Parallélisation Portfolio de Solveur PPC

Tarek Menouer, Bertrand Le Cun

Parallélisation Portfolio de Solveur PPC

Tarek Menouer [1] **Bertrand Le Cun**[1]

[1] Laboratoire PRISM, 45 avenue des Etats-Unis 78035 Versailles
{Tarek.menouer,Bertrand.lecun}@prism.uvsq.fr

Résumé

Cet article présente un solveur de Programmation Par Contraintes (PPC) parallèle, basé sur la méthode du Portfolio pour résoudre rapidement les problèmes de satisfaction et d'optimisation de contraintes. Le Portfolio est largement utilisé dans la parallélisation des solveurs de SATisfiabilité booléenne (SAT) et les solveurs de PPC. Le principe consiste à exécuter plusieurs stratégies de recherche pour résoudre le même problème, en utilisant différents cœurs de calcul. Classiquement une stratégie de recherche est exécutée sur un cœur de calcul (Portfolio N To N). La première stratégie qui répond aux besoins de l'utilisateur arrête toutes les autres stratégies. En utilisant une parallélisation interne pour chaque stratégie on peut exécuter N stratégies sur P cœurs de calculs avec $P > N$ (Portfolio N To P). La nouveauté, consiste à adapter l'ordonnancement des N stratégies de recherche entre elles afin de privilégier la stratégie la plus prometteuse et de lui donner plus de cœurs que les autres. Les performances obtenues en utilisant le solveur Portfolio sont illustrées par la résolution des problèmes de PPC modélisés en utilisant le format FlatZinc et résolus avec le solveur Google OR-Tools au-dessus de notre framework parallèle Bobpp.

1 Introduction

Nous présentons dans cet article un Portfolio Adaptatif pour résoudre les problèmes de Programmation Par Contraintes (PPC). La résolution des problèmes de PPC est basée sur le choix des stratégies de recherche. Une stratégie de recherche est un algorithme qui choisit pour chaque nœud dans l'arbre de recherche, une variable à assigner avec une certaine valeur. Le problème est qu'on ne peut pas connaître à l'avance quelle est la meilleure stratégie pour résoudre un problème de PPC. Il existe plusieurs stratégies de recherche mais il n'existe pas une stratégie optimale pour résoudre tous les problèmes de PPC. D'un autre coté, les ressources de calcul sont de plus en plus puissantes et disponibles (les data-centers, le cloud computing, etc). Pour bénéficier de la variété des stratégies de recherche et de la puissance de calcul, nous présentons une méthode de parallélisation Portfolio. Le principe de Portfolio consiste à exécuter plusieurs stratégies de recherche sur différents cœurs de calcul. La première stratégie qui répond aux besoins de l'utilisateur, arrête toutes les autres stratégies. Le but de Portfolio est de résoudre tous les problèmes le plus rapidement possible, mais l'inconvénient est qu'on utilise beaucoup de ressources de calcul. Le Porfolio est utilisé principalement pour résoudre les problèmes de SATisfiabilité booléenne (SAT) [18, 16, 8]. Il existe plusieurs modèles de Portfolio. Le premier modèle et le Portfolio N To N, il consiste à exécuter N stratégies de recherche séquentielle sur P cœur de calcul. L'intérêt de Portfolio N To N est que la performance obtenue est celle de la meilleure stratégie, mais la faiblesse est que le nombre de stratégie est limité par comparaison avec le grand nombre de cœurs de calcul utilisés par les machines parallèles. En utilisant une parallélisation interne pour chaque stratégie de recherche, on peut exécuter N stratégies sur P cœurs ($P > N$), ce modèle de Portfolio est appelé le Portfolio N To P. La parallélisation interne consiste à partitionner l'arbre de recherche de chaque stratégie en un ensemble de sous-arbres, ensuite affecter ces sous-arbres aux différents cœurs de calcul [9]. La première manière de réaliser ce modèle de Portfolio est de partitionner l'arbre de recherche de chaque stratégie en $\frac{P}{N}$ sous-arbres, et ensuite affecter chaque sous-arbre à un cœur de calcul. La deuxième manière, consiste à affecter dynamiquement les P cœurs de calcul à la totalité des stratégies de recherche (les N stratégies), en utilisant la technique du vol de travail. Les deux modèles de Portfolio (N To N et N To P) gaspillent beaucoup de ressource de calcul car on ne sait pas décider à priori qu'elle est la meilleure stratégie, mais pendant la recherche on peut estimer l'avancement respectif des stratégies. Comme

toutes les stratégies sont ordonnancées par le même framework parallèle (Bobpp [2]), donc on peut contrôler l'ordonnancement des N stratégies entre elles et privilégier la stratégie la plus prometteuse, celle qu'on estime avoir l'arbre de recherche le plus petit, et on lui donne plus de cœurs que les autres stratégies. Ce modèle de Portfolio est appelé le Portfolio Adaptatif.

La section suivante détaille plus précisément la résolution des problèmes de PPC. Dans la section 3, les différents modèles de Portfolio sont présentés. Des expérimentations pour résoudre des problèmes de PPC modélisés en utilisant le format Flatzinc sont données dans la section 4. Enfin, une conclusion et quelques perspectives sont proposées en section 5.

2 La Résolution des Problèmes de Programation Par Contraintes

Un problème de PPC est constitué d'un ensemble de variables, $X = \{x_1, x_2, ..., x_n\}$, pour chaque variable $x \in X$, il existe un ensemble fini de domaines de valeurs, $D(x) = \{a_1, a_2, ..., a_k\}$ et une collection finie de contraintes, $C = \{c_1, c_2, ..., c_m\}$.

Un problème (γ) de PPC peut être résolu comme suit : Au début, toutes les variables de γ sont non assignées. A chaque étape, une variable x est choisie, et une valeur possible $a \in D(x)$ est assignée à son tour. Chaque branche d'un arbre de recherche calculée par cette recherche définit une assignation. Ensuite, le mécanisme de propagation vérifie, pour chaque valeur, la cohérence de cette assignation partielle avec les contraintes. En cas de cohérence, un appel récursif est effectué. Chaque assignation partielle crée un nœud dans l'arbre de recherche. Ainsi, nous associons la consistance d'un nœud avec la cohérence implicite d'une assignation.

Il existe plusieurs heuristiques pour choisir les variables de branchement, et pour chaque variable la valeur à assigner, ces heuristiques sont appelées les stratégies de recherche [14, 1, 6]. Les stratégies de recherche utilisées par le solveur OR-Tools [13] dépendent uniquement des données locales d'un nœud, c'est à dire, le branchement ne dépend pas de l'historique de la recherche.

OR-Tools est un solveur de PPC open source séquentiel, il est développé en C++ par l'équipe de recherche Google. Le principe de ce solveur est d'explorer des espaces de recherche pour trouver une ou toutes les solutions possibles en utilisant un algorithme de recherche en profondeur d'abord (Depth First Search *DFS* [1])

Par exemple, pour résoudre le problème $X/Y = Y$ avec,

— Le domaine des valeurs de la variable X = $\{9, 18\}$

— Le domaine des valeurs de la variable Y = $\{3, 6\}$

On peut choisir deux stratégies de recherche, parmi les différentes stratégies qui existent :

— **Stratégie Min Bound** : cette stratégie assigne pour chaque variable non assignée la plus petite valeur dans le domaine des valeurs. La figure 1 présente le déroulement de cette stratégie pour résoudre le problème $X/Y = Y$. Nous commençons par assigner à la variable X la plus petite valeur dans le domaine des valeurs, qui est la valeur 9. Ensuite, nous assignons à la variable Y la plus petite valeur qui est la valeur 3. Maintenant toutes les variables sont assignées. Nous testons si la contrainte $X/Y = Y$ est satisfaite, dans ce cas, $9/3 = 3$ est satisfait, donc une solution est trouvée en explorant 2 nœuds.

— **Stratégie Max Bound** : cette stratégie est l'inverse de la stratégie Min Bound, elle sélectionne pour chaque variable non assignée la plus grande valeur dans le domaine des valeurs. La figure 2 présente le déroulement de cette stratégie pour résoudre le problème $X/Y = Y$. La solution est trouvée en explorant 6 nœuds.

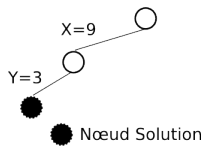


FIGURE 1 – Stratégie Min Bound

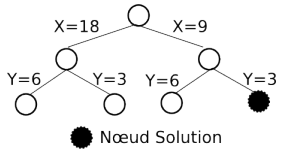


FIGURE 2 – Stratégie Max Bound

Comme on peut remarquer, pour résoudre un problème de PPC, la taille de l'arbre de recherche généré peut varier en fonction de la stratégie utilisée. Cette variation de taille influence fortement les performances obtenues.

Dans la section suivante, nous présentons quelques modèles de Portfolio pour résoudre les problèmes de PPC en utilisant le solveur OR-Tools au-dessus du framework parallèle Bobpp.

Bobpp est un framework parallèle open source, développé en C++ pour résoudre les problèmes d'optimisation combinatoire. Il peut être utilisé comme un support d'exécution. Le but de ce framework est de fournir une interface entre les solveurs de problèmes combinatoires et les machines parallèles en utilisant plusieurs environnements de programmation parallèle, tels que les POSIX threads, MPI, Hybride (POSIX threads+MPI) ou des bibliothèques plus spécialisées telles que Athapascasn/Kaapi [5].

3 Les Modèles de Portfolio

3.1 Portfolio N To N (N Stratégies $\times$ N Cœurs)

Le Portfolio N To N consiste à exécuter N stratégies de recherche sur N cœurs de calcul. Ensuite, la première stratégie qui répond aux besoins de l'utilisateur arrête toutes les autres stratégies, comme présenté dans la figure 3. Ce modèle de Portfolio est largement utilisé dans la parallélisation des solveurs SAT [18, 16, 8] et des solveurs de PPC [3].

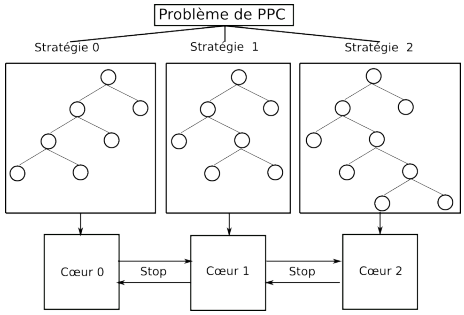


FIGURE 3 – Portfolio N To N

L'avantage de ce modèle de Portfolio est que la performance obtenue est celle de la meilleure stratégie [17], sauf que le nombre de stratégies de recherche qui existent dans la littérature est limité par comparaison avec le grand nombre de cœurs de calcul utilisés par les machines parallèles. Pour bénéficier de ce grand nombre de cœurs de calcul, nous proposons le Portfolio N To P avec $P > N$.

3.2 Portfolio N To P (N Stratégies $\times$ P Cœurs)

En utilisant une parallélisation interne pour chaque stratégie de recherche, on peut exécuter N stratégies de recherche sur P cœurs de calcul avec un bon équilibrage de charge entre les différents cœurs. Il existe plusieurs parallélisations internes [15, 7, 10, 4], dans notre cas nous avons choisi une parallélisation interne qui est présentée en détaille dans [9]. Elle consiste à partitionner l'arbre de recherche de chaque stratégie en un ensemble des sous-arbres disjoints, ensuite chaque sous-arbre est affecté à un cœur de calcul.

Pour réaliser ce modèle de Portfolio, il existe deux méthodes pour affecter les P cœurs de calcul au N stratégies de recherche : affectation statique ou dynamique.

3.2.1 Affectation Statique

L'affectation statique consiste à partitionner l'arbre de recherche de chaque stratégie en $\frac{P}{N}$ sous-arbres, et ensuite affecter statiquement chaque sous-arbre à un cœur de calcul. Généralement, les arbres de recherche générés pour résoudre les problèmes de PPC sont déséquilibrés, donc l'utilisation de cette affectation statique implique que parfois un cœur de calcul effectue quasiment la totalité de la recherche, tandis que les autres cœurs attendent que celui-ci termine sa recherche. Ce partitionnement donne un mauvais équilibrage de charge entre les cœurs de calcul, tel que présenté dans la figure 4. Pour résoudre ce problème, il existe une autre méthode d'affectation, qui est l'affectation dynamique.

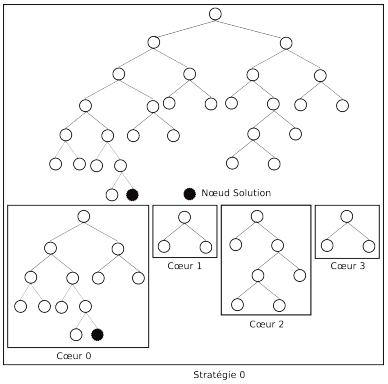


FIGURE 4 – L'équilibrage de charge pour une seule stratégie de recherche en utilisant l'affectation statique

3.2.2 Affectation Dynamique

Le principe de l'affectation dynamique est que l'arbre de recherche de chaque stratégie est partitionné entre les cœurs de calcul en demande et pendant l'exécution de l'algorithme de recherche, pour avoir un bon équilibrage de charge entre les cœurs. Pour partager les sous-arbres entre les cœurs de calcul, l'affectation dynamique utilise une File de Priorité Globale (FPG) implémentée dans le framework parallèle Bobpp. Chaque cœur de calcul effectue la recherche localement et de façon séquentielle en utilisant le solveur OR-Tools. La FPG contient initialement un nœud départ qui sera pris par le premier cœur de calcul pour commencer la recherche. Quand un cœur de calcul a terminé sa recherche dans son sous-arbre, il demande un nouveau sous-arbre à partir de la FPG. Si la FPG est vide, le cœur de calcul se déclare comme un cœur en attente. Les autres cœurs, les cœurs actifs, qui sont en train d'effectuer une recherche sur leurs sous-arbres, testent régulièrement s'il y a un cœur en attente. Si c'est le cas, le premier cœur actif qui détecte qu'il existe un cœur en attente, partitionne son sous-arbre en deux parties, il garde pour lui la partie droite et il insère la partie gauche dans la FPG. Le cœur en attente prend effet par l'insertion d'un nouveaux sous-arbre dans la

FPG.

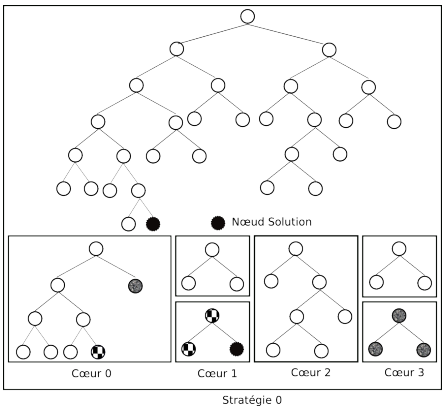


FIGURE 5 – L'équilibrage de charge pour une seule stratégie de recherche en utilisant l'affectation dynamique

La figure 5 présente l'équilibrage de charge obtenu en utilisant l'affectation dynamique. Le lecteur peut facilement voir la différence entre l'équilibrage de charge obtenu par l'affectation dynamique et l'équilibrage de charge obtenu par l'affectation statique présenté dans la figure 4. Il est clair que l'affectation dynamique donne un meilleur équilibrage de charge.

Le problème de l'affectation dynamique est qu'à chaque fois qu'un cœur actif détecte qu'il existe un cœur en attente, il partitionne son sous-arbre avec le cœur en attente sans connaître à priori la taille de son sous-arbre. L'utilisation de ce principe de partitionnement illimité, peut générer beaucoup de sous-arbres, ainsi pour les sous-arbres de petite taille, la résolution séquentielle est parfois plus rapide que la résolution parallèle. Cela vient du fait que le partitionnement des sous-arbres est cher. Pour effectuer un partitionnement, il faut :

- Sauvegarder l'état de toutes les variables dans ce que nous appelons un *Nœud-Bobpp*
- Insérer le *Nœud-Bobpp* dans la FPG
- Pour le cœur en attente qui récupère le *Nœud-Bobpp*, il doit :
  - Ré-initialiser l'état des variables
  - Recommencer la recherche

Pour résoudre le problème de partitionnement illimité, nous proposons d'évaluer la taille des sous-arbres avant d'effectuer le partitionnement. L'évaluation proposée, consiste à calculer le pourcentage des variables non assignées par rapport au nombre total des variables. Si ce pourcentage est plus grand qu'un certain seuil (α%), appelé le seuil de partitionnement, on autorise l'opération de partitionnement. Sinon, une exploration séquentielle est appliquée.

Pour déterminer la valeur du seuil de partitionnement (α%), nous avons effectué quelques expériences, dont les résultats sont présentés dans la figure 6.

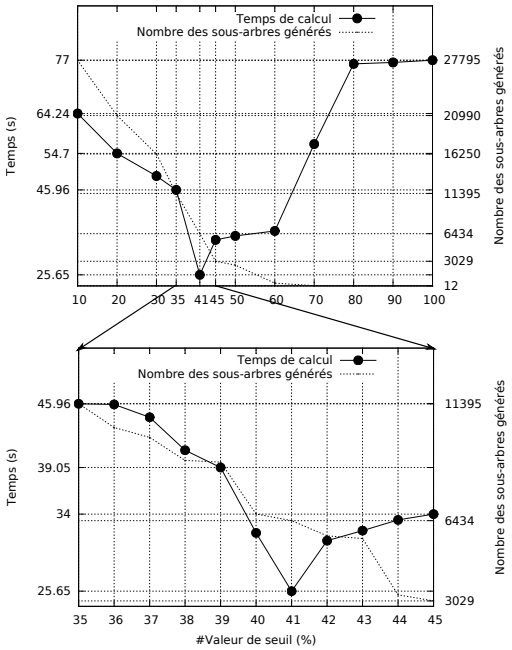


FIGURE 6 – Temps de calcul pour résoudre le problème de Naval Battle (sb_sb_13_13_6_4f̀zn [11]) qui est un problème de satisfaction de contraintes en utilisant un Portfolio 4 To 12 avec une affectation dynamique

La figure 6 montre le temps de calcul en fonction de la valeur du seuil de partitionnement pour résoudre le problème de Naval Battle (sb_sb_13_13_6_4f̀zn [11]), qui est un problème de satisfaction de contraintes en utilisant un Portfolio 4 To 12 avec une affectation dynamique. Dans la figure 6 :A, la valeur du seuil varie entre 10% et 100%. On remarque qu'entre 35% et 45% il y a un seuil optimal. Dans la figure 6 :B, nous avons agrandi le résultat en faisant varier le seuil de partitionnement entre 35% et 45%. Effectivement, la valeur optimale du seuil est de 41%. Donc, il existe un seuil optimal pour lequel le temps de calcul est réduit au minimum. Ce seuil représente le meilleur compromis entre un nombre limité des sous-arbres et un temps d'exécution réduit.

Plus le seuil est petit plus on génère un grand nombre de sous-arbres, ce qui facilite l'équilibrage de charge, mais augmente le temps de calcul. Inversement, plus le seuil est grand plus on se rapproche de l'affectation statique où le nombre des sous-arbres est limité et cela donne un mauvais équilibrage de charge.

Le Portfolio N To P avec une affectation dynamique et un seuil de partitionnement fonctionne bien et donne des bons résultats par rapport au Portfolio N To N (modèle classique), présenté dans la sous-section 3.1.

La différence principale entre les stratégies de recherche est la taille de l'arbre généré par chaque stratégie, plus la taille de l'arbre est petite, plus le temps de la résolution est plus petit. Dans le solveur OR-Tools le temps de traitement d'un nœud dans un arbre de recherche généré par n'importe quelle stratégie pour résoudre le même problème de PPC est pratiquement stable (le même temps d'éxecution).

Il est donc intéressant d'adapter le Portfolio N To P avec l'affectation dynamique et un seuil afin de privilégier au moment du partitionnement la stratégie qui est représentée par le plus petit arbre de recherche.

3.3 Portfolio N To P Adaptatif (Adaptation de N Stratégies $\times$ P Cœurs)

Comme toutes les stratégies sont ordonnancées par le même framework parallèle (Bobpp), donc on peut contrôler l'ordonnancement des stratégies entre elles et privilégier la stratégie la plus prometteuse, celle qu'on estime avoir l'arbre de recherche le plus petit, et on lui donne plus de cœurs que les autres stratégies.

Pour déterminer la stratégie qui est candidate à trouver une solution plus rapidement que les autres (la plus prometteuse), à chaque fois qu'un cœur de calcul visite un nœud dans un sous-arbre de n'importe quelle stratégie, il évalue la quantité de travail qui reste à faire. Lorsqu'un cœur actif détecte qu'il y a un cœur en attente, le cœur actif partitionne son sous-arbre si l'évaluation de son sous-arbre est plus grande que le seuil de partitionnement et si sa stratégie de recherche est la meilleure stratégie, c'est à dire que c'est la stratégie pour laquelle il reste le plus petite travail à faire. Sinon, il ne partitionne pas son sous-arbre et il poursuit une exploration séquentielle de l'arbre de recherche. En utilisant ce principe, nous pouvons être sûrs que la stratégie qui partitionne son sous-arbre est la stratégie de recherche la plus prometteuse et on affecte plus de cœurs de calcul à cette stratégie la plus prometteuse.

Pour estimer la quantité de travail qui reste à explorer, nous estimons le nombre maximal des branches qui restent à visiter, et qui est représenté par le produit cartésien de la taille du domaine de valeurs pour chaque variable non assignée. Par exemple, avec juste 100 variables non assignées, et 100 valeurs pour chaque variable, le produit cartésien est 100^{100}, qui est un très grand nombre. Pour éviter les problèmes de capacité et comme cette étude n'est qu'une estimation, nous proposons d'utiliser la somme des valeurs pour chaque variable non assignée, donc pour 100 variables non assignées et avec 100 valeurs pour chaque variable, l'estimation sera 10000.

4 Expérimentation

Pour valider l'approche utilisée dans cette étude, les expériences ont été effectuées en utilisant une machine parallèle Intel Xeon X5650 (2.67 GHz) (12 cœurs physiques) équipée de 48 Go de RAM, avec le système d'exploitation Linux. La version du solveur OR-Tools utilisée comme un moteur de recherche pour résoudre les problèmes de PPC est la version 2727. Tous les problèmes de PPC résolus sont modélisés en utilisant le format FlatZinc et sont issus du MiniZinc Challenge 2012 [11]. FlatZinc est un langage d'entrée de bas niveau utilisé par les solveurs de PPC, il est conçu pour faire une interface entre les problèmes et les solveurs [12]. Le but de ce Minizinc Challenge est de comparer les solveurs de PPC et les différentes méthodes de PPC utilisées pour résoudre le même problème. Dans cette expérience, nous avons résolu 11 problèmes de PPC. 6 sont des problèmes de satisfaction de contraintes et 5 sont des problèmes d'optimisation de contraintes. Les temps de calcul présentés dans cette section sont donnés en secondes et sont une moyenne de minimum de 3 exécutions sans utiliser un temps limite (timeout) pour l'exécution. La valeur du seuil de partitionnement utilisé par le Portfolio N To P est de 40%.

4.1 Pourquoi on utilise le Portfolio ?

Pour résoudre un problème de satisfaction de contraintes, comme le problème des N-Reines [11] de taille 35. L'utilisateur peut utiliser une *stratégie aléatoire* qui choisit aléatoirement une variable de branchement et pour chaque variable, elle assigne une valeur aléatoire. Pour trouver la première solution, la stratégie aléatoire génère un arbre de recherche qui contient 5.505.843 nœuds. Le temps d'exploration de ces nœuds en séquentiel est de 1024,72 secondes. Par contre, si l'utilisateur change la stratégie de recherche et choisit la *stratégie Min Bound*, qui sélectionne la plus petite valeur pour chaque variable non assignée, l'arbre de recherche généré contient un total de 341.593 nœuds et le temps d'exploration de ces nœuds en séquentiel est de 64.14 secondes.

Pour résoudre un problème d'optimisation de contraintes tel que le problème 2D Level Packing [11] (2DLevelPacking_class_1), afin de trouver la solution la plus optimale, l'utilisateur peut choisir la meilleure stratégie trouvée précédemment pour résoudre le problème des N-Reines qui est la stratégie Min Bound, cette stratégie génère un arbre de recherche de 2.745.714 nœuds et le temps de d'exploration de ces nœuds en séquentiel est de 43,90 secondes. D'autre part, si l'utilisateur choisit une autre stratégie de recherche, la *stratégie Impact Base search* [14],

basée sur l'impact des variables et de leurs valeurs, l'arbre de recherche généré contient 31.260 nœuds et le temps d'exploration de ces nœuds en séquentiel est de 1,92 secondes.

Il est donc clair que l'utilisation de la meilleure stratégie a un effet important sur la performance pour résoudre les problèmes de satisfaction et d'optimisation de contraintes. Par exemple, si l'utilisateur utilise la stratégie aléatoire avec 12 cœurs pour résoudre le problème N-Reines et on suppose qu'il a une accélération linéaire, le temps de calcul sera $1024,72/12 = 85{,}39$ secondes. Ce temps de calcul est plus grand que le temps de calcul obtenu en séquentiel en utilisant la stratégie Min Bound (64.14 secondes).

En utilisant un solveur Portfolio, nous pouvons être sûr que la performance obtenue est la même que la performance de la meilleure stratégie.

4.2 Les Performances Obtenues en Utilisant la Parallélisation Portfolio

Dans ce qui suit, l'accélération est calculé par rapport à un Portfolio *N* To *N*.

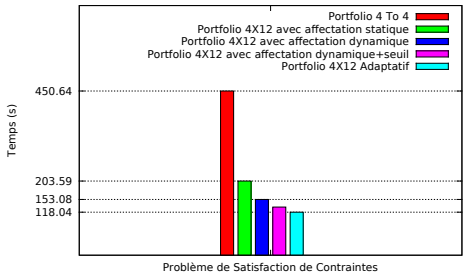


FIGURE 7 – Comparaison entre 5 modèles de Portfolio pour résoudre le problème de Naval Battle [11] (sb_sb_13_13_5_1f̂zn) qui est un problème de satisfaction de contraintes

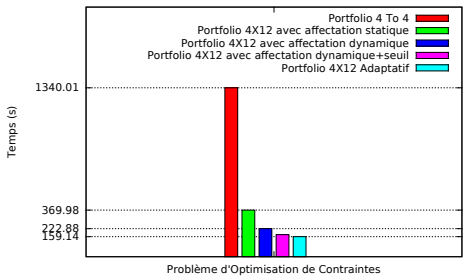


FIGURE 8 – Comparaison entre 5 modèles de Portfolio pour résoudre le problème de Pattern Set Mining [11] (pattern_set_mining_k1_germancredit) qui est un problème d'optimisation de contraintes

Les figures 7 et 8 montrent une comparaison entre les 5 modèles de Portfolio : N To N avec 4 stratégies et 4 cœurs, N To P avec 4 stratégies et 12 cœurs avec affectation statique, dynamique et dynamique avec un seuil de partitionnement. Le dernier modèle est le Portfolio N To P Adaptatif avec 4 stratégies et 12 cœurs. Cette comparaison est effectuée en résolvant le problème de Naval Battle [11] (sb_sb_13_13_5_1), qui est un problème de satisfaction de contraintes et le problème de Pattern Set Mining [11] (pattern_set_mining_k1_germancredit), qui est un problème d'optimisation de contraintes.

La performance des Portfolios N To N et N To P en utilisant une affectation statique est limitée, car ils ont une faible accélération par rapport aux différentes versions de Portfolio N To P avec affectation dynamique et le Portfolio Adaptatif. Le meilleur Portfolio est le Portfolio Adaptatif.

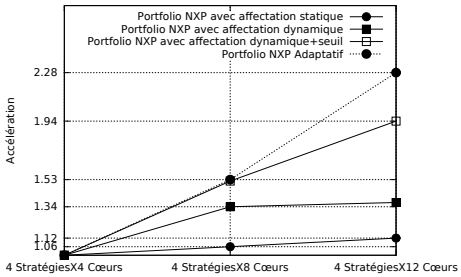


FIGURE 9 – Comparaison d'accélération moyenne pour résoudre 6 problèmes de satisfaction de contraintes en utilisant 4 modèles de Portfolio (4 stratégies de recherche pour chaque modèle)

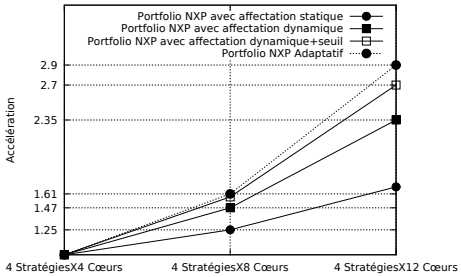


FIGURE 10 – Comparaison d'accélération moyenne pour résoudre 5 problèmes d'optimisation de contraintes en utilisant 4 modèles de Portfolio (4 stratégies de recherche pour chaque modèle)

Les figures 9 et 10 montrent une accélération moyenne pour résoudre 6 problèmes de satisfaction de contraintes et 5 problèmes d'optimisation de contraintes en utilisant 4 modèles de Portfolio N To P, chaque modèle de Portfolio utilise 4 stratégies de

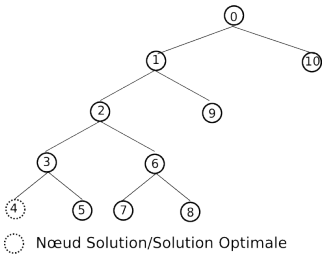


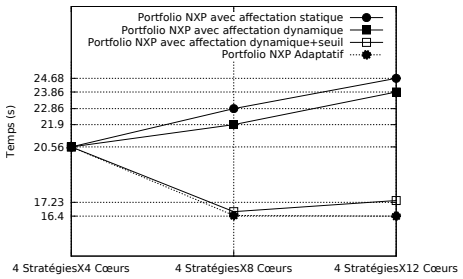
FIGURE 11 – Exemple d'arbre de recherche qui donne une mauvaise accélération

recherche et le nombre de cœurs varie entre 4, 8 et 12 cœurs.

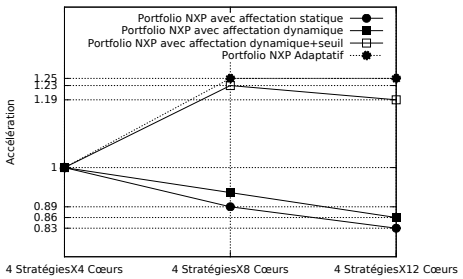
Parfois, le nœud solution est le nœud le plus à gauche dans l'arbre de recherche, tel que présenté dans la figure 11. Dans ce cas, le Portfolio N To P ne donne pas une bonne accélération, comme le montrent les figures 12 et 14. La figure 12 :a *(resp. 12 :b)* montre le temps de calcul *(resp. accélération)* pour résoudre le problème de Naval Battle(sb_sb_15_15_7_4) [11], qui est un problème de satisfaction de contraintes et qui utilise 4 stratégies de recherche pour chaque modèle de Portfolio. Le tableau de la figure 12 présente le nombre des nœuds OR-Tools visités par chaque modèle de Portfolio. La figure 13 montre une parallélisation de la meilleure stratégie pour résoudre le problème de Naval Battle (sb_sb_15_15_7_4). La mauvaise accélération vient de la parallélisation de l'arbre de recherche (anomalie de recherche).

La figure 14 :a *(resp. 14 :b)* montre le temps de calcul *(resp. accélération)* pour résoudre le problème de Pattern Set Mining (pattern_set_mining_k1_segment) [11], qui est un problème d'optimisation de contraintes en utilisant 4 stratégies de recherche pour chaque Portfolio. Le tableau de la figure 14 représente le nombre de nœuds OR-Tools visités par chaque modèle de Portfolio. La figure 15 montre une parallélisation de la meilleure stratégie pour résoudre le problème de Pattern Set Mining (pattern_set_mining_k1_segment). Le résultat est similaire au problème de satisfaction de contraintes, et la mauvaise accélération provient de la parallélisation de l'arbre de recherche (anomalie de recherche).

Il est possible de trouver un nœud solution au premier niveau de l'arbre de recherche, tel que présenté dans la figure 16. Dans ce cas, le Portfolio N To P donne une bonne accélération, comme le montrent les figures 17 et 18. La figure 17 :a *(resp. 17 :b)* montre le temps de calcul *(resp. accélération)* pour résoudre le problème de Naval Battle(sb_sb_13_13_5_1) [11], qui est un problème de satisfaction de contraintes. Le tableau de la figure 17 représente le nombre de nœuds OR-Tools visités par chaque modèle de Portfolio.



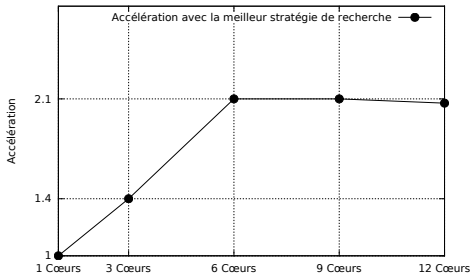
(a)



(b)

# Cœurs	Modèles de Portfolio				
	N To N	N To P			
	4 To 4	Statique	Dynamique	Dynamique avec seuil	Adaptatif
4	488,316	-	-	-	-
8	488,316	578,022	575,354	565,322	508,711
12	488,316	715,017	669,515	607,417	519,192

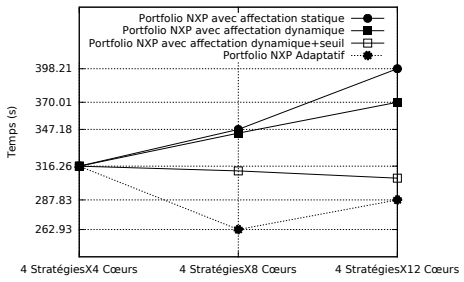
FIGURE 12 – Pire des cas (Worst case) pour résoudre le problème de Naval Battle(sb_sb_15_15_7_4) [11] qui est un problème de satisfaction de contraintes en utilisant 4 stratégies de recherche pour chaque Portfolio



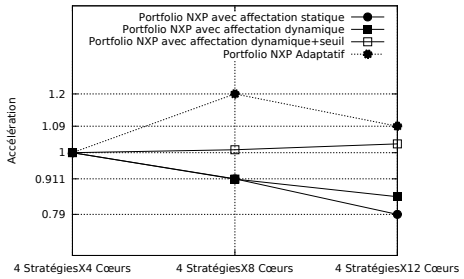
(a)

Number of nodes	Nombre des cœurs				
	1	3	6	9	12
	127,239	186,401	213,420	338,783	293,066

FIGURE 13 – Parallélisation de l'arbre de recherche généré par la meilleure stratégie pour résoudre le problème de Naval Battle (sb_sb_15_15_7_4) [11] qui est problème de satisfaction de contraintes



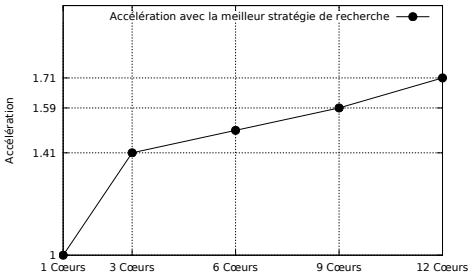
(a)



(b)

# Cœurs	Modèles de Portfolio				
	N To N	N To P			
	4 To 4	Statique	Dynamique	Dynamique avec seuil	Adaptatif
4	325,948	-	-	-	-
8	325,948	325,657	353,283	344,933	310,596
12	325,948	461,803	421,402	408,897	334,440

FIGURE 14 – Pire des cas (Worst case) pour résoudre le problème de Pattern Set Mining (pattern_set_mining_k1_segment) [11] qui est un problème d'optimisation de contraintes en utilisant 4 stratégies de recherche pour chaque Portfolio



(a)

Number of nodes	Nombre des cœurs				
	1	3	6	9	12
	82,281	109,000	135,548	137,828	141,819

FIGURE 15 – Parallélisation de l'arbre de recherche généré par la meilleure stratégie pour résoudre le problème de Pattern Set Mining (pattern_set_mining_k1_segment) [11] qui est un problème d'optimisation de contraintes

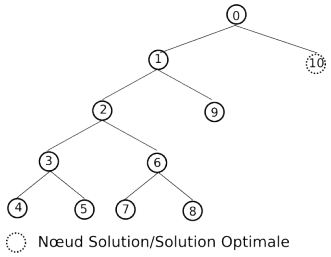
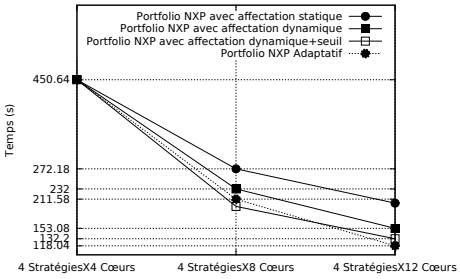
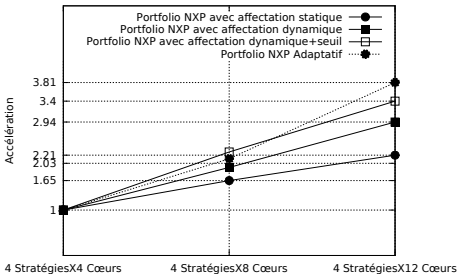


FIGURE 16 – Exemple d'arbre de recherche qui donne une bonne accélération



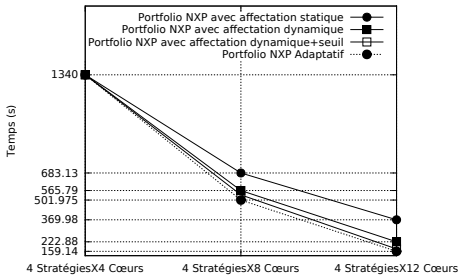
(a)



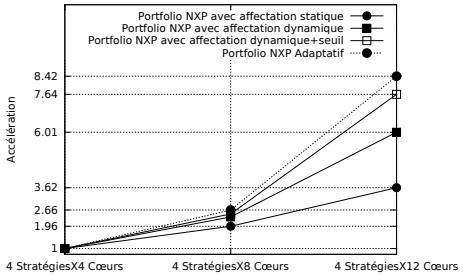
(b)

# Cœurs	Modèles de Portfolio				
	N To N	N To P			
	4 To 4	Statique	Dynamique	Dynamique avec seuil	Adaptatif
4	11,097,805	-	-	-	-
8	11,097,805	7,119,938	6,666,043	6,582,847	6,009,770
12	11,097,805	7,017,854	6,287,693	607,158	5,168,337

FIGURE 17 – Meilleur cas (Best case) pour résoudre le problème de Naval Battle(sb_sb_13_13_5_1) [11] qui est un problème de satisfaction de contraintes en utilisant 4 stratégies de recherche pour chaque Portfolio



(a)



(b)

# Cœurs	Modèles de Portfolio				
	N To N	N To P			
	4 To 4	Statique	Dynamique	Dynamique avec seuil	Adaptatif
4	3,925,159	-	-	-	-
8	3,925,159	1,633,120	1,596,446	1,535,643	1,517,335
12	3,925,159	333,404	306,790	267,996	232,213

FIGURE 18 – Meilleur cas (Best case) pour résoudre le problème de Patter Set Mining (pattern_set_mining_k1_germancredit) [11] qui est un problème d'optimisation de contraintes en utilisant 4 stratégies de recherche pour chaque Portfolio

La figure 18 :a *(resp. 18 :b)* montre le temps de calcul *(resp. accélération)* pour résoudre le problème de Pattern Set Mining (pattern_set_mining_k1_germancredit) [11], qui est un problème d'optimisation de contraintes. Le tableau de la figure 18 représente le nombre de nœuds OR-Tools visités par chaque modèle de Portfolio. Pour les deux problèmes, chaque modèle de Portfolio utilise 4 stratégies de recherche. Les figures 19 et 20 montrent le comportement des cœurs de calcul lorsque nous utilisons le Portfolio Adaptatif. Le résultat est que les temps de calcul et d'attente sont répartis de manière équitables entre les cœurs de calcul. De plus, tous les cœurs ont visité le même nombre de nœuds OR-Tools.

5 Conclusion et Perspectives

Cet article présente un Portfolio parallèle N To P pour résoudre les problèmes de satisfaction et d'optimisation de contraintes en utilisant le solveur OR-Tools au-dessus du framework parallèle Bobpp. Le meilleur Portfolio N To P est le Portfolio Adaptatif

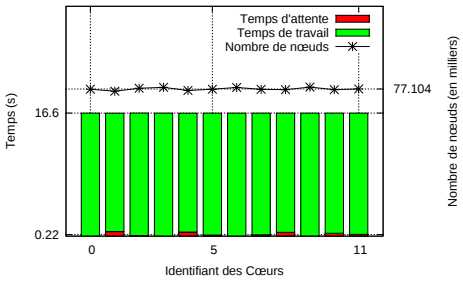


FIGURE 19 – Équilibrage de charge pour résoudre le problème de Naval Battle (sb_sb_15_15_7_0) [11] qui est un problème de satisfaction de contraintes avec un Portfolio Adaptatif de 4 stratégies et 12 cœurs

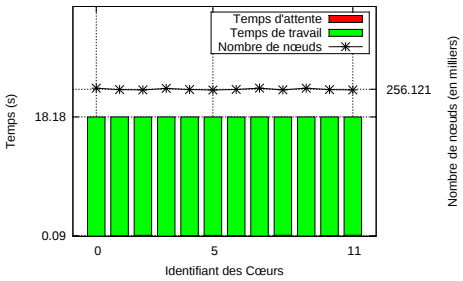


FIGURE 20 – Équilibrage de charge pour résoudre le problème de Open Stacks (open_stacks_01_problem_15_15) [11] qui est un problème d'optimisation de contraintes avec un Portfolio Adaptatif de 4 stratégies et 12 cœurs

qui privilège la stratégie la plus prometteuse.

Une première perspective est d'effectuer plusieurs expériences pour déterminer automatiquement la valeur du seuil de partitionnement utilisé par l'affectation dynamique en fonction des problèmes de PPC.

L'utilisation du framework parallèle Bobpp donne une bonne accélération sur des architectures à mémoire partagée. Ces résultats sont obtenus avec plusieurs types de problèmes d'optimisation combinatoire sur différents ordinateurs. Bobpp a également une parallélisation sur des architectures à mémoire distribuée version mixte MPI/Pthreads. Une deuxième perspective est de proposer une parallélisation distribuée pour le Portfolio Adaptatif.

Enfin, il serait intéressant de retourner à l'utilisateur toujours la même solution quel que soit le mode d'exécution. En séquentiel, c'est automatique, car c'est la première solution trouvée. Cependant, en parallèle la première solution trouvée n'est pas nécessairement la même solution que la solution retournée en séquentiel. Une dernière perspective serait de proposer un Portfolio Adaptatif déterministe qui retourne à l'utilisateur toujours la même solution si cette fonctionnalité est

demandée par l'utilisateur.

Références

[1] Arbelaez Alejandro, Hamadi Youssef, and Sebag Michèle. Online Heuristic Selection in Constraint Programming, 2009. International Symposium on Combinatorial Search - 2009.

[2] Bertrand Le Cun, Tarek Menouer, and Pascal Vander-Swalmen. Bobpp. `http://forge.prism.uvsq.fr/projects/bobpp`.

[3] e. o'mahony, Emmanuel Hebrard, Alan Holland, and Conor Nugent. Using case-based reasoning in an algorithm portfolio for constraint solving. In *IRISH CONFERENCE ON ARTIFICIAL INTELLIGENCE AND COGNITIVE SCIENCE*, 2008.

[4] Xie Feng and Davenport Andrew. Solving scheduling problems using parallel message-passing based constraint programming. In *Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems COPLAS*, pages 53–58, 2009.

[5] Francois Galea and Bertrand Le Cun. Bob++ : a framework for exact combinatorial optimization methods on parallel machines. In *International Conference High Performance Computing & Simulation 2007 (HPCS'07) and in conjunction with The 21st European Conference on Modeling and Simulation (ECMS 2007)*, pages 779–785, June 2007.

[6] Diarmuid Grimes and Richard J. Wallace. Sampling strategies and variable selection in weighted degree heuristics. In *13th International Conference on Principles and Practice of Constraint Programming 2007, Providence, RI, USA*, 2007.

[7] Joxan Jaffar, Andrew E. Santosa, Roland H. C. Yap, and Kenny Qili Zhu. Scalable distributed depth-first search with greedy work stealing. In *ICTAI*, pages 98–103, 2004.

[8] Stephan Kottler and Michael Kaufmann. SArTagnan - A parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*, 2011.

[9] Tarek Menouer and Bertrand Le Cun. Anticipated dynamic load balancing strategy to parallelize constraint programming search. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1771–1777, May 2013.

[10] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Transparent parallelization of constraint programming. *INFORMS JOURNAL ON COMPUTING*, 21(3) :363–382, 2009.

[11] Minizinc challenge `http://www.minizinc.org/challenge2012/`, 2012.

[12] NICTA. Specification of zinc and minizinc. Technical report, Victoria Research Lab, Melbourne, Australia, 2011.

[13] Laurent Perron. Search procedures and parallelism in constraint programming. *International Conference on Principles and Practice of Constraint Programming*, 1999.

[14] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP*, pages 557–571, 2004.

[15] Jean-Charles Regin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In *19th International Conference on Principles and Practice of Constraint Programming 2013 Uppsala Sweden*, 2013.

[16] Olivier Roussel. ppfolio. `http://www.cril.univ-artois.fr/~roussel/ppfolio/`.

[17] Vincent Vidal, Lucas Bordeaux, and Youssef Hamadi. Adaptive k-parallel best-first search : A simple but efficient algorithm for multi-core domain-independent planning, 2010.

[18] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla : Portfolio-based algorithm selection for sat. *J. Artif. Int. Res.*, 32(1) :565–606, June 2008.