POSL: Un langage orienté parallèle pour modéliser des solveurs de contraintes

Alejandro Reyes Amaro
LINA - UMR 6241, TASC - INRIA Université de Nantes, France.
alejandro.reyes@univ-nantes.fr

Résumé

La technologie multi-cœur et les architectures massivement parallèles sont de plus en plus accessibles à tous, à travers des matériaux comme le Xeon Phi ou les cartes GPU. Cette stratégie d'architecture a été communément adoptée par les producteurs pour faire face à la loi de Moore. Or, ces nouvelles architectures impliquent d'autres manières de concevoir et d'implémenter les algorithmes, pour exploiter complètement leur potentiel, en particulier dans le cas des solveurs de contraintes traitant de problèmes d'optimisation combinatoire.

Cette thèse présente Parallel-Oriented Solver Language (POSL, prononcé *"puzzle"*) : un système pour construire des méta-heuristiques interconnectées travaillant en parallèle. Le but de ce travail est d'obtenir un système pour facilement construire des solveurs et réduire l'effort de leur développement en proposant un mécanisme de réutilisation de code entre les différents solveurs. La nouveauté de cette approche porte sur le fait que l'on voit un solveur comme un ensemble de composants spécifiques, écris dans un langage orienté parallèle basé sur des opérateurs. POSL permets aux composants d'un solveur d'être transmis et exécutés par d'autres solveurs. Il propose également une couche supplémentaire permettant de définir des connexions entre solveurs.

1 Introduction

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

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

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

Grâce à ces développements, les algorithmes parallèles ont ouvert de nouvelles façons de résoudre les problèmes de contraintes : Adaptive Search [2] est un algorithme efficace, montrant de très bonnes performances et passant à l'échelle de plusieurs centaines ou même milliers de cœurs, en utilisant la recherche locale *multi-walk* en parallèle. Munera et al. [7] ont présenté une autre implémentation d'Adaptive Search en utilisant la coopération entre des stratégies de recherche. *Meta–S* est une implémentation d'un cadre théorique présenté dans [3], qui permet d'attaquer les problèmes par la coopération de solveurs de contraintes de domaine spécifique. Ces travaux ont montré l'efficacité du schéma parallèle multi-walk.

De plus, le temps de développement nécessaire pour coder des solveurs en parallèle est souvent sous-estimé, et dessiner des algorithmes efficaces pour résoudre certains problèmes consomment trop de temps. Dans cette thèse nous présentons POSL, un langage orienté parallèle pour construire des solveurs de contraintes basés sur des méta-heuristiques, qui résolvent des *CSP*s. Il fournit un mécanisme pour coder des stratégies de communication indépendantes du Le but de cet article est de proposer des nouveaux opérateurs de communication, très utiles pour dessiner des stratégies de communication, et de présenter une analyse détaillée des résultats obtenus en résolvant plusieurs instances des problèmes CSP. Sachant que créer des solveurs utilisant différentes stratégies de solution peut être compliqué et pénible, POSL donne la possibilité de faire des prototypes de solveurs communicants facilement.

2 Des travaux reliés

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

HYPERION [1] est un système codé en Java pour méta et hyper-heuristiques basé sur le principe d'interopérabilité, fournissant des patrons génériques pour une variété d'algorithmes de recherche locale et évolutionnaire, et permettant des prototypages rapides avec la possibilité de réutiliser le code source. POSL offre ces avantages, mais il fournit également un mécanisme permettant de définir des protocoles de communication entre solveurs. Il fournit aussi, à travers d'un simple langage basé sur des opérateurs, un moyen de construire des abstract solvers, en combinant des modules déjà définis (computation modules et communication modules). Une idée similaire a été proposée dans [4] sans communication, qui introduit une approche évolutive en utilisant une simple composition d'opérateurs pour découvrir automatiquement les nouvelles heuristiques de recherche locale pour SAT et les visualiser comme des combinaisons d'un ensemble de blocs.

Récemment, [9] a montré l'efficacité de combiner différentes techniques pour résoudre un problème donné (hybridation). Pour cette raison, lorsque les composant du solveurs peuvent être combinés, POSL est dessiné pour exécuter en parallèle des ensembles de solveurs différents, avec ou sans communication. Une autre idée intéressante est proposée dans TEMPLAR. Il s'agit d'un système qui génère des algorithmes en changeant des composants prédéfinis, et en utilisant des méthodes hyper-heuristiques [8]. Dans la dernière phase du processus de codage avec POSL, les solveurs peuvent être connectés les uns aux autres, en fonction de la structure de leurs communication modules, et de cette façon, ils peuvent partager non seulement des informations, mais aussi leur comportement, en partageant leurs computation modules. Cette approche donne aux solveurs la capacité d'évoluer au cours de l'exécution.

Renaud De Landtsheer et al. présentent dans [5] un cadre facilitant le développement des systèmes de recherches en utilisant des *combinators* pour dessiner les caractéristiques trouvées très souvent dans les procédures de recherches comme des briques, et les assembler. Dans [6] est proposée une approche qui utilise des systèmes coopératifs de recherche locale basée sur des méta-heuristiques. Celle-ci se sert de protocoles de transfert de messages. POSL combine ces deux idées pour assembler des composants de recherche locale à travers des opérateurs fournis (ou en créant des nouveaux), mais il fournit aussi un mécanisme basé sur opérateurs pour connecter et combiner des solveurs, en créant des stratégies de communication.

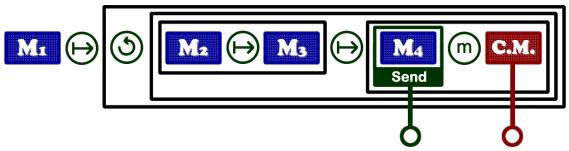
Dans cette thèse, nous présentons quelques nouveaux opérateurs de communication afin de concevoir des stratégies de communication. Avant de clore cet article par une brève conclusion et de travaux futurs, nous présentons quelques résultats obtenus en utilisant POSL pour résoudre certaines instances des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

3 Solveurs parallèles POSL

POSL permet de construire des solveurs suivant différentes étapes :

1. L'algorithme du solveur considéré est exprimé via une décomposition en modules de calcul. Ces modules sont implémentés à la manière de *fonctions* séparées. Nous appelons *computation module* ces morceaux de calcul (figure 1a, blocs bleus). En suite, il faut décider quelles sont les types d'informations que l'on souhaite recevoir des autres solveurs. Ces informations sont encapsulées dans des composants appelés communication module, permettant de transmettre des données entre solveurs (figure 1a, bloc rouge)
2. Une *stratégie générique* est codée à travers POSL, en utilisant les opérateurs fournis par le langage appliqués sur des modules *abstraite* qui représentent les *signatures* des composants donnés lors l'étape 1, pour créer abstract solvers. Cette stratégie définie non seulement les informations échangées, mais détermine également l'exécution paral-

(a) Definition du modules et l'communication modules



(b) Definition de l'abstract solver

(c) Definition de la strategie de communication

FIGURE 1 – Construire des solveurs parallèles avec POSL

lèle de composants. Lors de cette étape, les informations à partager sont transmises via les opérateurs ad-hoc. On peut voir cette étape comme la définition de la colonne vertébrale des solveurs.

3. Les solveurs sont créés en instanciant l'abstract solver, par computation modules et communication module (figure 1b).
4. Les solveurs sont assemblés en utilisant les opérateurs de communication fournis par le langage, pour creér des strategies de communication. Cet entité final s'appelle *solver set*(figure 1c).

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

3.1 Computation module

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

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

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

$$\mathcal{C}m : \mathcal{I} \rightarrow \mathcal{O} \tag{1}$$

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

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

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

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

3.2 Communication module

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

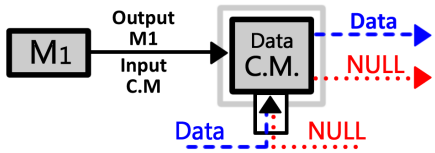
Un communication module peut recevoir deux types d'informations, provenant toujours d'un solveur tiers : des données et des computation modules. En ce qui concerne les computation modules, leur communication peut se faire via la transmission d'identifiants permettant à chaque solveur de les instancier.

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

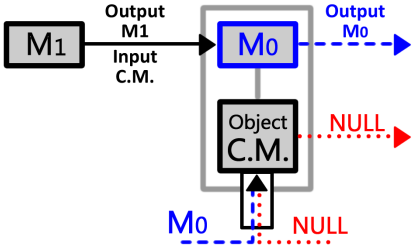
Definition 2 ***(Data communication module)*** *Un data communication module $\mathcal{C}h$ est un composant produisant une application définie comme suit :*

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

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



(a) Data communication module



(b) Object communication module

FIGURE 2 – Mécanisme interne du communication module

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

$$\mathcal{C}h : I \times \{\mathbf{M} \cup \{NULL\}\} \rightarrow O \cup \{NULL\} \qquad (4)$$

Puisque les communication modules reçoivent des informations provenant d'autres solveurs sans pour autant avoir de contrôle sur celles-ci, il est nécessaire de définir l'information *NULL*, signifiant l'absence d'information. La figure 2 montre le mécanisme interne d'un communication module. Si un data communication module reçoit une information, celle-ci est automatiquement retournée (figure 2a, lignes bleues). Si un object communication module reçoit un computation module, ce dernier est instancié et exécuté avec l'entrée de l'communication module, et le résultat est retourné (figure 2b, lignes bleues). Dans les deux cas, si aucune information n'est reçue, l'communication module retourne l'objet *NULL* (figure 2, lignes rouges).

3.3 Abstract solver

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

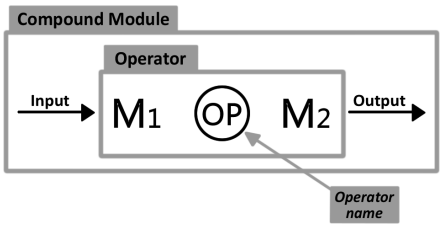


FIGURE 3 – Un compound module

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

1. *un computation module ; ou*
2. *un communication module ; ou*
3. $[OP\ \mathcal{M}]$*, la composition d'un module* $\mathcal{M}$ *exécuté sequentielement, en retournant une sortie, en dépendant de la nature de l'opérateur unaire* OP *; ou*
4. $[\mathcal{M}_1\ OP\ \mathcal{M}_2]$*, la composition de deux modules* $\mathcal{M}_1$ *et* $\mathcal{M}_2$ *exécuté séquentiellement, en retournant une sortie, en dépendant de la nature de l'opérateur binaire* OP*.*
5. $[\mathcal{M}_1\ OP\ \mathcal{M}_2]$*, la composition de deux modules* $\mathcal{M}_1$ *et* $\mathcal{M}_2$ *exécuté, en retournant une sortie, en dépendant de la nature de l'opérateur binaire* OP*. Ces deux opérateurs vont être exécutes en parallèle si et seulement si* OP *support le parallélisme, ou il lance une exception en cas contraire.*

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

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

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

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

L'opérateur suivant nous permet d'exécuter deux modules séquentiellement, l'un après l'autre.

Definition 5 (Sequential Execution Operator) *Soient i)* $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *et ii)* $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$*. deux modules différents. Alors l'opération* $|\mathcal{M}_1 \ (\mapsto)\ \mathcal{M}_2|$ *définit le compound module* $\mathcal{M}_{seq}$ *comme le résultat de l'exécution de* $\mathcal{M}_1$ *suivi de* $\mathcal{M}_2$*.*

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

L'opérateur présenté dans la définition 5 est un exemple d'opérateur ne supportant pas une exécution parallèle de ses compound modulesimpliqués, puisque

l'entrée du second compound module est la sortie du premier.

L'opérateur suivant est utile pour exécuter des modules séquentiels créant des branchements de calcul selon une condition booléenne :

Definition 6 (Conditional Execution Operator) *Soient i)* $\mathcal{M}_1 : \mathcal{D} \to \mathcal{I}_1$ *et ii)* $\mathcal{M}_2 : \mathcal{D} \to \mathcal{I}_2$*, deux modules différents. Alors l'opération* $\left|\mathcal{M}_1 \textcircled{?}_{<cond>} \mathcal{M}_2\right|$ *définit le compound module* $\mathcal{M}_{cond}$ *le résultat de l'exécution en séquentiel de* $\mathcal{M}_1$ *si* $< cond >$ *es* **vrai** *or* $\mathcal{M}_2$*, autrement :*

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

Nous pouvons exécuter séquentiellement des modules créant des boucles de calcul, en définissant les compound modules avec un autre opérateur conditionnel :

Definition 7 (Cyclic Execution Operator) *Soit* $\mathcal{M} : \mathcal{D} \to \mathcal{I}$ *un module, où* $\mathcal{I} \subseteq \mathcal{D}$*. Alors, l'opération* $|\circlearrowleft_{<cond>} \mathcal{M}|$ *définit le compound module* $\mathcal{M}_{cyc}$ *en répétant séquentiellement l'exécution de* $\mathcal{M}$ *tant que* $< cond >$ *est* **vrai** *:*

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

POSL offre la possibilité de faire muter les solveurs. En fonction de l'opération, un ou plusieurs module opérande(s) sera exécutée(s), mais seule la sortie de l'un d'entre eux sera retournée par le compound module. Nous présentons ces opérateurs dans deux définitions, groupant ceux qui exécutent uniquement un opérande de module (définition 8 et 9) et ceux exécutant les deux opérandes (définition 10, 11 et 12).

Definition 8 Random Choice Operator *Soient i)* $\mathcal{M}_1 : \mathcal{D} \to \mathcal{I}_1$ *et ii)* $\mathcal{M}_2 : \mathcal{D} \to \mathcal{I}_2$*, deux modules différents, et un numéro réel* $\rho \in (0, 1)$*. Alors, l'operation* $\left|M_1 \textcircled{\rho} \mathcal{M}_2\right|$ *définit le compound module* $\mathcal{M}_{rho}$ *qui exécute* $\mathcal{M}_1$ *en suivant une probabilité* ρ*, ou en exécutant* $\mathcal{M}_2$ *en suivant une probabilité* $(1 - \rho)$ *:*

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

Definition 9 Not *NULL* **Execution Operator** *Soient i)* $\mathcal{M}_1 : \mathcal{D} \to \mathcal{I}_1$ *et ii)* $\mathcal{M}_2 : \mathcal{D} \to \mathcal{I}_2$*, deux modules différents. Alors, l'operation* $\left|\mathcal{M}_1 \textcircled{\vee} \mathcal{M}_2\right|$ *définit le compound module* $\mathcal{M}_{non}$ *qui exécute* $\mathcal{M}_1$ *et retourne une sortie si elle n'est pasNULL, ou exécute* $\mathcal{M}_2$ *et retourne une sortie autrement :*

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

La définition suivante fait appelle aux notions de *parallélisme coopératif* et de *parallélisme compétitif*. Nous disons qu'il y a parallélisme coopératif quand deux unités de calcul ou plus s'exécutent simultanément, et que le résultat obtenu provient de la combinaison des résultats calculés par chaque unité de calcul (voir définitions 10 et 11). À l'opposé, nous considérons qu'il y a parallélisme compétitif lorsque le résultat obtenu est une solution ne provenant que d'un seul processus exécuté en parallèle ; en général le premier processus à terminer (voir définition 12).

Definition 10 Minimum Operator *Soient*

1. $\mathcal{M}_1 : \mathcal{D} \to \mathcal{I}_1$ *et*
2. $\mathcal{M}_2 : \mathcal{D} \to \mathcal{I}_2$,

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

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

De la même manière nous définissons l'opérateur **Maximum** :

Definition 11 Minimum Operator *Soient*

1. $\mathcal{M}_1 : \mathcal{D} \to \mathcal{I}_1$ *et*
2. $\mathcal{M}_2 : \mathcal{D} \to \mathcal{I}_2$,

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

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

Definition 12 Race Operator *Soient i)* $\mathcal{M}_1 : \mathcal{D} \to \mathcal{I}_1$ *et ii)* $\mathcal{M}_2 : \mathcal{D} \to \mathcal{I}_2$*, deux modules différents, où* $\mathcal{D}_1 \subseteq \mathcal{D}_2$ *et* $\mathcal{I}_1 \subset \mathcal{I}_2$*. Alors, l'opération* $\left|\mathcal{M}_1 \textcircled{\downarrow} \mathcal{M}_2\right|$ *définit le compound module* $\mathcal{M}_{race}$ *qui exécute les deux modules* $\mathcal{M}_1$ *et* $\mathcal{M}_2$*, et retourne la sortie du module qui termine en premier :*

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

Les opérateurs introduits par les définitions 8, 9 et 10 sont très utiles en terme de partage d'informations entre solveurs, mais également en terme de partage de comportements. Si un des opérandes est un communication module alors l'opérateur peut recevoir le

computation module d'un autre solveur, donnant la possibilité d'instancier ce module dans le solveur le réceptionnant. L'opérateur va soit instancier le module s'il n'est pas *NULL* et l'exécuter, soit exécuter le module donné par le second opérande.

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

Definition 13 Sending Data Operator *Soit* $\mathcal{M} : \mathcal{D} \rightarrow \mathcal{I}$ *un module. Alors, l'opération* $|(\!|\mathcal{M}|\!)^d|$ *définit le compound module* $\mathcal{M}_{sendD}$ *qui exécute le module* $\mathcal{M}$ *puis envoie la sortie vers un communication module :*

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

Definition 14 Sending Module Operator *Soit* $\mathcal{M} : \mathcal{D} \rightarrow \mathcal{I}$ *un module. Alors, l'opération* $|(\!|\mathcal{M}|\!)^m|$ *définit le compound module* $\mathcal{M}_{sendM}$ *qui exécute le module* $\mathcal{M}$*, puis envoie le module lui même vers un communication module :*

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

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

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

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

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

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

3.4 Créer les solveurs

Maintenant on peut créer les solveurs en instanciant les modules. Il est possible de faire ceci en spécifiant

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

abstract solver *as_01*
computation *:* I, V, S, A
connection *:* $C.M.$
begin
$I \circlearrowright \mapsto$
$[\circlearrowleft$ (ITR $< K_1$)
$\left[V \circlearrowright \mapsto S \circlearrowright \mapsto \left[C.M. \circlearrowright m \; (\!|A|\!)^d\right]\right]$
]
end

que un **solver** donné doit implémenter (en utilisant le mot clé **implements**) un abstract solver donné, suivi par la liste de computation puis communication modules. Ces modules doivent correspondre avec les signatures exigé par l'abstract solver.

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

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

3.5 Connecter les solveurs : créer le solver set

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

Les communications sont établies en respectant les règles suivantes :

1. À chaque fois qu'un solveur envoie une information via les opérateurs $(\!|.|\!)^d$ ou $(\!|.|\!)^m$, il créé une *prise mâle de communication*
2. À chaque fois qu'un solveur contient un communication module, il créé une *prise femelle de communication*
3. Les solveurs peuvent être connectés entre eux en reliant *prises mâles* et *femelles*.

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

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

Definition 15 Connection One-to-One Operator *Soient*

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

Alors, l'opération

$$\mathcal{J} \ (\rightarrow) \ \mathcal{O}$$

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

Definition 16 Connection One-to-N Operator *Soient*

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

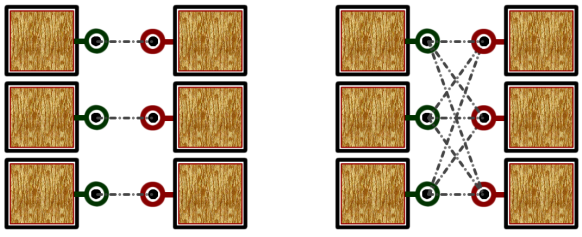
Alors, l'opération

$$\mathcal{J} \ (\rightsquigarrow) \ \mathcal{O}$$

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

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

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



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

FIGURE 4 – Représentation graphique des opérateurs de communication

4 Les résultats

Le but principal de cette section est de sélectionner quelques instances de problèmes de référence, pour analyser et illustrer la versatilité de POSL pour étudier des stratégies de solution basées sur la recherche locale méta-heuristique avec communication. Grâce à POSL nous pouvons analyser des résultats et formuler des conclusions sur le comportement de la stratégie de recherche, mais aussi sur la structure de l'espace de recherche du problème. Dans cette section, nous expliquons la structure des solveurs de POSL que nous avons générés pour les expériences, et les résultats.

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

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

4.1 Social Golfers Problem

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

Nous avons utilisé uns stratégie de communication cyclique pour résoudre ce problème, en échangeant la

configuration courante entre deux solveurs avec des caractéristiques différentes. Les résultats montrent que cette stratégie marche très bien pour ce problème.

Algorithm 3: Solveur pour *SGP*

```
abstract solver as_eager  // Itr → nombre d'itérations
computation : I, V, S1, S2, A
    // Sci → nombre d'itérations avec le même coût
begin
    I (↦)
    [↻ (Itr < K1)
        V (↦) [S1 (?)Sci%K2 S2] (↦) A
    ]
end
solver Solver_eager implements as_eager
    computation : I_BP, V_BAS, S_first, S_rand, A_AI
```

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

1. S_{BP} génère une configuration aléatoire s, en respectant la structure du problème, c'est-à-dire que la configuration est un ensemble de w permutations du vecteur $[1..n]$.
2. V_{BAS} définit le voisinage $V(s)$ permutant le joueur qui a contribué le plus au coût, avec d'autres joueurs dans la même semaine.
3. S_{rirst} sélectionne la première configuration $s' \in V(s)$ qui améliore le coût actuel, et retourne (s, s')
4. S_{rand} sélectionne une configuration aléatoire $s' \in V(s)$, et retourne (s, s')
5. A_{AI} retourne toujours la configuration sélectionnée (s').

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

Algorithm 4: Solveur standard pour *SGP*

```
abstract solver as_standard
computation : I, V, S1, S2, A
communication : C.M.
begin
    I (↦)
    [↻ (Itr < K1)
        V (↦) [S1 (?)Sci%K2 S2] (↦) [C.M. (m) ⦅A⦆^d]
    ]
end
solver Solver_standard implements as_standard
    computation : I_BP, V_BAS, S_first, S_rand, A_AI
    communication : CM_last
```

Algorithm 5: Solveur compagnon pour *SGP*

```
abstract solver as_compagnon
computation : I, V, S1, S2, A
communication : C.M.
begin
    I (↦)
    [↻ (Itr < K1)
        V (↦) [S1 (?)Sci%K2 S2] (↦) [C.M. (∨) ⦅A⦆^d]
    ]
end
solver Solver_compagnonandard implements as_compagnon
    computation : I_BP, V_BP(p), S_first, S_rand, A_AI
    communication : CM_last
```

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

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

4.2 Costas Array Problem

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

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

TABLE 1 – Résultats pour *SGP*

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

TABLE 2 – Résultats pour *CAP* 19

de l'ordre 17 le nombre de solutions diminue drastiquement.

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

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

Pour résoudre *CAP* nous avons eu besoin d'utiliser un computation module de *reset* (T_{AS}) comme machinisme d'exploration. L'algorithme 6 montre le solveur utilisé pour résoudre ce problème séquentiellement. Les résultats des runs en séquentiel et en parallèle sans communication sont montrés dans le tableau 2. Ils montrent le succès de l'approche en parallèle est montré encore une fois. Afin d'améliorer ces résultats, nous avons appliqué uns stratégies simple de communication : communiquer la configuration courante au moment d'exécuter le critère d'acceptation. Les algorithmes 7 et **??** montrent les solveurs envoyeur et récepteur.

Algorithm 6: Solveur pour *CAP*

```
abstract solver as_hard
computation : I, T, V, S, A
begin
    I (↦)
    [↻ (Itr < K1)
        T (↦) [↻ (Itr % K2) [V (↦) S (↦) A] ]
    ]
end
solver Solver1 implements as_hard
    computation : Iperm, TAS, VAS, Sfirst, AAI
```

Un des buts principaux de cette étude a été d'explorer des différentes stratégies de communication. Nous avons ensuite mis en place et testé différentes variantes de la stratégie exposée ci-dessus en combinant deux opérateurs de communication (one to one et one to N)

Algorithm 7: Solveur envoyeur pour *CAP*

```
abstract solver as_hard_sen
computation : I, T, V, S, A
begin
    I (↦)
    [↻ (Itr < K1)
        T (↦) [↻ (Itr % K2) [V (↦) S (↦) (|A|)^d] ]
    ]
end
solver Solver_sender implements as_hard_sen
    computation : Iperm, TAS, VAS, Sfirst, AAI
```

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

4.3 Golomb Ruler Problem

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

Algorithm 8: Solveur récepteur pour *CAP*

```
abstract solver as_hard_rec
computation : I, T, V, S, A
communication : C.M.
begin
  I (↦)
  [↻ (Itr < K1)
    T (↦) [↻ (Itr % K2) [V (↦) S (↦) [A (m) C.M.]] ]
  ]
end
solver Solver_receiver implements as_hard_rec
    computation : Iperm, TAS, VAS, Sfirst, AAI
    communication : CMlast
```

munication intéressante : communiquer la configuration actuelle afin d'éviter son voisinage, c'est à dire, une configuration *tabu*.

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

1. I_{sort} : renvoie une configuration aléatoire s en tant que vecteur d'entiers trié. La configuration est générée *loin* de l'ensemble des configurations *tabu* arrivés via communication entre solveurs.
2. V_{sort} : donné une configuration, retourne le voisinage en changeant une valeur tout en gardant l'ordre, à savoir, le remplacement de la valeur s_i par toutes les valeurs possibles $s'_i \in D_i$ en satisfaisant $s_{i-1} < s'_i < s_{i+1}$.

Nous avons également ajouté un module de reset T : il reçoit et renvoie une configuration. Le computation module utilisé pour l'instancier (T_{tabu}) insère la configuration reçue dans une liste *tabu* à l'intérieur du solveur et retourne la configuration d'entrée tal cual. L'algorithme 9 présente le solveur utilisé pour envoyer des informations (solveur envoyeur).

Algorithm 9: Solveur envoyeur pour *GRP*

```
abstract solver as_golomb_sender
computation : I, V, S, A, T
begin
    [↻ (Itr < K1)
        I ↦ [↻ (Itr % K2) [V ↦ S ↦ A] ] ↦ ⟪T⟫^d
    ]
end
solver Solver_sender implements as_golomb_sender
    computation : I_sort, V_sort, S_first, A_AI, , T_tabu
```

Le module T_{tabu} est exécuté lorsque le solveur est incapable de trouver une meilleure configuration autour de l'actuelle : elle est supposée être un minimum local, et elle est envoyée au solveur récepteur. L'algorithme 10 présente solveur utilisé pour recevoir l'information. Le communication module CM_{set} reçoit plusieurs configurations qui sont reçus par le computation module I_{sort} comme entrées.

Le bénéfice de l'approche en parallèle avec POSL est aussi prouvé pour le *GRP* (voir les tableaux 3 et 3). Dans ces tableaux, la colonne **R** représente le nombre de redémarrages exécutées. Cette expérience a été réalisée en utilisant des solveurs similaires à ceux présentés précédemment, mais sans communication modules.

Pour *GRP*, la stratégie de communication que nous adoptions a été différente. L'idée de cette stratégie est de profiter des nombreux redémarrages indiqués dans les tableaux 3 et 4. Chaque fois qu'un solveur redémarre, la configuration actuelle est communiquée pour

Algorithm 10: Solveur récepteur pour *GRP*

```
abstract solver as_golomb_receiver
computation : I, V, S, A, T
connection : C.M.
begin
    [↻ (Itr < K1)
        [C.M. ↦ I] ↦
        [↻ (Itr % K2) [V ↦ S ↦ A] ] ↦ T
    ]
end
solver Solver_receiver implements as_golomb_receiver
    computation : I_sort, V_sort, S_first, A_AI, , T_tabu
    communication : CM_set
```

Instance	T	It.	R	% success
8–34	0.66	10,745	53	100.00
10–55	67.89	446,913	297	88.00
11–72	117.49	382,617	127	30.00

TABLE 3 – Résultats séquentielles pour *GRP*

alerter les solveurs et éviter son voisinage. De cette façon, chaque fois qu'un solveur redémarre, il génère une nouvelle configuration assez loin de ces "zones pénalisées".

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

Comme nous pouvons voir dans les tableaux 5 et 6 l'amélioration en temps d'exécution avec communication est plus visible quand on utilise l'opérateur de communication one to N, parce-que à chaque nouvelle itération, le solveur récepteur a plus d'information afin de générer une nouvelle configuration loin des "zones pénalisées".

5 Conclusions

Dans cette thèse, nous avons présenté POSL, un système pour construire des solveurs parallèles coopé-

Instance	T	It.	R
8–34	0.43	349	1
10–55	4.92	20,504	13
11–72	85.02	155,251	51

TABLE 4 – Résultats en parallèle non coopératifs pour *GRP*

Instance	T	It.	R
8–34	0.44	309	1
10–55	3.90	15,437	10
11–72	85.43	156,211	52

TABLE 5 – Résultats avec communication sans liste tabu pour *GRP*.

Instance	T	It.	R
8–34	**0.43**	283	1
10–55	**3.16**	12,605	8
11–72	**60.35**	110,311	36

TABLE 6 – Résultats avec communication avec liste tabu pour *GRP*.

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

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

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

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

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

Références

[1] 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.

[2] 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.

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

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

[7] 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.

[8] 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.

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