Étude de stratégies parallèles de coopération avec POSL

Alejandro Reyes Amaro, Eric Monfroy, Florian Richoux

Étude de stratégies parallèles de coopération avec POSL

Alejandro REYES-AMARO[1] **Éric MONFROY**[1] **Florian RICHOUX**[1]

[1] LINA Inria–TASC, Université de Nantes, Nantes, France
{alejandro.reyes,eric.monfroy,florian.richoux}@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. Dans cet article on utilise un Langage pour créer des Solveurs Orienté Parallèle (POSL pour Parallel-Oriented Solver Language) : cadre permettant de construire des solveurs basés sur des méta-heuristiques interconnectées travaillant en parallèle, dans le but de résoudre des instances des problèmes *Social Golfers* et *Costas Array* et de mesurer sa performance. Nous testons plusieurs stratégies de résolution, grâce au langage orienté parallèle, basé sur des opérateurs, que POSL fournis.

Abstract

The multi-core technology and massive parallel architectures are nowadays more accessible for a broad public through hardware like the Xeon Phi or GPU cards. This architecture strategy has been commonly adopted by processor manufacturers to stick with Moore's law. However, this new architecture implies new ways to design and implement algorithms to exploit its full potential. This is in particular true for constraint-based solvers dealing with combinatorial optimization problems. In this paper we use Parallel-Oriented Solver Language (POSL), a framework to build interconnected meta-heuristic-based solvers working in parallel, by using communications operators, to solve instances of *Social Golfers* and *Costas Array* problems and measure its performance. We test many different solution's strategies, thanks to a parallel-oriented language provided, based on operators.

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.

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 [5] est un algorithme efficace, montrant de très bonnes performances et passant à l'échelle de plusieurs centaines ou même milliers de cœurs, en utilisant la recherche locale *multi-walk* en parallèle. Munera et al. [10] ont présenté une autre implémentation d'Adaptive Search en utilisant la coopération entre des stratégies de recherche. 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 [13] nous pré-

sentons 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 *Social Golfers* et *Costas Array*. 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 [3] 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 *computation strategies*, en combinant des *composants* déjà définis (*operation modules* et *open channels*). Une idée similaire a été proposée dans [6] 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, [15] 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 [14]. 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 *open channels*, et de cette façon, ils peuvent partager non seulement des informations, mais aussi leur comportement, en partageant leurs *operation modules*. Cette approche donne aux solveurs la capacité d'évoluer au cours de l'exécution.

Renaud De Landtsheer et al. présentent dans [7] un cadre facilitant le développement des systèmes de recherches en utilisant des *combinators* pour dessiner les caractéristiques trouvées très souvent dans les procédures de recherches comme des briques, et les assembler. Dans [9] est proposée une approche qui utilise des systèmes coopératifs de recherche locale basée sur des méta-heuristiques. Celle-ci se sert de protocoles de transfert de messages. POSL combine ces deux idées pour assembler des composants de recherche locale à travers des opérateurs fournis (ou en créant des nouveaux), mais il fournit aussi un mécanisme basé sur opérateurs pour connecter et combiner des solveurs, en créant des stratégies de communication.

Une présentation plus détaillée de POSL est disponible dans [11,12], où le lecteur peut trouver une description formelle sur la façon d'utiliser le langage basé sur des opérateurs pour construire des prototypes de solveurs. Dans cet article, 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* et *Costas Array*.

3 Construire des solveurs en parallèle avec POSL

Dans cette section, nous présentons un résumé des étapes à suivre pour construire des solveurs parallèles communicatifs en utilisant POSL. L'idée principale est de combiner des modules disponibles dans ce cadre, ou d'en créer des nouveaux, et de les coller à travers POSL pour créer des solveurs indépendants. Après, nous pouvons les connecter en utilisant des *opérateurs de connexion*. Nous appelons l'entité finale obtenue POSL **Meta-Solver**.

3.1 PREMIÈRE ÉTAPE : Créer les modules de POSL

Il existe deux types de modules de base dans POSL : les *operation modules* et les *open channel*. Un *operation module* est une fonction qui reçoit une entrée, puis il exécute un algorithme interne et renvoie une sortie. Les types d'entrée et de sortie définiront l'*operation module*. Il peut être remplacé dynamiquement ou combiné avec d'autres *operation modules*, car ils peuvent être partagés entre les solveurs travaillant en parallèle. De cette manière, le programme de calcul peut changer son comportement au cours de l'exécution. La figure 1 montre un exemple d'*operation module* : il reçoit une configuration S puis calcule l'ensemble V de ses configurations voisines $\{S^1, S^2, \ldots, S\}$.

Un *open channel* est également une fonction qui reçoit et renvoie des informations, mais contrairement à l'*operation module*, l'*open channel* peut recevoir des informations de deux façons différentes : par le paramètre ou de l'extérieur,

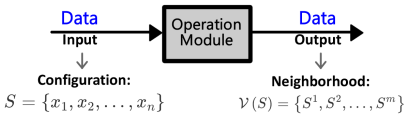
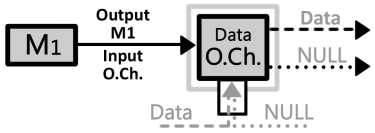
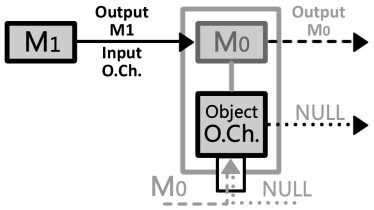


FIGURE 1 – Un *operation module*



(a) Data *open channel*



(b) Object *open channel*

FIGURE 2 – Les *open channels*

c'est-à-dire en communiquant avec un module d'un autre solveur. L'*open channel* est le composant responsable de la réception de l'information dans les communications entre les solveurs. Il peut interagir avec les *operation modules* grâce aux opérateurs.

Un *open channel* peut recevoir deux types d'informations, toujours en provenance d'un solveur externe : des données ou des *operation modules*. Il est important de remarquer que lorsque nous parlons de l'envoi–réception d'*operation modules*, nous entendons l'envoi–réception d'information seulement nécessaire pour les identifier et pouvoir les instancier.

Afin de distinguer les deux types d'*open channels*, nous appellerons Data Open Channel celui qui est responsable de la réception de données (figure 2a), et Object Open Channel de la réception et l'instanciation d'*operation modules* (figure 2b).

Les utilisateurs de POSL peuvent implémenter des nouveaux modules (*operation modules* et *open channels*) mais POSL contient déjà plusieurs modules très utiles pour résoudre un large éventail de problèmes.

3.2 DEUXIÈME ÉTAPE : **Assembler des modules de** POSL

Dans cette étape, une stratégie générique est codée par POSL, en utilisant les modules mentionnés dans la première étape. Elle permet non seulement l'échange d'information, mais aussi l'exécution de modules en parallèle. Nous appelons cela la *computation strategy*.

La *computation strategy* est le cœur du solveur. C'est un programme qui joint les *operation modules* et *open channels* d'une manière cohérente, en restant indépendant des ces derniers. Grâce à la *computation strategy* nous pouvons aussi choisir les informations qui seront envoyées à d'autres solveurs. Une présentation plus formelle de la spécification des opérateurs de POSL est disponible dans [11, 12].

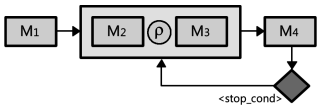


FIGURE 3

La figure 3 présente un exemple simple de la façon de combiner des *modules* en utilisant les opérateurs mentionnés précédemment, et l'algorithme 1 [1] est le pseudo-code POSL correspondant. Dans cet exemple nous montrons quatre *operation modules* qui font partie d'un *compound module* représentant une méthode trivial de recherche locale.

- `M1` : produit une configuration au hasard,
- `M2` : calcule le voisinage d'une configuration donnée, en sélectionnant une variable au hasard, et en changeant sa valeur,
- `M3` : calcule le voisinage d'une configuration donnée, en sélectionnant K variables au hasard, et en changeant leur valeur,
- `M4` : sélectionne (et sauvegarde), dans un ensemble de configurations, celle qui a le coût le plus petit.

Dans cet exemple :

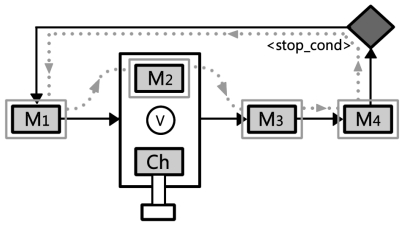
- La fonction **execute** `(A,B)` exécute le module `A` puis le module `B`
- La fonction **while** exécute le module tant qu'une condition donnée est vraie. Dans l'exemple ci-dessous, le processus est répété `N` fois.
- L'*operation module* `M2` est exécuté avec une probabilité ρ (`p` dans le pseudo-code), et `M3` est exécuté avec une probabilité $(1-\rho)$, en utilisant la fonction (opérateur) **rho**.

Algorithm 1: POSL pseudo-code pour la figure 3

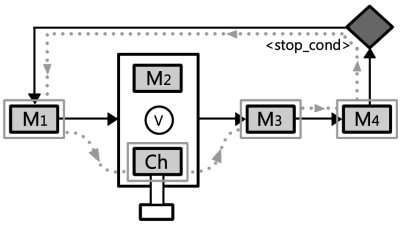
```
1 execute(M1,  while(loops < N)
2                 execute( rho(p, M2, M3), M4);
3               end
4 );
```

La figure 4 montre la manière de combiner un *open channel* avec l'*operation module* `M2` en utilisant l'opérateur (V). Dans ce cas, l'*operation module* `M2` est exécuté tant que l'*open channel* est *NULL*, c'est-à-dire qu'il n'y a

1. Il est nécessaire de signaler que ce pseudo-code est uniquement utilisé pour faciliter la compréhension du principe du POSL. Des exemples du vrai code POSL son disponibles sur https ://github.com/alejandro-reyesamaro/POSL



(a) Le solveur exécute son propre *operation module* si aucune information n'a été reçue par l'*open channel*



(b) Le solveur exécute l'*operation module* qui est arrivé par l'*open channel*

FIGURE 4 – Deux comportements différents du même solveur

pas d'information provenant de l'extérieur. Le comportement est représenté dans la figure 4a par les lignes pointillées. Si un *operation module* a été reçu par l'*open channel*, il est exécuté au lieu de l'*operation module*() `M2`, et ce comportement est représenté dans la figure 4b par des lignes pointillées.

En utilisant ces opérateurs, nous pouvons créer l'algorithme en manipulant différents composants pour trouver la solution d'un problème donné. Ces algorithmes sont fixes mais génériques par rapport à leurs composants (*operation modules* et *open channels*). Cela signifie que nous pouvons construire différents solveurs en utilisant la même stratégie (*computation strategy*), mais avec l'instanciation de différents composants, s'ils ont la bonne signature d'entrée-sortie.

Pour définir une *computation strategy*, nous utilisons l'environnement présenté dans l'algorithme 2, où `Mi` et `Chi` représentent les types des *operation modules* et les types des *open channels* utilisés par la *computation strategy* `St`, déclarée en utilisant le mot-clé **strategy**. Entre accolades, le champ `<computation strategy>` correspond au code POSL basé sur les opérateurs en combinant des *compound modules* déjà déclarés.

Algorithm 2: Définition de la *computation strategy*

```
1 St := strategy
2    oModules: M1, M2, ..., Mn;
3    oChannels: Ch1, Ch2, ..., Chm;
4 {
5    <computation strategy>
6 }
```

Algorithm 3: Définition du solveur

```
1 Sk := solver
2 {
3    cStrategy: st;
4    oModules: m1, m2, ..., mn;
5    oChannels: ch1, ch2, ..., chm;
6 }
```

3.3 TROISIÈME ÉTAPE : **Créer les solveurs**

Avec les *operation modules* et les *open channels* déjà assemblés par la *computation strategy*, nous pouvons créer des solveurs en instanciant les composants déclarés. POSL fournit un environnement à cette fin, présenté dans l'algorithme 3, où `mi` et `chi` représentent les instances des *operation modules* et celles des *open channels* qui sont passés par des paramètres à la *computation strategy* `St`. Cela nous permet de créer de nombreux différents solveurs partageant la même *computation strategy*, mais en exécutant différents *operation modules*. Dans le pseudo-code nous utilisons le mot-clé **solver** pour signaler que nous déclarons un solveur.

3.4 QUATRIÈME ÉTAPE : **Connecter les solveurs**

Une fois que nous avons défini notre stratégie de solveur, la dernière étape consiste à déclarer les *communication channels*, c'est-à-dire connecter les solveurs les uns aux autres. Jusqu'ici, les solveurs sont déconnectés, mais ils ont tout pour établir la communication. Dans cette dernière étape, POSL fournit à l'utilisateur une plateforme pour définir facilement des *méta-stratégies* coopératives que les solveurs doivent suivre.

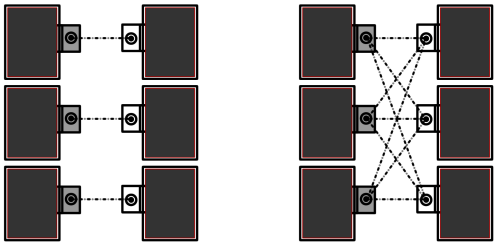
La communication est établie en suivant les règles suivantes :

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

Avec l'opérateur $(\cdot)$ nous pouvons avoir accès aux noms des *open channels* d'un solveur, et aux *operation modules* qui envoient des informations. Par exemple : $Solver_0 \cdot M_0$ fournit un accès à l'*operation module* M_0 de $Solver_0$ si et seulement si il est utilisé par l'opérateur $(\!|.|\!)^o$ (ou $(\!|.|\!)^m$), et $Solver_1 \cdot Ch_1$ fournit un accès à l'*open channel* Ch_1 de $Solver_1$.

Donc, si nous définissons deux ensembles de solveurs, nous pouvons les connecter de deux manières différentes :

1. En connectant chaque solveur du premier ensemble, avec un solveur dans l'autre ensemble (un à un) (voir figure 5a)



(a) 1 à 1 (b) 1 à N

FIGURE 5 – Opérateurs de communication

2. En connectant chaque solveur du premier ensemble, avec tous les solveurs dans l'autre ensemble (un à N) (voir figure 5b)

Ces opérateurs i) affectent (programment) une unité disponible de calcul (de processus ou de fil, selon l'architecture utilisée) à chaque solveur impliqué, et ii) connectent les solveurs impliqués les uns aux autres.

Les opérateurs définis ci-dessus n'expriment que la possibilité de définir statiquement des stratégies de communication. À l'avenir, nous aimerions améliorer POSL en incluant des opérateurs qui permettraient plus de souplesse et d'expressivité en terme de communication entre solveurs, notamment à travers des stratégies dynamiques de communication.

4 Concevoir des expériences

Le but principal de cet article est de sélectionner deux problèmes de référence, *Social Golfers* et le *Costas Array*, pour analyser et illustrer la versatilité de POSL pour étudier des stratégies de solution basées sur la recherche locale méta-heuristique. 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 [2] que nous avons générés pour les expériences.

Le problème de *Social Golfers* 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 [8]. 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.

2. Le code source de POSL est disponible sur https ://github.com/alejandro-reyesamaro/POSL

Le problème *Costas Array* consiste à trouver une matrice *Costas Array*, 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 dooivent tous être différents. Ceci est un problème très complexe trouvant une application utile dans certains domaines comme le sonar et l'ingénierie de radar, et présente de nombreux problèmes mathématiques ouverts. Ce problème a aussi une caractéristique intéressante : même si son espace de recherche grandit factoriellement, à partir de l'ordre 17 le nombre de solutions diminue drastiquement.

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) [2].

4.1 Résoudre le problème *Social Golfers*

Afin de résoudre des instances du problème *Social Golfers*, nous utilisons les *operation modules* suivants fournis par POSL :

1. *operation module* de génération initiale :

   M_S 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. *operation modules* de voisinage :

   M_V^{Std} définit le voisinage $V(s)$ permutant les joueurs parmi les groupes.

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

3. *operation modules* de sélection :

   $M_{\hat{S}}^{First}$ sélectionne la première configuration $\hat{s} \in V(s)$ qui améliore le coût actuel

   $M_{\hat{S}}^{Best}$ sélectionne la meilleure configuration $\hat{s} \in V(s)$ qui améliore le coût actuel

   $M_{\hat{S}}^{Rand}$ sélectionne une configuration aléatoire $\hat{s} \in V(s)$.

4. *operation module* d'acceptation :

   M_D évalue un critère d'acceptation pour $\hat{s}$. Dans tous les cas, le critère d'acceptation est toujours de choisir la configuration avec le moindre coût.

Grâce aux potentiel de POSL, nous avons pu tester de nombreuses stratégies en très peu de temps. Une première expérience a été réalisée pour sélectionner la meilleure fonction de voisinage pour résoudre le problème, en comparant une stratégie basique qui utilise l'*operation module* M_V^{Std}, avec une stratégie basique qui utilise l'*operation module* de voisinage M_V^{AS} basé sur l'algorithme d'*Adaptive Search* ainsi qu'avec les combinaisons de M_V^{Std} et M_V^{AS} après avoir appliquer les opérateurs (ρ) et $(\cup)$. Les algorithmes 4, 5 et 6 représentent respectivement les *computation strategies* pour chaque cas.

Les stratégies mentionnées ci-dessus ne sont pas capables de résoudre les problèmes avec plus de 3 semaines, nous avons donc implémenté une autre stratégie décrite dans l'algorithme 7. Ce dernier combine les *operation modules* de sélection $M_{s'}^{First}$ avec $M_{s'}^{Rand}$, et il tente d'améliorer le coût un certain nombre de fois. Si cela n'est pas possible, il sélectionne un voisin aléatoire pour l'itération suivante. Cette stratégie utilise l'*operation module* M_V^{AS}, et nous avons comparé deux *operation modules* de sélection : $M_{s'}^{First}$ et $M_{s'}^{Best}$.

Après cela, nous avons choisi la meilleure stratégie pour construire des solveurs communicatifs (algorithmes 8 et 9) pour comparer leurs performances avec les stratégies non communicatives. En utilisant les opérateurs de communication, nous concevons différentes stratégies :

— *Stratégie complète de communication* : tous les serveurs sont connectés (soit *1 à 1*, soit *1 à N*)

— *Stratégie de communication hybride* : un certain pourcentage des solveurs sont connectés et le reste sont des solveurs non communicatifs.

4.2 Résoudre le problème *Costas Array*

Pour étudier le problème *Costas Array*, nous avons réutilisé certains *operation modules* utilisés dans la résolution du problème *Social Golfers* : les *operation modules* de *Sélection* et d'*Acceptation*. Les nouveaux modules sont :

1. *operation module* de génération initiale :

   M_S génère une configuration aléatoire s, comme une permutation du vecteur $[1..n]$.

2. *operation module* de voisinage :

   M_V^{AS} définit le voisinage $V(s)$ permutant la variable qui a contribué le plus au coût, avec d'autres.

Nous ajoutons aussi un module de *Reset* (M_R) [3]. La stratégie de base que nous utilisons pour résoudre ce problème est présentée dans l'algorithme 10. Nous le prenons comme base pour construire l'ensemble des différentes stratégies de communication, en combinant des solveurs communicatifs et non communicatifs. En utilisant un solveur émetteur basé sur la stratégie de l'algorithme 11, nous avons testé la communication d'une bonne configuration après avoir sélectionné la configuration pour la prochaine itération (algorithme 12). Nous avons également testé la communication d'une bonne configuration pour la recevoir au moment où le solveur calcule le *reset*, (algorithmes 13 et 14).

Instance	T	It.	% réussi
5-3-7	8.13	17,347	100.00
8-4-7	16.92	7,829	100.00
9-4-8	79.60	20,779	94.28
11-7-5	3.37	664	100.00

TABLEAU 1 – Expériences avec *Social Golfers* (séquentiel)

STRATÉGIE	T	It.
Adaptive Search (AS)	**1.06**	352
Std (ρ) AS	41.53	147
Std $(\cup)$ AS	59.65	198
Standard (Std)	87.90	146

TABLEAU 2 – Expériences avec *Social Golfers* 10-10-3

5 Analyse des résultats

Les expériences [4] [5] 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. Nous utilisons 40 cœurs pour le problème *Social Golfers* et seulement 20 pour *Costas Array*, la machine n'étant pas complètement disponible au moment des expériences. Nous présentons dans les tableaux 1 et 7 les résultats du lancement des POSL métasolveurs pour résoudre chaque cas de manière séquentielle. Le tableau 1 montre de moyennes de temps et d'itérations beaucoup plus élevées que les moyennes que on peut trouver dans le tableau 3 colonne $OM_{s'}$ *First Improvement* (sans communication), et les résultats avec communication (tableaux 4, 5 et 6). Le même effet est visible dans le tableau 7. Ceci montre comment l'approche parallèle augmente la probabilité de trouver la solution dans un délai raisonnable (en secondes), par rapport au schéma séquentiel [1]. La colonne marquée **% success** indique le pourcentage de solveurs qui ont été capables de trouver une solution avant d'arriver à certain nombre maximal d'itérations (soit 25 000) ou au temps d'expiration (soit 5 minutes).

3. Il est basé sur le code de Daniel Díaz disponible dans https ://sourceforge.net/projects/adaptivesearch/

4. POSL source code is available in https ://github.com/alejandro-reyesamaro/POSL

5. Document supplémentaire *Experiments[02-2016].ods* disponible sur : https ://goo.gl/apsVSF

Instance	$OM_{s'}$ Best Improvement		$OM_{s'}$ First Improvement	
	T	It.	T	It.
5-3-7	4.99	4,421	**1.32**	1322
8-4-7	5.10	954	**1.82**	445
9-4-8	12.37	1,342	**6.43**	873
11-7-5	5.19	351	**2.22**	273

TABLEAU 3 – Expériences avec *Social Golfers* en comparant les fonctions de sélection

Il est important de souligner que POSL ne vise pas à obtenir les meilleurs résultats en termes de performance, mais de donner la possibilité de prototyper rapidement et d'étudier différentes stratégies de recherche coopératives ou non coopératives. Notre objectif n'est donc pas de construire des solveurs plus rapides, mais d'obtenir des moyens plus rapides pour les étudier.

Dans la première étape des expériences, nous utilisons le langage basé sur des opérateurs fournis par POSL pour construire et tester de nombreuses et différentes stratégies non communicatives. L'objectif est de sélectionner la meilleure stratégie pour exécuter des tests avec communication. D'abord, nous nous sommes concentrés sur le choix de la bonne fonction de voisinage. Dans le cas du problème de *Social Golfers*, cette expérience a été lancée en utilisant une stratégie basique montrée dans l'algorithme 4 pour bien nous concentrer sur l'étude des modules et non sur la *computation strategy*. Cette stratégie n'a pas été capable de résoudre les problèmes pour les cas de plus de trois semaines, car elle tombait très souvent dans des minimums locaux. Cela a été la raison pour laquelle nous effectuons cette expérience avec l'instance 10–10–3.

Les résultats du tableau 2 ne sont pas surprenants. L'*operation module* de voisinage OM_V^{AS} est basé sur l'algorithme *Adaptive Search*, qui a montré de très bons résultats. Il sélectionne la variable (joueur) qui contribue le plus au coût et permute sa valeur avec les autres variables (joueurs) pour tous les groupes, et pour chaque semaine. OM_V^{Std} n'utilise aucune information supplémentaire. Il effectue donc chaque permutation possible entre deux joueurs dans les différents groupes, chaque semaine. Cela signifie que ce voisinage est $g \times p$ (nombre de groupes et nombre de joueurs, respectivement) fois plus grand que le précédent. La combinaison avec l'opérateur (ρ) exécute le module OM_V^{Std} ou le OM_V^{AS}, en fonction de la probabilité ρ. La combinaison avec l'opérateur $(\cup)$ est l'union de ces voisinages. Dans ces trois derniers, le temps passé pour la recherche à l'intérieur du voisinage à chaque itération est significatif, même si le nombre d'itérations dans ces trois cas est inférieure à celui qu'utilise le module OM_V^{AS}, car ce dernier effectue de nombreux *Reset* pendant le processus de recherche.

Une fois avoir obtenu la bonne fonction de voisinage, nous nous sommes concentrés sur le choix de la meilleure fonction de *sélection*. Nous avons comparé deux *operation modules* différents en utilisant la *computation strategy* de l'algorithme 7. Cette dernière combine les *operation modules* de sélection ($M_{s'}^{First}$ ou $M_{s'}^{Best}$) avec $M_{s'}^{Rand}$, pour éviter les minimums locaux : elle tente d'améliorer le coût un certain nombre de fois. Si elle n'y arrive pas, elle sélectionne un voisin aléatoire pour l'itération suivante. Le premier *operation module* a été $OM_{S'}^{Best}$ qui sélectionne la meilleure configuration à l'intérieur du voisinage. Il n'a pas seulement passé plus de temps à chercher une meilleure configuration, mais il est également plus sensible pour tomber dans des minimums locaux. Le deuxième *operation module* a été $OM_{S'}^{First}$ qui sélectionne la première configuration à l'intérieur du voisinage en améliorant le coût actuel. En utilisant ce module, la stratégie perd le facteur d'intensification, mais gagne en vitesse et en diversification. Le tableau 3 présente les résultats de cette expérience, en montrant qu'une stratégie orientée plus exploratoire est plus efficace pour le problème de *Social Golfers*.

Dès lors que la meilleure stratégie a été choisie, nous lançons des expériences pour étudier le comportement de POSL pour résoudre les problèmes ciblés dans des scénarios avec communication. Certaines compositions de POSL méta-solveurs ont été prises en compte : i) la structure de la communication (avec–sans communication ou un mélange), et ii) la stratégie de communication codée dans le solveur.

Instance	OP : 1 à 1		OP : 1 à N	
	T	It.	T	It.
5-3-7	1.19	1,156	1.11	1,067
8-4-7	**1.30**	**317**	1.46	347
9-4-8	4.38	**597**	5.51	736
11-7-5	1.76	214	**1.62**	**202**

TABLEAU 4 – Expériences avec *Social Golfers* en testant 100% de communication

Instance	OP : 1 à 1		OP : 1 à N	
	T	It.	T	It.
5-3-7	1.04	1,019	1.04	1,031
8-4-7	1.40	337	1.43	353
9-4-8	4.64	637	5.75	776
11-7-5	1.81	220	1.82	222

TABLEAU 5 – Expériences avec *Social Golfers* en testant 50% de communication

Instance	OP :1 à 1		OP : 1 à N	
	T	It.	T	It.
5-3-7	**0.90**	**881**	1.19	1,170
8-4-7	1.39	341	1.46	352
9-4-8	**4.33**	599	4.53	625
11-7-5	1.99	242	1.63	224

TABLEAU 6 – Expériences avec *Social Golfers* en testant 25% de communication

Chaque fois qu'un POSL méta-solveur est lancé, de nombreux processus de recherche indépendants sont exécutés. Dès qu'une bonne configuration est trouvée, elle est transmise du solveur émetteur au récepteur. À ce moment, si l'information est acceptée, il y a quelques solveurs qui recherchent dans le même sous-ensemble de l'espace de recherche, et la stratégie de recherche devient plus orientée à l'exploitation. Cela peut être problématique si la stratégie conduit souvent les solveurs dans des minimums locaux. Dans ce cas, nous ne perdons pas qu'un seul solveur, mais deux ou plus, en fonction de la stratégie de communication. Nous pouvons éviter ce phénomène par une simple mais efficace opération : si un solveur n'est pas capable de trouver une meilleure configuration à l'intérieur du voisinage avec l'exécution de l'*operation module* $OM_{S'}^{First}$, il en sélectionne une au hasard, en exécutant donc l'*operation module* $OM_{S'}^{Rand}$. En utilisant la communication entre les solveurs, cette stratégie produit un certain gain en terme de temps d'exécution (tableau 3 par rapport aux tableaux 4, 5 et 6 pour *Social Golfers*, et tableau 7 par rapport au tableau 8 pour *Costas Array*). Le pourcentage de solveurs récepteurs qui ont été capables de trouver la solution avant les autres, était important, comme nous pouvons le voir dans les tableaux du document supplémentaire disponible sur Internet [6]. Cela montre que la communication a joué un rôle important lors de la recherche, malgré les frais généraux de la communication entre les processeurs (réception, interprétation des informations, pris des décisions, etc.).

Pour la résolution du problème *Costas Array*, le tableau 8 montre que la stratégie 12 est plus performante que les stratégies 13 et 14. Même si la communication de la stratégie 12 est plus soutenue (à chaque itération), elle est faite au bon moment, après avoir sélectionné la configuration pour la prochaine itération : à cette étape, le solveur recherche une configuration avec un coût global inférieur à la configuration courante. Il décide ainsi s'il doit prendre pour la prochaine itération une configuration du voisinage ou la configuration communiquée. La communication dans les stratégies 13 et 14 est faite au moment de faire le *reset*, donc nettement moins souvent. Cette stratégie empêche aussi de faire correctement le *reset* car à chaque fois il est comparé avec une configuration qui a forcement un meilleur coût. Par contre, le surcoût de la communication est élevé, quand la stratégie 12 est utilisée avec une communication de type 1 à N.

Dans les expériences réalisées, l'information partagée était dans tous les cas la meilleure configuration trouvée. Jusqu'à présent, il n'y a pas de résultat en affirmant que c'est la meilleure stratégie. En fait, [4] montre que la configuration courante n'est pas une information intéressante pour partager entre les solveurs. Voilà pourquoi ce sujet mérite une étude plus approfondie. Nous envisageons dans l'avenir, d'enquêter sur d'autres informations à communiquer, telles que des configurations très coûteuses, afin d'éviter d'autres semblables, directions de recherche à éviter ou à prendre en compte, etc.

6. Document supplémentaire *Experiments[02-2016].ods* disponible sur : https ://goo.gl/apsVSF

STRATÉGIE	T	It.	% réussi
Séquentiel (1 cœur)	2.24	35,299	48.50
Parallèle (20 cœurs)	1.44	24,041	100.00

TABLEAU 7 – Expériences avec *Costas Array* 17 (sans communication)

	100% comm.		OP : 50% comm.	
STRATÉGIE	T	It.	T	It.
Str A : 1 à 1	0.89	10,995	1.12	15,427
Str A : 1 à N	1.04	12,847	1.07	15,086
Str B : 1 à 1	0.85	11,431	0.95	14,441
Str B : 1 à N	0.94	12,723	1.07	15,820
Str C : 1 à 1	1.07	11,948	**0.80**	**10,324**
Str C : 1 à N	1.34	17,473	1.32	19,007

TABLEAU 8 – Expériences avec *Costas Array* 17 avec communication (20 cœurs)

6 Conclusions

Dans cet article, nous avons présenté quelques premiers résultats en utilisant POSL pour résoudre des instances des problèmes *Social Golfers* et *Costas array*. Il a été possible d'implémenter différentes stratégies communicatives et non communicantives, grâce au langage basé sur des opérateurs fournis, pour combiner différents *operation modules*. POSL donne la possibilité de définir des *canaux de communication* reliant 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 d'*operation modules* et d'*open channels* prête à utiliser, sur la base d'une étude approfondie sur les algorithmes métaheuristiques 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] Noga Alon, Chen Avin, M Koucký, Michal Koucky, Gady Kozma, Zvi Lotker, and Mark R. Tuttle. Many Random Walks Are Faster Than One. *Combinatorics, Probability and Computing*, 20(4) :481–502, 2011.

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

[3] 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.

[4] 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.

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

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

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

[8] Frédéric Lardeux, Éric Monfroy, Broderick Crawford, and Ricardo Soto. Set Constraint Model and Automated Encoding into SAT : Application to the Social Golfer Problem. 2014.

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

[10] 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.

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

[12] 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.

[13] Alejandro Reyes-Amaro, Éric Monfroy, and Florian Richoux. Un langage orienté parallèle pour modéliser des solveurs de contraintes. In *Onzièmes Journées Francophones de Programmation par Contraintes (JFPC)*, Bordeaux, 2015.

[14] 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.

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

Annexe A : POSL stratégies

Algorithm 4: Stratégie standard

```
1 st_std := strategy
2    oModules: M_S, M_V, M_SE, M_D;
3 { // Iter : number of iterations
4    execute(M_S,  while(Iter < k1)
5                     execute(M_V,  execute(M_SE, M_D));
6                  end
7    );
8 }
```

Algorithm 5: Combinaison de deux fonctions de voisinage en utilisant l'opérateur *RHO*

```
1 st_rho := strategy
2    oModules: M_S, M_V1, M_V2, M_SE, M_D;
3 {
4    execute(M_S,  while(Iter < k1)
5                     execute( rho(p, M_V1, M_V2),
6                              execute(M_SE, M_D)
7                     );
8                  end
9    );
10 }
```

Algorithm 6: Ensemble de deux fonctions de voisinage

```
1 st_u := strategy
2    oModules: M_S, M_V1, M_V2, M_SE, M_D;
3 {
4    execute(M_S,  while(Iter < k1)
5                     execute( union(M_V1, M_V2),
6                              execute(M_SE, M_D)
7                     );
8                  end
9    );
10 }
```

Algorithm 7: Stratégie pour échapper des minimums locaux

```
st_eager := strategy
   oModules: M_S, M_V, M_SE, M_D;
{  // Sci: number of iterations with the same cost
   execute(M_S,
           while(Iter < k1)
              execute(M_V,
                      execute(?(Sci < k2, M_SE, M_SE.Rand),
                              M_D
                      )
              );
           end
   );
}
```

Algorithm 8: Stratégie communicative (émetteur)

```
st_eager_sender := strategy
   oModules: M_S, M_V, M_SE, M_D;
{
   execute(M_S,
           while(Iter < k1)
              execute(M_V,
                      execute(?( Sci < k2,
                                 send(M_SE),
                                 M_SE.Rand),
                              M_D
                      )
              );
           end
   );
}
```

Algorithm 9: Stratégie communicative (récepteur)

```
st_eager_sender := strategy
   oModules: M_S, M_V, M_SE, M_D;
   oChanels: Ch_DP;
{
   execute(M_S,
           while(Iter < k1)
              execute(M_V,
                      execute(?( Sci < k2,
                                 ?( Sci < k3,
                                    min(M_SE, Ch_DP) ,
                                     M_SE),
                                 M_SE.Rand),
                              M_D
                      )
              );
           end
   );
}
```

Algorithm 10: Stratégie basée sur des Resets

```
st_hard := strategy
   oModules: M_S, M_R, M_V, M_SE, M_D;
{
   execute(M_S,
           while(Iter < k1)
              execute(M_R,
                      while(Iter < k2)
                         execute(M_V,  execute(M_SE, M_D));
                      end
              );
           end
   );
}
```

Algorithm 11: Stratégie basée sur des Resets (émetteur)

```
  st_hard_sender := strategy
oModules: M_S, M_R, M_V, M_SE, M_D;
{
   execute(M_S,
           while(Iter < k1)
              execute(M_R,
                      while(Iter < k2)
                         execute(M_V,  execute(M_SE, send(M_D)));
                      end
              );
           end
   );
}
```

Algorithm 12: Stratégie A basée sur des Resets (récepteur)

```
st_hard_receiver_a := strategy
   oModules: M_S, M_R, M_V, M_SE, M_D;
   oChannels: Ch_Sol;
{
   execute(M_S,
           while(Iter < k1)
              execute(M_R,
                      while(Iter < k2)
                         execute(M_V,  execute(M_SE,
                                               min(M_D, Ch_Sol)
                                 )
                         );
                      end
              );
           end
   );
}
```

Algorithm 13: Stratégie B basée sur des Resets (récepteur)

```
st_hard_receiver_b := strategy
   oModules: M_S, M_R, M_V, M_SE, M_D;
   oChannels: Ch_Sol;
{
   execute(M_S,
           while(Iter < k1)
              execute(?(Iter % k2, M_R,  min(M_R, Ch_Sol)),
                      while(Iter < k2)
                         execute(M_V,  execute(M_SE, M_D)));
                      end
              );
           end
   );
}
```

Algorithm 14: Stratégie C basée sur des Resets (récepteur)

```
st_hard_receiver_c := strategy
   oModules: M_S, M_R, M_V, M_SE, M_D;
   oChannels: Ch_Sol;
{
   execute(M_S,
           while(Iter < k1)
              execute(min(M_R, Ch_Sol),
                      while(Iter < k2)
                         execute(M_V,  execute(M_SE, M_D));
                      end
              );
           end
   );
}
```