

POSL proposes a solver construction platform following different stages. First of all, the solver algorithm is modeled by decomposing it into small pieces/modules of computation. After that, they are implemented as separated *functions*. We name them *operation module*. The next step is to decide what information is interesting to receive from other solvers. This information is encapsulated into other objects called *open channel*s, allowing data transmission among solvers. In a third stage, a generic strategy is coded through POSL, using the mentioned components in the previous stages, allowing not only the information exchange, but also to execute the components in parallel. This will be the solver's backbone. Finally, solvers are defined by instantiating and connecting the *strategy*, *operation module*s and *open channel*s, and by connecting them each others. The next subsections explain in details each of these steps.

2.1 Operation module

An *operation module* is the most basic and abstract way to define a piece of computation. It can be dynamically replaced by or combined with other *operation module*s, since they can be sheared among solvers working in parallel. This way, the solver can mutate its behavior during execution.

An *operation module* receives an input, executes an internal algorithm and gives an output. They are joined through *computation strategies*.

Definition 1. (Operation Module) An *operation module Om* is a mapping defined by:

$$Om : D \rightarrow I \tag{1}$$

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

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

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

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

2.2 Open channels

Open Channels are the solver's components in charge of the information reception in the communication between solvers. They can interact with *operation modules*, depending on the *computation strategy*. *Open Channels* play the role of outlets, allowing solvers to be connected and to share information.

An *open channel* can receive two types of information, always coming from an external solver: data or *operation module*s. It is important to notice that when we are talking about sending/receiving *operation module*s, we mean sending/receiving only required information to identify it and being able to instantiate it.

In order to distinguish between the two different types of *open channel*s, we will call Data Open Channel the *open channel* responsible for the data reception, and Object Open Channel the one responsible for the reception and instantiation of *operation module*s.

Definition 2. (Data Open Channel) A *Data Open Channel Ch* is a component that produces a mapping defined as follows:

$$Ch : U \rightarrow I \tag{2}$$

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

Definition 3. (Object Open Channel) If we denote by $\mathbb{M}$ the space of all the *operation module*s defined by Definition 1, then an *Object Open Channel Ch* is a component that produces an *operation module* coming from an external solver as follows:

$$Ch : \mathbb{M} \rightarrow \mathbb{M} \tag{3}$$

Due to the fact that *open channel*s receive information coming from outside and have no control on them, it is necessary to define the *NULL* information, to denote the absence of any information. If a Data Open Channel receives a piece of information, it is returned automatically. If a Object Open Channel receives an *operation module*, the latter is instantiated and executed with the *open channel*'s input, and its result is returned. In both cases, if no available information exists (no communications are performed), the *open channel* returns the *NULL* object.

2.3 Computation strategy

The *computation strategy* is the solver's backbone: it joins *operation module*s and *open channel*s in a coherent way, while remaining independent from them. Through the *computation strategy* we can decide also what information to sent to other solvers.

The *computation strategy* is an operator–based language, that we define as a free-context grammar as follows:

Definition 4. (POSL**'s Grammar**) $G_{POSL} = (\mathbf{V}, \Sigma, \mathbf{S}, \mathbf{R})$, where:

1. $\mathbf{V} = \{CM, OP\}$ is the set of *variables*,
2. $\Sigma = \left\{ om, och, be, [,], [\![,]\!]_p, (,), \{,\}, (\!|, |\!)^m, |\!)^o, \longmapsto, \circlearrowleft, \textcircled{ρ}, \textcircled{$\vee$}, \textcircled{M}, \textcircled{m}, \textcircled{$\downarrow$} \right\}$ is the set of *terminals*,
3. $\mathbf{S} = \{CM\}$ is the set of *start variables*,
4. and the set of *rules* $\mathbf{R} =$

$$CM \rightarrow om \mid och \mid (\!|om|\!)^o \mid (\!|om|\!)^m \mid [OP] \mid [\![OP]\!]_p$$

$$OP \rightarrow CM \longmapsto CM$$

$$OP \rightarrow CM \longmapsto (be)\{CM;CM\}$$

$$OP \rightarrow CM \circlearrowleft (be)\{CM\}$$

$$OP \rightarrow CM\textcircled{ρ}CM \mid CM\textcircled{$\vee$}CM \mid CM\textcircled{M}CM \mid CM\textcircled{m}CM \mid CM\textcircled{$\downarrow$}CM$$

We would like to explain some of the concepts presented in Definition 4:

- The variable *CM*, as well as *OP* are two entities very important in the language, as can be seen in the grammar. We name them *compound module* and *operator* respectively.
- The terminals *om* and *och* represent an *operation module* and an *open channel* respectively,
- The terminal *be* is a boolean expression.
- The terminals $[\,], [\![\,]\!]_p$ are symbols for grouping and defining the way of how the involved *compound module*s are executed. Depending on the nature of the operator, they can be executed sequentially or in parallel:

  1. $[OP]$: The involved operator is executed sequentially.
  2. $[\![OP]\!]_p$: The involved operator is executed in parallel if and only if *OP* supports parallelism. Otherwise, an exception is threw.

- The terminals (and) are symbols for grouping the boolean expression in some operators.
- The terminals { and } are symbols for grouping *compound module*s in some operators.
- The terminals $(\!|.|\!)^m, (\!|.|\!)^o$, are operators to send information to other solvers (explained bellow).
- The rest of terminals are POSL operators.

2.3.1 POSL **operators**

In this section we briefly present operators provided by POSL to code the *computation strategy*. A formal presentation of POSL's specification is available in [12].

$\boxed{Op.\ 1:}$ **Operator Sequential Execution:** the operation $M_1 \longmapsto M_2$ represents a *compound module* as result of the execution of M_1 followed by M_2. This operator is an example of an operator that does not support the execution of its involved *compound module*s in parallel, because the input of the second *compound module* is the output of the first one.

$\boxed{Op.\ 2:}$ **Operator Conditional Sequential Execution:** the operation $M_1 \overset{?}{\underset{<cond>}{}} M_2$ represents a *compound module* as result of the sequential execution of M_1 if $< cond >$ is **true** or M_2 otherwise.

$\boxed{Op.\ 3:}$ **Operator Cyclic Execution:** the operation $\circlearrowleft (< cond >) \{I_1\}$ represents a *compound module* as result of the sequential execution of I_1 repeated while $< cond >$ remains **true**.

$\boxed{Op.\ 4:}$ **Operator Random Choice:** the operation $M_1 \overset{\rho}{} M_2$ represents a *compound module* that executes and returns the output of M_1 depending on the probability ρ, or M_2 following $(1 - \rho)$

$\boxed{Op.\ 5:}$ **Operator Not *NULL* Execution:** the operation $M_1 \overset{\vee}{} M_2$ represents a *compound module* that executes M_1 if it is not *NULL* or M_2 otherwise.

$\boxed{Op.\ 6:}$ **Operator MAX:** the operation $M_1 \overset{M}{} M_2$ represents a *compound module* that returns the maximum between the outputs of modules M_1 and M_2 (tacking into account some order criteria).

$\boxed{Op.\ 7:}$ **Operator MIN:** the operation $M_1 \overset{m}{} M_2$ represents a *compound module* that returns the minimum between the outputs of modules M_1 and M_2 (tacking into account some order criteria).

$\boxed{Op.\ 8:}$ **Operator Speed:** the operation $M_1 \overset{\downarrow}{} M_2$ represents a *compound module* that returns the output of the *module* ending first.

In Figure 1 we present a simple example of how to combine *module*s using POSL operators introduced above. Algorithm 1 shows the corresponding code. In this example we show four *operation module*s being part of a *compound module* representing a dummy local search method. In this example:

- M_1: generates a random configuration.
- M_2: computes a neighborhood of a given configuration by selecting a random variable and changing its value.
- M_3: computes a neighborhood of a given configuration by selecting K random variables and changing theirs values.
- M_4: selects, from a set of configurations, the one with the smallest cost, and stores it.

Here, the *operation module* M_2 is executed with probability ρ, and M_3 is executed with probability $(1 - \rho)$. This operation is repeated a number N of times (`< stop_cond >`).
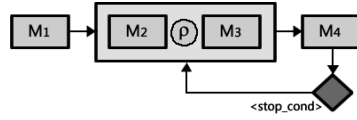

Fig. 1

Algorithm 1: POSL code for Figure 1

$$M_1 \longmapsto [\circlearrowleft (loops < N) \{$$
$$\left[M_2 \textcircled{ρ} M_3 \right] \longmapsto M_4$$
$$\}]$$

In Algorithm 1, *loops* represent the number of iterations performed by the operator.

$\boxed{Op. - 9:}$ **Operator Sending:** allows us to send two types of information to other solvers:

1. The operation $(\!| M |\!)^o$ represents a *compound module* that executes the *compound module M* and sends its output
2. The operation $(\!| M |\!)^m$ represents a *compound module* that executes the *compound module M* and sends *M* itself
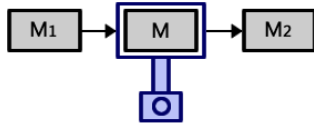

Fig. 2

Algorithm 2: POSL code for Figure 2 case (1.)

$$M_1 \longmapsto (\!| M |\!)^o \longmapsto M_2$$

Algorithm 3: POSL code for Figure 2 case (2.)

$$M_1 \longmapsto (\!| M |\!)^m \longmapsto M_2$$

Algorithms 2 and 3 show POSL's code corresponding to Figure 2 for both cases: a) sending the result of the execution of the *operation module M*, or b) sending the *operation module M* itself.

This operation is very useful in terms of sharing behaviors between solvers. Figure 3 shows another example, where we can combine an *open channel* with the *operation module* M_2 through the operator $\textcircled{$\vee$}$. In this case, the *operation module* M_2 will be executed as long as the *open channel* remains *NULL*, i.e. there is no *operation module* coming from outside. This behavior is represented in Figure 3a by red lines. If some *operation module* has been received by the *open channel*, it is executed instead of the *operation module* M_2, represented in Figure 3b by blue lines.

In this stage, and using these operators, we can create the algorithm managing different components to find the solution of a given problem. These algorithms are fixed, but generic w.r.t. their components (*operation module*s and *open channel*s). It means that we can build different solvers using the same strategy, but instantiating it with different components, as long as they have the right input/output signature.

To define a *computation strategy* we use the environment presented in Algorithm 4, where M_i and Ch_i represent the types of the *operation module*s and the
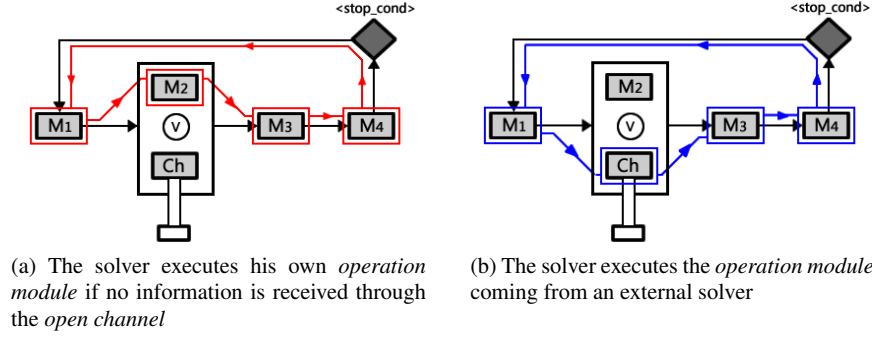
(a) The solver executes his own *operation module* if no information is received through the *open channel*

(b) The solver executes the *operation module* coming from an external solver

Fig. 3: Two different behaviors in the same solver

types of the *open channel*s used by the *computation strategy St*. Between brackets, the field < ...computation strategy... > corresponds to POSL code based on operators combining already declared *module*s.

Algorithm 4: *Computation strategy definition*

$St \leftarrow$ **strategy**
 oModule $M_1, M_2, \ldots, M_n$;
 oChannel $Ch_1, Ch_2, \ldots, Ch_m$;
 {
     < ...computation strategy... >
 }

Algorithm 5: Solver definition

$solver_k \leftarrow$ **solver**
 {
     cStrategy St;
     oModule $m_1, m_2, \ldots, m_n$;
     oChannel $ch_1, ch_2, \ldots, ch_m$;
 }

2.4 Solver definition

With *operation module*s, *open channel*s and *computation strategy* defined, we can create solvers by instantiating the declared components. POSL provides an environment to this end, presented in Algorithm 5, where m_i and ch_i represent the instances of the *operation module*s and the instances of the *open channel*s to be passed by parameters to the *computation strategy St*.

2.5 Communication definition

Once we have defined our solver strategy, the next step is to declare communication channels, i.e. connecting the solvers each others. Up to here, solvers are disconnected, but they have everything to establish the communication. In this last stage, POSL provides to the user a platform to easily define cooperative *meta–strategies* that solvers must follow.

The communication is established by following the next rules guideline:

1. Each time a solver sends any kind of information by using the operator $(\!\mid\!)^o$ or $(\!\mid\!)^m$, it creates a *communication jack*
2. Each time a solver uses an *open channel* into its definition, it creates a *communication outlet*
3. Solvers can be connected each others by creating *subscription*s, connecting *communication jack*s with *communication outlet* (see Figure 4).

With the operator $(\cdot)$ we have access to *operation module*s sending information and to the *open channel*'s names in a solver. For example: $Solver_1 \cdot M_1$ provides access to the *operation module* M_1 in $Solver_1$ if and only if it is affected by the operator $(\!\mid\!)^o$ (or $(\!\mid\!)^m$), and $Solver_2 \cdot Ch_2$ provides access to the *open channel* Ch_2 in $Solver_2$. Tacking this into account, we can define the *subscription*s.

Definition 5. Let two different solvers $Solver_1$ and $Solver_2$ be. Then, we can connect them through the following operation:

$$Solver_1 \cdot M_1 \rightsquigarrow Solver_2 \cdot Ch_2$$

The connection can be defined if and only if:

1. *Solver$_1$* has an *operation module* called M_1 encapsulated into an operator $(\!\mid\!)^o$ or $(\!\mid\!)^m$.
2. *Solver$_2$* has an *open channel* called Ch_2 receiving the same type of information sent by M_1.

Definition 5 only gives the possibility to define static communication strategies. However, our goal is to develop this subject until obtaining operators more expressive in terms of communication between solvers, to allow dynamic modifications of communication strategies, that is, having such strategies adapting themselves during runtime.

3 A POSL **solver**

In this section we explain the structure of a POSL solver created by using the operators-based language provided, to solve some instances of the *Social Golfers* Problem (SGP). It consists to schedule $n = g \times p$ 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 $g - p - w$.

We choose one of the more classic solution methods for combinatorial problems: local search meta-heuristics algorithms. These algorithms have a common structure: they start by initializing some data structures (e.g. a *tabu list* for *Tabu Search* [8], a *temperature* for *Simulated Annealing* [10], etc.). Then, an initial configuration s is generated (either randomly or by using heuristic). After that, a new configuration s^* is selected from the neighborhood $V(s)$. 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 some criterion (e.g. penalizing features of local optimums, like in *Guided Local Search* [2]).

Restarts are classic mechanisms to avoid becoming trapped in local minimum. They are trigged if no improvements are done or by a timeout.

Operation Modules composing each solver of the POSL-solver are described bellow:

1. Generate a configuration s.
2. Define the neighborhood $V(s)$
3. Select $s^* \in V(s)$. In every case for this experiment the selection criteria is to choose the first configuration improving the cost.
4. Evaluate an acceptance criteria for s^*. In every case for this experiment the acceptance criteria is to choose always the configuration with less cost.

For this particular experiment we have created three different solvers (see Figure 4):

1. **Solver 1:** A solver sending the best configuration every K iterations (sender solver). It sends the found configuration to the solver it is connected with. Algorithm 6 shows its computation strategy.
2. **Solver 2:** A solver receiving the configuration coming from a sender solver (**Solver 1**). It takes the received configuration, if its current configuration's cost is not better than the received configuration's cost, and takes a decision. This solver receives the configuration through an *open channel* joined to the *operation module M_3* with the operator (m). Algorithm 7 shows its computation strategy.
3. **Solver 3:** A simple solver without communication at all. This solver does not communicate with any other solver, i.e. it searches the solution into an independent walk though the search space. Algorithm 8 shows its computation strategy.
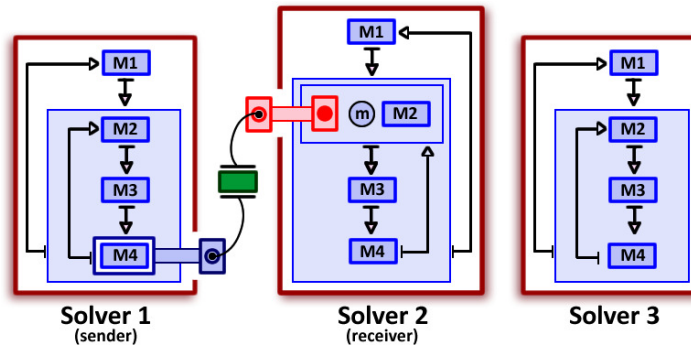

Fig. 4: Three solvers composing the POSL-solver

3.1 Connecting solvers

After the instantiation of each *operation module*, the next step is to connect the solvers (*sender* with *receiver*), by using the proper operator. If one solver Σ_1

(*sender*) sends some information and some other solver Σ_1 (*receiver*) is able to receive it though an *open channel*, then they can be connected as the Algorithm 9 shows.

Algorithm 6: POSL code for **solver 1** in Figure 4

$St_1 \leftarrow$ **strategy**   // ITR $\rightarrow$ number of iterations
oModule : M_1, M_2, M_2, M_4 ;
   {
      $[\circlearrowright (\text{ITR}\%30)\{ M_1 \longmapsto [\circlearrowright (\text{ITR}\%300) \{M_2 \longmapsto M_3 \longmapsto (\!|M_4|\!)^o\}]\}]$
   }

Algorithm 7: POSL code for **solver 2** in Figure 4

$St_2 \leftarrow$ **strategy**   // ITR $\rightarrow$ number of iterations
oModule : M_1, M_2, M_2, M_4 ;
oChannel : Ch_1 ;
   {
      $[\circlearrowright (\text{ITR}\%30)\{ M_1 \longmapsto [\circlearrowright (\text{ITR}\%300)$
      $\{M_2 \longmapsto \left[Ch_1(\!m\!)M_3\right] \longmapsto M_4\}]\}]$
   }

Algorithm 8: POSL code for **solver 3** in Figure 4

$St_3 \leftarrow$ **strategy** ;   // ITR $\rightarrow$ number of iterations
oModule : M_1, M_2, M_2, M_4 ;
   {
      $[\circlearrowright (\text{ITR}\%30)\{ M_1 \longmapsto [\circlearrowright (\text{ITR}\%300) \{M_2 \longmapsto M_3 \longmapsto M_4\}]\}];$
   }

Algorithm 9: Inter–solvers communication definition

$\Sigma_1 \cdot M_4 \rightsquigarrow \Sigma_2 \cdot Ch_1$

---

4 Results

We ran experiments to study the behavior of POSL's solvers in different scenarios solving instances of the Social Golfers Problem. For that reason we classified runs taking into account the composition of POSL[1] solvers:

- Without communication: we use a set of **solvers 3** without communication.
- Some communicating solvers: some of the solvers are **solvers 3** without communication, the others are couples of connected solvers (**solver 1** and **solver 2**)
- All communicating solvers: we use a set of couples of connected solvers (**solver 1** and **solver 2**)

---

[1] POSL source code is available in https://github.com/alejandro-reyesamaro/POSL

Our first experiment uses our desktop computer (Intel® Core™ i7 (2.20GHz) with 16 Gb RAM), for solving instances of SGP with 1 (sequential), 4 and 8 cores. Results can be found in Table 1. In this table, as well as in Table 2, **C** are the numbers of used cores, **T** indicates the runtime in milliseconds, and **It.** the number of iterations. Values are the mean of 25 runs for each setup.

The other set of runs were performed on the server of our laboratory (Intel® Xeon™ E5-2680 v2 (10×4 cores, 2.80GHz)). Table 2 shows obtained results.

Inst	C	No Comm.		50% Comm.		All Comm.	
		T	It.	T	It.	T	It.
6-6-3	1	6,089	159	-	-	-	-
	4	1,500	109	**1,354**	**97**	3,512	181
	8	**1,980**	83	2,049	**78**	5,323	113
7-7-3	1	17,243	831	-	-	-	-
	4	6,082	208	**5,850**	**170**	13,094	270
	8	6,125	136	**5,975**	**124**	13,864	219
8-8-3	1	32,042	428	-	-	-	-
	4	23,358	270	**22,512**	**222**	56,740	340
	8	**19,309**	126	19,925	**121**	28,036	144
9-9-3	1	198,450	1,516	-	-	-	-
	4	94,867	662	**91,556**	**517**	102,974	596
	8	102,629	394	**98,060**	**335**	126,799	466

Table 1: Intel Core i7

Inst	C	No Comm.		15% Comm.		25% Comm.		30% Comm.		50% Comm.		75% Comm.		All Comm.	
		T	It.	T	It.	T	It.	T	It.	T	It.	T	It.	T	It.
6-6-3	1	2,684	229	-	-	-	-	-	-	-	-	-	-	-	
	10	1,810	131	1,636	107	1,479	99	1,634	107	**1,406**	**79**	1,532	91	3,410	182
	20	1,199	82	1,094	75	964	**70**	1,096	76	1,124	78	1,299	87	1,769	101
	30	1,214	75	1,092	64	1,010	68	1,101	68	**766**	**52**	1,366	85	1,984	73
	40	**1,043**	50	1,063	**49**	1,104	54	1,299	58	1,186	**49**	1,462	63	1,824	69
7-7-3	1	11,070	533	-	-	-	-	-	-	-	-	-	-	-	
	10	6,636	245	5,992	189	**5,139**	179	5,456	**177**	6,055	205	6,398	197	8,450	221
	20	2,734	104	2,880	102	**2,517**	**90**	3,028	111	2,970	111	3,465	124	4,153	143
	30	3,141	100	2,864	91	**1,972**	**69**	2,312	79	2,907	97	3,028	82	3,236	89
	40	2,615	68	2,810	70	**2,111**	**55**	2,984	71	2,981	74	3,636	79	3,934	86
8-8-3	1	24,829	315	-	-	-	-	-	-	-	-	-	-	-	
	10	17,652	193	17,067	168	**16,008**	163	16,167	161	16,624	**147**	21,244	185	27,248	226
	20	8,430	102	8,218	92	**6,197**	**77**	7,950	93	7,962	92	8,550	91	12,958	125
	30	7,424	81	6,439	66	**6,268**	**71**	7,413	80	7,407	75	9,806	89	10,420	90
	40	9,700	75	10,068	76	9,377	72	**8,983**	**68**	9,360	72	11,805	84	12,859	91
9-9-3	1	190,965	1,315	-	-	-	-	-	-	-	-	-	-	-	
	10	47,300	331	45,946	293	**43,682**	**276**	45,433	286	47,820	327	67,113	439	79,938	506
	20	28,193	200	25,370	178	24,936	**161**	**24,786**	169	28,369	194	30,147	203	33,610	232
	30	22,035	**123**	21,792	127	**19,518**	125	23,426	133	25,989	163	31,904	172	32,982	203
	40	27,669	125	26,030	116	**24,196**	**112**	28,284	125	26,405	118	32,464	149	34,316	140

Table 2: Intel Xeon

Results show how the parallel multi–walk strategy increases the probability of finding the solution within a reasonable time, when compared to the sequential scheme. Thanks to POSL it was possible to test different solution strategies easily and quickly. With the *Intel Xeon* server we were able to test seven strategies,

and with the desktop machine only 3, due to the limitation in the number of cores. Results suggest that strategies where there exist a lot of communication between solvers (sending or receiving information) are not good (sometimes is even worse than sequential). That is not only because their runtimes are higher, but also due to the fact that only a low percentage of the receivers solvers were able to find the solution before the others did. This result is not surprising, because inter–process communications imply overheads in the computation process, even with asynchronous communications. This phenomenon can be seen in Figure 5a, where it is analyzed the percentage mentioned above versus the numbers of running solvers.

(a) *Communication rate*: % of solutions found by communicating solvers.

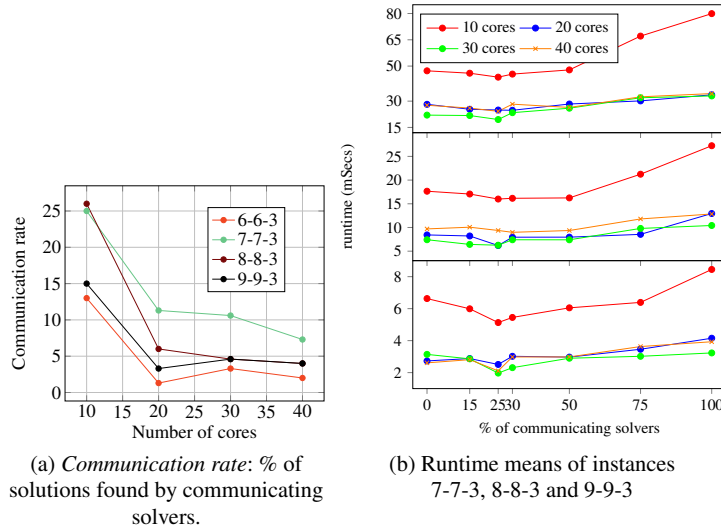(b) Runtime means of instances 7-7-3, 8-8-3 and 9-9-3

Fig. 5

When we face the problem of building a parallel strategy, it is necessary to find an equilibrium between the numbers of communicating solvers and the number of independent solvers. Indeed the communication cost is not negligible: it implies data reception, information interpretation, making decisions, etc.

Slightly better results were obtained with the strategy *25% Comm* when compared to those obtained with the rest, suggesting that the solvers cooperation can be a good strategy. In general, the results obtained using any of the afore mentioned strategies were significantly better than when using the *All Comm* strategy. Figure 5b shows for each instance, the runtime means using different numbers of cores.

The fact we send the best configuration found to other solvers has an impact on communication evaluations. If the percentage of communicating solvers is high and the communication manage to be effective, i.e. the receiver solver accepts the configuration for the next iteration, then we are losing a bit the *independent multi-walk* effect in our solver, that is, most of the solvers are looking for a solution in

the same search space area. However, this is not a problem: if a solver is trapped, a restart is performed. Determining what information to share and to not share among solvers has been few investigated and deserves a deep study.

In many cases, using all cores available did not improves the results. This phenomenon can be observed clearly in runs with communication, and one explanation can be the resulting overhead, which is way bigger. Another reason why we obtain these results can be the characteristic of the architecture, that is, in many cases, not uniform in terms of reachability between cores [1]. We can observe that, even if runtimes are not following a strict decreasing pattern when the number of cores increases, iterations do, suggesting once again that the parallel approach is effective.

With communications, the larger the problem, the more likely effective cooperations between processors are, although sometimes a decreasing pattern occurs while approaching the maximum number of cores, due to communication overheads and architecture limitations.

Before we perform these experiments, we compared runtimes between two solvers: one using an *operation module* to select a configuration from a computed neighborhood that selects the *first* configuration improving the current configuration's cost, and other selecting the *best* configuration among all configurations in the neighborhood. Smallest runtimes were obtained by the one selecting the *first* best configuration, and that is way we used this *operation module* in our experiments. It explains the fact that some solvers need more time to perform less iterations.

5 Conclusions

In this chapter we have presented POSL, a framework for building cooperating solvers. It provides an effective way to build solvers which exchange any kind of information, including other solver's behavior, sharing their *operation module*s. Using POSL, many different solvers can be created and ran in parallel, using only one generic strategy, but instantiating different *operation module*s and *open channel*s for each of them.

It is possible to implement different communication strategies, since POSL provides a layer to define *communication channels* connecting solvers dynamically using *subscription*s.

At this point, the implementation of POSL remains in progress, in which our principal task is creating a design as general as possible, allowing to add new features. Our goal is obtaining a rich library of *operation module*s and *open channel*s to be used by the user, based on a deep study of the classical meta-heuristics algorithms for solving combinatorial problems, in order to cover them as much as possible. In such a way, building new algorithms by using POSL will be easier.

At the same time we pretend to develop new operators, depending on the new needs and requirements. It is necessary, for example, to improve the *solver definition* language, allowing the process to build sets of many new solvers to be faster and easier. Furthermore, we are aiming to expand the communication definition language, in order to create versatile and more complex communication strategies, useful to study the solvers behavior.

As a medium term future work, we plan to include machine learning techniques, to allow solvers to change automatically, depending for instance on results of their neighbor solvers.

References

1. Blake, G., Dreslinski, R.G., Mudge, T.: A survey of multicore processors. Signal Processing Magazine **26**(6), 26–37 (2009)
2. Boussaïd, I., Lepagnot, J., Siarry, P.: A survey on optimization metaheuristics. Information Sciences **237**, 82–117 (2013)
3. Brownlee, A.E., Swan, J., Özcan, E., Parkes, A.J.: 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, pp. 1133–1140. ACM, Vancouver, BC (2014)
4. Cahon, S., Melab, N., Talbi, E.G.: ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. Journal of Heuristics **10**(3), 357–380 (2004)
5. Diaz, D., Richoux, F., Codognet, P., Caniou, Y., Abreu, S.: Constraint-Based Local Search for the Costas Array Problem. In: Learning and Intelligent Optimization, pp. 378–383. Springer Berlin Heidelberg (2012)
6. Frank, S., Hofstedt, P., Mai., P.R.: Meta-S: A Strategy-Oriented Meta-Solver Framework. In: Florida AI Research Society (FLAIRS) Conference, pp. 177–181 (2003)
7. Fukunaga, A.S.: Automated discovery of local search heuristics for satisfiability testing. Evolutionary computation **16**(1), 31–61 (2008)
8. Gendreau, M., Potvin, J.Y.: Tabu Search. In: M. Gendreau, J.Y. Potvin (eds.) Handbook of Metaheuristics, vol. 146, 2nd edn., chap. 2, pp. 41–59. Springer (2010)
9. Munera, D., Diaz, D., Abreu, S., Codognet, P.: A Parametric Framework for Cooperative Parallel Local Search. In: C. Blum, G. Ochoa (eds.) Evolutionary Computation in Combinatorial Optimisation, *LNCS*, vol. 8600, pp. 13–24. Springer Berlin Heidelberg, Granada (2014)
10. Nikolaev, A.G., Jacobson, S.H.: Simulated Annealing. In: M. Gendreau, J.Y. Potvin (eds.) Handbook of Metaheuristics, vol. 146, 2nd edn., chap. 1, pp. 1–39. Springer (2010)
11. Pajot, B., Monfroy, E.: Separating Search and Strategy in Solver Cooperations. In: Perspectives of System Informatics, pp. 401–414. Springer Berlin Heidelberg (2003)
12. Reyes-Amaro, A., Monfroy, E., Richoux, F.: A Parallel-Oriented Language for Modeling Constraint-Based Solvers. In: Proceedings of the 11th edition of the Metaheuristics International Conference (MIC 2015). Springer (2015)
13. Swan, J., Burles, N.: Templar - a framework for template-method hyper-heuristics. In: P. Machado, M.I. Heywood, J. McDermott, M. Castelli, P. García-Sánchez, P. Burelli, S. Risi, K. Sim (eds.) Genetic Programming, *LNCS*, vol. 9025, pp. 205–216. Springer International Publishing (2015)
14. Talbi, E.G.: Combining metaheuristics with mathematical programming, constraint programming and machine learning. 4or **11**(2), 101–150 (2013)