



PROJET DE FIN D'ÉTUDES 2019 ECOLE NATIONALE DE L'AVIATION CIVILE, TOULOUSE,
FRANCE. VERSION OF SUNDAY 15TH SEPTEMBER, 2019 AT 12:16¹

<https://github.com/toulbar2/toulbar2/tree/kad2>

STAGE DU 18 MARS 2019 AU 15 SEPTEMBRE 2019 SUPERVISÉ PAR LES DR. SIMON DE GIVRY ET DAVID ALLOUCHE, CHERCHEURS À L'INRA, UNITÉ DE MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES DE TOULOUSE (MIAT), EQUIPE STATISTICS AND ALGORITHMICS APPLIED TO BIOLOGY (SAB), 24 CHEMIN DE BORDE-ROUGE, CS 52627, 31326 AUZEVILLE-TOLOSANE CEDEX FRANCE.

¹Le présent rapport est publié sur <https://github.com/kad15/SandBoxToulbar2/blob/master/kad/rapport.pdf>



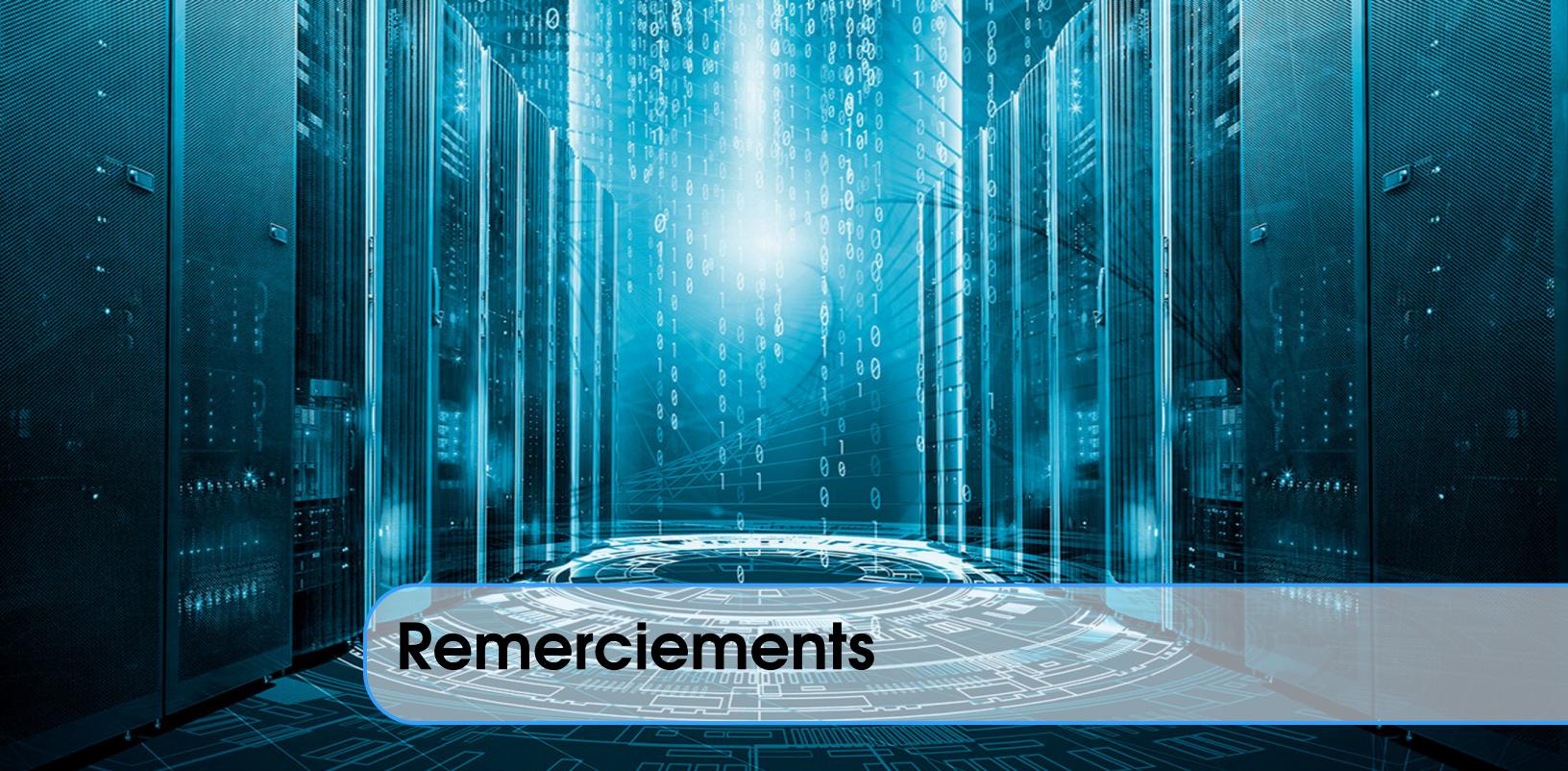
Preface

"If you build it, they will come."

And so we built them. Multiprocessor workstations, massively parallel supercomputers, a cluster in every department ... and they haven't come. Programmers haven't come to program these wonderful machines. Oh, a few programmers in love with the challenge have shown that most types of problems can be forcefit onto parallel computers, but general programmers, especially professional programmers who "have lives", ignore parallel computers.

And they do so at their own peril. Parallel computers are going mainstream. Multithreaded microprocessors, multicore CPUs, multiprocessor PCs, clusters, parallel game consoles... parallel computers are taking over the world of computing. The computer industry is ready to flood the market with hardware that will only run at full speed with parallel programs. But who will write these programs?

*Timothy Mattson , Beverly Sanders , Berna Massingill **Patterns for parallel programming**, Addison-Wesley Professional, 2004*



Remerciements

A l'heure d'entamer la rédaction de ce rapport, ce vendredi 2 août 2019, je souhaiterais remercier le centre de recherche de l'INRA Toulouse et son [unité de Mathématiques et Informatique de Toulouse \(MIAT\)](#) pour m'avoir accueilli durant ce [Projet de Fin d'Etudes \(PFE\)](#) de 6 mois, entre le 18 mars et le 15 septembre 2019, et plus spécifiquement mes superviseurs les Dr. Simon de Givry et David Allouche pour avoir proposé ce sujet et l'aide précieuse apportée durant ce stage exigeant.

Je remercie également :

- Marie-Stéphane Trotard et Didier Laborie, pour le support sur le *cluster*² de la plateforme bio-informatique du réseau GenoToul³,
- Fabienne Ayrignac et Alain Perault pour le support administratif,
- Damien Berry et Mikaël Grialou pour le support informatique et réseaux
- et, last but not least, les développeurs de la librairie Boost⁴ et plus particulièrement de Boost.MPI⁵ qui m'auront évité d'utiliser l'interface C de Open MPI⁶ dans un programme écrit en C++ ...

²Un **Cluster** est une grappe d'ordinateurs ou nœuds, relativement homogènes, connectés par un réseau local, propriété d'une entité déterminée et unique qui en assure la gestion. https://fr.wikipedia.org/wiki/Grappe_de_serveurs

³GenoToul est la contraction de Génopole Toulouse qui est le réseau toulousain de plateformes de recherche en sciences du vivant qui mutualise les ressources : <https://www.genotoul.fr/vie-de-genotoul/>. Dans la suite, on utilisera les termes cluster bio-informatique ou simplement cluster pour désigner cet équipement

⁴<https://www.boost.org/>

⁵https://www.boost.org/doc/libs/1_70_0/doc/html/mpi.html

⁶<https://www.open-mpi.org/>



Table des matières

Résumés	13
Sujet	25
Introduction	27
1 Environnement de travail	29
1.1 Présentation de la structure d'accueil	29
1.2 Rôle de l'équipe SAB	30
1.3 Ressources documentaires	31
1.3.1 Bases de données utilisées	31
1.3.2 Stratégie bibliographique associée	31
1.4 Ressources matérielles	31
1.5 Ressources logicielles	34
1.5.1 Systèmes d'exploitation	34
1.5.2 Gestion de version	34
1.5.3 Autres	35
1.5.4 Environnement de compilation/exécution	36
1.5.5 Environnement de développement	36

2	Aspects gestion de projet	39
2.1	Planning prévisionnel	39
2.1.1	Approche initiale pour traiter le sujet	40
2.2	Planning effectif du stage	40
3	Présentation des wcsp	43
3.1	Définitions	43
3.1.1	WCSP	43
3.1.2	Fonctions de coûts	44
3.1.3	Modèle initial	45
3.1.4	Fonction de coûts globale	45
3.1.5	Réseaux de fonctions de coûts	45
3.2	Formalisme des WCSP	47
3.2.1	Description succincte	47
3.2.2	Propagation de contraintes souples dans l'hypergraphe des fonctions de coûts	47
3.3	Exemple de modélisation	49
3.4	Exemple de projection	49
3.5	Exemple jouet : Le problème pondéré des 4 reines	52
4	Toulbar2	55
4.1	Généralités	55
4.2	Caractéristiques de toulbar2	56
4.3	Utilisation de toulbar2	57
4.4	Elément synoptique du code de toulbar2	59
4.4.1	Description succincte du code	59
4.4.2	Fonction général de toulbar2	60
5	Hybrid Best-First Search	61
5.1	Definitions	61
5.1.1	Depth-First Search	61
5.1.2	Breadth-First Search	61
5.1.3	Best-First Search	62
5.1.4	Noeud ouvert	62
5.1.5	Backtrack adaptatif	62
5.1.6	Hybrid Best-First Search	63

6 Stratégies de parallélisation	67
6.1 Introduction	67
6.2 Définitions	68
6.2.1 Processus	68
6.2.2 Thread	68
6.2.3 Socket	68
6.2.4 Parallelism	68
6.2.5 Instruction-Level Parallelism	69
6.2.6 Thread-Level Parallelism	69
6.2.7 Hyperthreading	69
6.2.8 Non Uniform Memory Access	70
6.2.9 Algorithme parallèle	70
6.2.10 Algorithme séquentiel sous-jacent	70
6.2.11 Unité d'exécution (UE)	71
6.2.12 Tâche	71
6.2.13 Worker	71
6.2.14 Master	71
6.2.15 Communicateur	71
6.2.16 Wallclock time	71
6.2.17 Speedup	72
6.2.18 Efficacité parallèle	72
6.2.19 Fraction séquentielle	72
6.2.20 Amdahl's law	72
6.2.21 Scalabilité ou échelonnabilité	72
6.2.22 Ramp-up, Ramp-down	72
6.2.23 Overhead ou surcoût	73
6.2.24 Granularité	73
6.2.25 Adaptivity	74
6.2.26 Partage de connaissances	74
6.2.27 Déterminisme	74
6.2.28 Synchronisation	74
6.2.29 Compromis ou Trade-off	75
6.2.30 Load balancing	75
6.3 Mémoire partagée contre mémoire distribuée	76
6.3.1 Solveurs existants	76
6.3.2 Choix pour toulbar2	76
6.4 Stratégies envisagées	77
6.4.1 Embarrassingly parallel search	77
6.4.2 Work stealing	78
6.4.3 Utilisation d'un framework	78
6.4.4 Paradigme Master-Worker	79

6.4.5	Autres paradigmes	80
6.5	Bilan	80
7	Embarrassingly Parallel Search	81
7.1	Généralités sur la méthode de parallélisation embarrassante	81
7.2	Implémentation de l'EPS	81
7.3	Génération a priori des sous-problèmes	82
7.4	Génération des sous-problèmes avec HBFS	83
7.5	Quelques expérimentations	86
7.5.1	Influence de la borne supérieure <i>cub</i>	86
7.5.2	Problème du nombre de problèmes à générer	86
7.5.3	Problème de la distribution des temps de résolution	86
7.5.4	Performances globales de la parallélisation	89
7.6	Bilan	90
8	Paradigme Master-Worker	93
8.1	Description du paradigme Master-Worker	93
8.2	Description de l'algorithme	95
8.2.1	Vue globale	95
8.2.2	Algorithm Master	96
8.3	Implémentation de l'algorithme	96
8.3.1	Bibliothèques utilisées	96
8.3.2	Utilisation du hbfs parallèle	97
8.3.3	Description synoptique	97
8.3.4	Algorithme Worker	98
8.4	Expérimentations	98
8.5	Comparaison entre EPS et Master-Worker	98
8.5.1	Comparaison avec des travaux similaires	101
8.6	Bilan	102
9	Conclusion	103
10	Annexes	105
10.1	Annexe A : Génération a priori des sous problèmes	105
10.2	Annexe B : Utilisation de toulbar2 avec Eclipse	110

10.3	Annexe C : Compilation sur le cluster genotoul	114
10.4	Annexe D : Utilisation de toulbar2 avec l'option eps	126
10.5	Annexe E : Résumé étendu pour la CP 2019	127
	Acronymes	131
	Glossaire	133
	Bibliography	137
	Articles	137
	Books	139



Résumés

Abstract

The algorithm [Hybrid Best-First Search \(HBFS\)](#) combines a Bounded recursive Depth-First Search (DFS) Branch and Bound (B&B) with a Best-First Search in order to find a complete assignment of the minimization combinatorial problem variables. The problem can be seen as an hypergraph which represents a Cost Function Network, where the nodes are the variables and the edges the soft constraints, according to the [Weighted Constraint Satisfaction Problems \(WCSP\)](#) formalism.

The local soft constraints are combined to compute a global cost function. [Equivalent Preserving Transformations \(EPT\)](#) are then applied to the latter in order to get a *good* local lower bound in polynomial time which is used in the B&B in the same way, a relaxation gives a local lower bound in [Integer Linear Programming \(ILP\)](#), used to prune the decision tree. In HBFS, a current global lower bound (clb) is computed together with the Global Upper Bound (cub) which is the current best value, or *incumbent* value, associated with the best complete assignment or best solution found so far. The interval $[clb, cub]$ is called the *optimality gap*.

Part of the C++ solver [toulbar2](#), HBFS is still a sequential algorithm. As such, it does not get the best out of nowadays parallel architectures. To tackle this issue, we describe the phases that lead us to a parallelized release of HBFS.

First we present, a simplified overview of the underlying theory. Second, we examine two trails: the [Embarrassingly Parallel Search \(EPS\)](#), also known as natural parallelization, and the [Master-Worker](#) paradigm, popular among the [High Performance Computing \(HPC\)](#) community. The tests are performed on desktops, servers and/or on the cluster's data center [GenoToul](#) of the French Agronomic Research Institute of Toulouse (INRA).

Keywords: tree search, toulbar2, combinatorial optimization, branch and bound, hypergraph, hyper-graph, local soft consistency, depth first search, best first search, weighted

constraint satisfaction problem, cost function networks, optimality gap, incumbent value, incumbent solution, clusters, data center, high performance computing, parallelism, embarrassingly parallel search, Master-Worker paradigm, message passing interface.

Résumé

L'algorithme hybride de recherche par meilleur nœud d'abord, ou [Hybrid Best-First Search \(HBFS\)](#), combine un algorithme par séparation et évaluation, ou Branch and Bound, parcourant récursivement en profondeur l'arbre de recherche avec une sélection des nœuds *ouverts* les plus prometteurs d'abord. Un nœud ouvert est un sous-problème, i.e. un sous arbre, qui n'a pas encore été exploré. Ces nœuds ouverts sont classés dans une file de priorité *open* selon leurs bornes inférieures locales croissantes. En cas d'égalité, le classement se fait selon leurs profondeurs décroissantes puisque le nœud le plus profond est supposé être plus proche des feuilles de l'arbre de décisions donc plus à même de fournir rapidement une solution améliorante.

L'objectif de cette recherche arborescente est de trouver une affectation complète des variables d'un réseau de fonctions de coûts, représenté par un hypergraphe dont les nœuds sont les variables et les arêtes les contraintes et qui minimise la fonction de coûts globale du problème. Cette dernière combine des contraintes locales, dures et souples, selon le formalisme des problèmes de contraintes souples pondérées (WCSP).

Cette fonction de coûts globale subit des transformations en temps polynomiale qui la préservent tout en faisant évoluer l'hypergraphe de manière à exhiber le plus grand minorant possible : on dit que le problème wcsp subit une séquence de transformations préservant l'équivalence ou [Equivalent Preserving Transformations \(EPT\)](#).

Cette borne inférieure locale *lb* est alors utilisée dans l'algorithme d'évaluation et séparation pour élaguer l'arbre de recherche et calculer le minorant global courant noté *clb* de manière à avoir, presque à tout instant, un encadrement noté $[clb, cub]$ de la valeur optimale recherchée de la fonction de coût globale. L'intervalle $[clb, cub]$ est appelé le *gap d'optimalité*. La plus petite des bornes inférieures locales *lb* parmi les nœuds *frontière* fournit la borne inférieure globale courante *clb*. La borne supérieure globale *cub* désigne la meilleure valeur courante ; c'est la valeur retournée par la fonction de coûts globale à laquelle on applique la meilleure affectation complète coutante des variables. Cette meilleure affectation complète est appelée solution titulaire ou *incumbent solution*. On distinguera la valeur optimale qui est un scalaire de la solution optimale qui est un tuple ou un vecteur. La meilleure valeur courante *cub* (current upper bound) est la valeur la plus petite trouvée jusque là puisqu'on cherche à minimiser le coût global du wcsp.

L'implémentation séquentielle, déjà bien optimisée, de HBFS dans le solver C++ [toulbar2](#) n'exploite pas les architectures parallèles actuelles et ne bénéficie donc pas de ce gisement de gain en performances. Pour y remédier, autant que faire ce peut, ce rapport présente donc les phases qui ont conduit à proposer une parallélisation de HBFS.

On présente d'abord un état de l'art simplifié, puis l'exploration de deux pistes : la parallélisation naturelle, connue sous le terme pittoresque de [Embarrassingly Parallel Search \(EPS\)](#) et l'utilisation du paradigme [Master-Worker](#), populaire dans le domaine du calcul haute performance, accompagnées des tests et résultats sur PC, serveurs et/ou cluster en data center de l'INRA Toulouse.

Mots-clés : optimisation combinatoire, séparation et évaluation, parcours en profondeur d'abord, parcours en meilleur nœud d'abord, gap d'optimalité, hypergraphe, réseau de

fonctions de coûts, satisfaction de contraintes pondérées, cohérence locale souple, valeur titulaire, parallélisation embarrassante, paradigme Master-Worker, solution titulaire, toulbar2, recherche arborescente, parallélisme, calcul haute performance, clusters, data center, message passing interface.



Table des figures

1.1	Vue arrière d'une des séries de baies du data center de l'INRA Toulouse. Consommation électrique annuelle : 100 000€. Chaque baie accueille des racks contenant plusieurs nœuds. Chaque nœud possède sa propre RAM et tourne sous sa propre instance de système d'exploitation en l'occurrence CentOS 7. Ils partagent un même système de fichier.	33
1.2	Vue de face d'une baie : chaque nœud du nouveau cluster SLURM genologin est constitué de 32 cœurs (64 threads) répartis sur deux puces de 16 cœurs montées sur une carte-mère demi-largeur équipée de 256GB de RAM.	34
3.1	Hypergraphe du problème des 4 reines : 4wqueens.wcsp. Le problème est représenté par un simple graphe car l'arité maximale des contraintes est égale à 2. Nous avons donc d'une part, le graphe représentant le problème, et d'autre part, l'arbre de recherche, ou arbre de décision, développé progressivement pour explorer l'espace d'états ici de taille $4^4 = 256$, valeur qui correspond au cardinal du produit cartésien des domaines des variables. Ici quatre variables : V1 à V4 de domaine identique $D = \{1, 2, 3, 4\}$. V4 = 2 signifie qu'on place une reine en colonne 4 et ligne 2. Dans ce type de graphe, la micro-structure n'apparaît pas ; les domaines ne sont pas représentés dans chaque nœud donc les fonctions de coûts unaires n'apparaissent pas non plus.	46
3.2	Hypergraphe du problème 404.wcsp. Problème-type utilisé lors des développements de hybridSolvePara(). On note qu'il présente clairement une structure en clusters.	46
3.3	Hypergraphe du problème scen06.wcsp	46

3.4	Exemple de graphe avec micro-structure d'un problème simple à deux variables de même domaine $\{a, b\}$. Par convention, les contraintes de coût nul ne sont pas représentées. Ainsi la contrainte unaire w_1 sur x_1 est telle que $w_1(b) = 0$, la contrainte binaire w_{12} est telle que $w_{12}(a, a) = 1$ et $w_{12}(a, b) = 0$. Si $k > 1$, le graphe indique que l'on préfère éviter le cas (a, a) . Si $k = 1$, ce cas est interdit.	47
3.5	Trois types de transformations de base. Initialement, figure de gauche, on a $w_\emptyset = C_\emptyset = 0$. La variable z peut prendre la valeur a dont le coût est 3 ou la valeur b dont le coût est 2. Cela donne une représentation "tabulaire" On peut donc opérer une projection unaire en transférant un coût égal à $2 = \min(2, 3)$ de la fonction de coûts unaire vers la fonction de coût d'arité nulle w_\emptyset	48
3.6	Micro-structure d'un réseaux de fonctions de coûts.	50
3.7	Nouveau réseau obtenu par projection unaire à partir de la 3.6.c. La contrainte constante est désormais $w_\emptyset = 1$. Ce qui constitue un minorant de l'optimum de la fonction de coûts globale.	51
3.8	Le problème des 4 reines.	52
5.1	Arbre binaire de décision partiellement exploré par un DFS avec une limite de backtrack, $Z = 3$. Les nœuds verts sont les feuilles de l'arbre donnant lieu à une affectation complète des variables. Les nœuds jaunes non cerclés sur les branches droites de l'arbre sont les nœuds ouverts i.e. le sous arbre associé n'a pas encore été exploré. Ces nœuds sont placés dans la file <i>open</i> par le DFS lorsqu'il atteint la limite $Z = 3$. En rouge, ce sont les nœuds fermés. Les nombres indiquent l'ordre de parcours DFS de l'arbre.	62
5.2	Illustration de l'algorithme : un nœud est retiré de la file <i>open</i> , donné au DFS qui retourne des nœuds ouverts dans la file <i>open</i> , qui les classe par clb croissants. Un vecteur de décision, i.e. de points de choix, optimise l'espace mémoire en évitant les redondances. En effet, deux nœuds peuvent avoir en commun un même chemin dans l'arbre de décisions. Il est plus économique en mémoire d'utiliser deux index, first et last, qui pointent sur le chemin du nœud que de mémoriser dans chaque nœud le chemin complet.	65
7.1	Un temps minimum d'exécution obtenu pour une partition en 200 sous-problèmes.84	
7.2	Zoom de la figure 7.1 : un temps minimal obtenu entre 230 et 260 sous-problèmes.	85
7.3	Maximum speedup observé pour des valeurs entre 28 et 33 sous-problèmes par cœur.	85
7.4	Temps de résolution en secondes pour les 208 sous-problèmes du problème 404.wcsp classés par temps de résolution croissant. La grande majorité des processus terminent en une fraction de seconde. Les processus les plus gourmands terminent en un peu moins de 10s. Ici, le <i>cub</i> n'a pas été utilisé. . . 87	

7.5	Distibution du log10 des critères de complexité. Le type <i>long double</i> a dû être utilisé pour éviter le dépassement de capacité des <i>long long int</i> . C'est pourquoi le critère est un flottant.	88
7.6	Temps d'exécution en fonction du critère de complexity des sous-problèmes. En dessous du seuil 33, on peut inférer que le problème est facile mais au-dessus on ne peut rien conclure.	89
8.1	HBFS Parallélisé : le master fournit le nœud 1 au worker qui lance un B&B dans le sous-arbre binaire de recherche associé à ce nœud. Les bornes inférieures sont propagées dans les nœuds enfants et servent à l'élagage de l'arbre. Le worker transmet au master les nœuds ouverts dès que le nombre maximum de backtracks, ici égal à 3, est atteint.	95



Liste des tableaux

3.1	Représentation tabulaire de la contrainte dure binaire w_{12}	49
7.1	Ce tableau présente les temps de génération en secondes des sous problèmes sur un serveur 24 cœurs ainsi que les temps séquentiels et parallèles. Le speed-up total tient compte du temps de génération. Les problèmes nug12 et 404 montrent l'influence du nombre sous problèmes sur les speed-ups.	90
8.1	Speed-up S et Efficacité E pour le Master-Worker(M-W) et l' <i>Embarrassingly Parallel Search</i> (ESP). Expérimentations faites sur serveur 24 cœurs. Le nombre de variables n et la taille max. des domaines d donnent une idée de la complexité du problème.	99
8.2	Exécutions sur serveur 24 cœurs. Le temps CPU est utilisé pour le calcul des speed-ups. La colonne Parallel time indique le temps CPU total. La colonne pre time indique la durée du préprocessing.	99
8.3	Benchmark warehouse [16] où n désigne le nombre de variables du problème et d la taille du plus grand domaine de ces variables.	100
8.4	Benchmarks <i>Computational Protein Design</i> (CPD) [22]. Les tirets indiquent que le problème n'a pu être résolu en moins d'une heure.	101
8.5	Benchmark <i>linkage</i> [8] avec divers nombre de cœurs. Entre parenthèses, sont indiqués les speed-ups.	101
8.6	Autre benchmark CPD [2]. Un tiret indique que la méthode n'est pas parvenue à prouver l'optimalité en moins de 9000s.	102



Liste des algorithmes

1	Hybrid Best-First Search. Initial call : HBFS(w_\emptyset, k) with $Z = 1$	66
2	Parallel Hybrid Best-First Search : Master	96
3	Parallel Hybrid Best-First Search : Worker	98



Contexte

Le cadre générique des réseaux de fonctions de coûts (Cooper, 2010) permet de résoudre des problèmes d'optimisation combinatoire variés. Il s'appuie sur des travaux menés en Intelligence Artificielle dans la communauté de la programmation par contraintes. Depuis une dizaine d'année, la vitesse des processeurs n'augmentant plus, plusieurs méthodes exactes d'optimisation ont exploité les nouvelles architectures multi-coeurs permettant une parallélisation à mémoire partagée et offrant des gains de performance importants. C'est notamment le cas des outils de la programmation linéaire en nombre entiers tels que IBM ILOG cplex et FICO Xpress. L'équipe SAB mène des travaux en optimisation combinatoire dans les sciences du vivant et développe un outil C++ d'optimisation qui a remporté plusieurs compétitions sur les modèles graphiques probabilistes (toulbar2 : <http://www7.inra.fr/mia/T/toulbar2/>).

Sujet

L'objectif du stage est d'étudier la parallélisation d'une méthode de recherche arborescente hybride développée dans l'équipe [3] et récemment étendue dans [13]. Plusieurs stratégies de parallélisation seront considérées et analysées avant de faire des choix de parallélisation et d'implémentation ([17], [1], [26], [14], [22], [25]). Une comparaison à un travail similaire dans le cadre des modèles graphiques [14] sera menée, ainsi que sur des problèmes issus de la recherche opérationnelle.



Bien que généralisées depuis les années 2000, les architectures parallèles présentent toujours un certain décalage avec les habitudes de programmation. Les algorithmes séquentiels ne bénéficient plus d'une amélioration automatique de leur performances avec l'augmentation de la fréquence des processeurs qui plafonne à quelques Giga Hertz (GHz). Le mur *thermique*, voire *quantique*, est passé par là. Le "*free lunch*" est terminé.

L'objectif du stage⁷ est d'étudier la parallélisation d'une méthode de recherche arborescente hybride développée par l'équipe Statistiques et Algorithmique pour la Biologie (SAB) de l'INRA Toulouse et baptisé **Hybrid Best-First Search (HBFS)**; HBFS est un des algorithmes utilisé dans un solveur C++ de réseaux de contraintes *souples* performant : **toulbar2**. HBFS, contrairement aux algorithmes implémentant des méta-heuristiques : *tabu search*, recuit simulé, génétiques, à population, etc., est de fournir, quand elle existe, une solution exacte avec en outre sa preuve d'optimalité. Il permet de résoudre des problèmes de satisfaction de contraintes souples(WCSP) qui généralisent les CSP.

Un état de l'art sur les wcsp et le parallélisme est d'abord présenté⁸. Puis, plusieurs stratégies de parallélisation sont considérées qui ont donné lieu à deux choix de parallélisation et d'implémentation : l'**Embarrassingly Parallel Search (EPS)** et le paradigme **Master-Worker**.

⁷Le présent rapport est publié sur <https://github.com/kad15/SandBoxToulbar2/blob/master/kad/rapport.pdf>

⁸Un résumé est rédigé en tête de chaque chapitre pour en faciliter, ou en éviter la lecture. En effet, ce rapport a aussi été rédigé pour éventuellement servir d'outil de prise en main de toulbar2 et des formalismes associés.

Enfin, Les résultats des tests sont présentés qui vont confirmer ou non l'intérêt des pistes de parallélisations choisies ...



1. Environnement de travail

Résumé

Ce Projet de Fin d'études s'est déroulé dans l'[équipe Statistiques et Algorithmique pour la Biologie \(SAB\)](#) au sein de l'[unité de Mathématiques et Informatique de Toulouse \(MIAT\)](#), UR0875, sur le site de l'[Institut National de Recherche Agronomique \(INRA\)](#) Toulouse, située au 24 chemin de Borde-Rouge CS 52627 31326 Auzeville-Tolosane, secretariat-miat@toulouse.inra.fr, +33 (0)5 61 28 50 72.

1.1 Présentation de la structure d'accueil

Au même titre que le CNRS ou encore l'INRIA, l'[Institut National de Recherche Agronomique \(INRA\)](#) possède le statut d'[Etablissement Public à Caractère Scientifique et Technologique \(EPST\)](#) ce qui le range dans la catégorie des Etablissement Public à Caractère Administratif (EPA) lui permettant ainsi d'équilibrer son budget via des ressources non étatiques ; un EPA est, en effet, habilité à facturer des services à ses clients. C'est notamment le cas des prestations fournies par la plateforme bio-informatique, GenoToul.

En 2018, les ressources de l'INRA se montaient à 905 M€ dont 77% de subventions de l'Etat. De fait, le Ministère de tutelle de ce type d'établissement assure, via un contrat d'objectifs et de performance (COP), le suivi des orientations stratégiques de ces derniers en veillant à ce que leurs actions s'inscrivent dans les politiques publiques auxquelles ils participent. Généralement, un COP est établi par chaque établissement pour une période de 3 ans. Pour davantage d'information, on pourra télécharger ici le [rapport d'activité 2018](#).

L'INRA comprend 13 départements scientifiques parmi lesquels le [département de Mathématiques et Informatique \(MIA\)](#)¹. Ce dernier se décline en plusieurs unités dont celle de Toulouse, l'[unité de Mathématiques et Informatique de Toulouse \(MIAT\)](#)², anciennement

¹<http://www.mia.inra.fr/>

²<https://mia.toulouse.inra.fr/Accueil>

dénommée Station de Biométrie et Intelligence Artificielle, située sur le site de Castanet-Tolosan/Auzeville-Tolosane.

Le centre Inra Occitanie-Toulouse³, présidé par Michèle Marin, est partie prenante du projet ANITI (Artificial and Natural Intelligence Toulouse Institute) sur l'intelligence artificielle hybride initié suite au rapport⁴ Villani (2018).

L'**unité de Mathématiques et Informatique de Toulouse (MIAT)**, dirigée par Sylvain Jasson, est organisée en équipes parmi lesquelles, on trouve l'**équipe Statistiques et Algorithmique pour la Biologie (SAB)** dans laquelle s'est déroulé ce projet de fin d'études.

Fusion INRA-IRSTEA en 2020

Au 1er janvier 2020, suite à sa fusion avec l'**Institut National de Recherche en Sciences et Technologies pour l'Environnement et l'Agriculture (IRSTEA)**, ex-CEMAGREF, l'**INRA** deviendra l'**Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE)**. Le **département de Mathématiques et Informatique (MIA)** sera inclus dans une structure qui aura pour domaine la science des données, l'intelligence artificielle, les technologies robotiques et capteurs, la modélisation de systèmes complexes.

1.2 Rôle de l'équipe SAB

L'**équipe Statistiques et Algorithmique pour la Biologie (SAB)**, animée par Simon de Givry⁵, a pour objectif de développer et de mettre à disposition des biologistes des méthodes mathématiques, statistiques et informatiques permettant de contribuer à la compréhension du vivant. Elle s'intéresse à la localisation et à l'identification d'éléments fonctionnels dans les génomes des bactéries, plantes et animaux, et aux interactions qui existent entre ces différents éléments. Pour traiter ces problèmes, l'équipe mobilise et développe des méthodes en mathématiques, statistiques, probabilités (modélisation, inférence, modèles de mélanges de lois, régression pénalisée, modèles graphiques stochastiques, processus) et en informatique (modélisation, optimisation combinatoire, réseaux de contraintes, modèles graphiques déterministes, algorithmique) en vue de valoriser les méthodes développées dans des outils logiciels directement utilisables par les partenaires biologistes et rendant compte le mieux possible de la complexité et de la variété des données utilisables tout en capitalisant les développements méthodologiques dans des logiciels génériques, éventuellement déclinés ensuite sur différentes applications.

L'équipe développe en particulier des méthodes originales dans le domaine de l'optimisation combinatoire, en s'appuyant sur les réseaux de contraintes pondérées, aussi appelés "réseaux de fonctions de coûts", un modèle graphique dédié à l'optimisation et généralisant les réseaux de contraintes utilisés en programmation par contraintes et proches des Champs de Markov. Ces techniques, implémentées dans l'outil toulbar2 développé dans l'équipe, et très bien placé dans différentes compétitions internationales, sont ensuite mises en œuvre sur

³<http://www.toulouse.inra.fr/Toutes-les-actualites/ANITI>

⁴https://www.aiforhumanity.fr/pdfs/9782111457089_Rapport_Villani_accessible.pdf

⁵<http://www7.inra.fr/mia/T/degivry/>

des problèmes issus de la bio-informatique (conception de protéines, localisation d'ARNs de familles connues, diagnostics de pedigrees complexes de grande taille...).

1.3 Ressources documentaires

1.3.1 Bases de données utilisées

1. Bibliothèque ENAC
2. Bibliothèque Université Paul Sabatier Toulouse III
3. https://doc.lagout.org/science/0_Computer%20Science/5_Parallel%20and%20Distributed/
4. <https://login.proxy.lib.enac.fr/login>. Ce site d'accès aux ressources de la bibliothèque de l'ENAC rassemble un ensemble de bases de données et permet de télécharger des articles parfois payant en dehors de cet abonnement académique d'où la nécessité d'un dépôt documentaire privé sur [GitLab](#)
5. <https://link-springer-com.proxy.lib.enac.fr/>,
6. <https://dl.acm.org.proxy.lib.enac.fr/>
7. <https://scholar.google.fr/>
8. <https://www.lilo.org/fr/>

1.3.2 Stratégie bibliographique associée

La bibliographie relative aux [Weighted Constraint Satisfaction Problems \(WCSP\)](#) est conséquente. En outre, d'autres formalismes existent dont l'idée est de prendre en compte un réalité floue ou non déterministe. En la matière, l'exhaustivité aurait été contre productive vis à vis de l'objectif du stage, en l'occurrence améliorer les performances de l'algorithme séquentiel [Hybrid Best-First Search \(HBFS\)](#) de toulbar2 pour tenter de résoudre des problèmes non résolus ou des instances plus difficiles de problèmes connus. La bibliographie sur les WCSP et les techniques de parallélisation est basée sur un sous-ensemble d'articles sélectionnés par les encadrants du stage complétée par des recherches personnelles d'articles et de manuels sur internet ou en bibliothèque. Certains articles ont été présentés et discutés lors de réunions organisées régulièrement à l'INRA Toulouse.

1.4 Ressources matérielles

Sur le plan des ressources matérielles, un PC de bureau performant à 4 processeurs réels, 8 avec le multithreading, équipé de 32 Go de RAM a permis de tester localement toulbar2 tout en permettant un développement confortable sous l'environnement de développement Eclipse.

Par ailleurs, 4 serveurs équipés de 16 ou 24 cœurs, ont servi à diversifier les tests même s'ils étaient souvent monopolisés par d'autres utilisateurs.

Enfin, le data center "GenoToul", à finalité bioinformatique du site avec ses 3054 cœurs, 6218 en prenant en compte l'hyperthreading, connectés par réseau à haut débit et

faible latence **InfiniBand**, pourra le cas échéant être mis à contribution sous réserve de disponibilité⁶.

La nouvelle tranche du cluster **genologin** comprend, entre autres, 48 nœuds⁷, 32 coeurs et 256 GB de RAM par nœud. Le Workload manager SGE a été remplacé par **Simple Linux Utility for Resource Management (SLURM)**⁸. Total : 1584 coeurs / 3168 threads / 51 TFlops de stockage. Les processeurs ont une microarchitecture broadwell⁹, gravure 14 nanomètres. SLURM est limité à quelques 20000 processus lancés en parallèle. Ainsi, pour exécuter la version parallèle de toulbar2, en mode non batch¹⁰, sur des cpu à micro-architecture broadwell, on pourra réserver 3 nœuds pour exécuter 96 processus avec la commande suivante :

```
srun -N 3 --ntasks-per-node=32 -n 96 --time=1 --exclusive  
--constraint=broadwell ./toulbar2 404.wcsp -para
```

nb : --time=1 indique qu'au bout de 1 minute le processus sera détruit. La résolution est la minute : si on indique des secondes, le système fera un arrondi à la minute supérieure. Cette fonctionnalité, outre le fait de permettre une utilisation parcimonieuse des ressources, permet de constituer des benchmarks pertinents sur la base d'un time-out prédéfini.

Pour utiliser l'ancienne micro-architecture, on remplacera broadwell par ivy. Les performances sur l'architecture ivy, censée être moins performante, a montré au contraire un léger gain en performances.

En mode batch, on peut créer un script essai.sh

```
#!/bin/bash  
#SBATCH -J mpi_job  
#SBATCH --ntasks=64  
#SBATCH --cpus-per-task=2  
#SBATCH --mem-per-cpu=8G  
#SBATCH --exclusive  
#SBATCH --time=1  
  
mpirun -n $SLURM_NTASKS ./toulbar2 404.wcsp -para
```

puis utiliser la commande slurm : sbatch essai.sh pour soumettre le job au cluster.

Ces outils, PC, serveurs et data center, sont accessibles via une connexion VPN assurant l'ubiquité nécessaire notamment au télétravail des personnels de l'INRA ou aux tests à distance.

⁶<http://bioinfo.genotoul.fr/>

⁷hostnames du node101 à node148

⁸<https://slurm.schedmd.com/>

⁹[https://en.wikipedia.org/wiki/Broadwell_\(microarchitecture\)](https://en.wikipedia.org/wiki/Broadwell_(microarchitecture))

¹⁰<https://slurm.schedmd.com/sbatch.html>



FIGURE 1.1 : Vue arrière d'une des séries de baies du data center de l'INRA Toulouse. Consommation électrique annuelle : 100 000€. Chaque baie accueille des racks contenant plusieurs nœuds. Chaque nœud possède sa propre RAM et tourne sous sa propre instance de système d'exploitation en l'occurrence CentOS 7. Ils partagent un même système de fichier.



FIGURE 1.2 : Vue de face d'une baie : chaque nœud du nouveau cluster SLURM genologin est constitué de 32 cœurs (64 threads) répartis sur deux puces de 16 cœurs montées sur une carte-mère demi-largeur équipée de 256GB de RAM.

1.5 Ressources logicielles

Toutes les applications utilisées durant ce stage sont des [Free Open Source Software \(FOSS\)](#).

1.5.1 Systèmes d'exploitation

- Ubuntu 18 LTS sur PC de bureau avec droit administrateurs (sudo) permettant la configuration du système sans passer par les administrateurs,
- Ubuntu 19 sur portable,
- Debian 10 sur les serveurs,
- CentOS 7 sur le cluster.

1.5.2 Gestion de version

- Dépôt sur GitHub avec droit d'écriture une branche personnelle : <https://github.com/toulbar2/toulbar2/tree/kad>

- Fork de toulbar2 sur GitHub. Utilisé comme bac à sable en début de stage : <https://github.com/kad15/SandBoxToulbar2>
- Dépôt sur GitLab privé pour gérer les documents relatifs au stage : articles, rapport, résultats, etc. GitHub ne permet pas la création sans abonnement de dépôts privés et placer des documents parfois sous copyright dans un dépôt public est à éviter : <https://gitlab.com>.

Conseil :

1. cloner les dépôts GitHub et GitLab en ssh i.e. adresse de type git@github.com :toulbar2/toulbar2.git et non en https et copier sa clé ssh publique sur GitHub (cf. annexe 10.2). Cela permettra d'éviter la saisie systématique d'un user et lot de passe.
2. Utiliser un script-maître et des script secondaire pour mettre à jour les dépôts en un clic :

```
script maître :
#!/bin/bash
echo "pfe:"
git pull origin kad 1>/dev/null 2>&1
git add . 1>/dev/null 2>&1
git commit -m "divers" 1>/dev/null 2>&1
git push origin kad 1>/dev/null 2>&1
git push origin kad | grep "Everything up-to-date"
echo "toulbar2 branch kad:"
cd ../toulbar2 && ./kad_maj_git.sh

script secondaire :
#!/bin/bash
git pull origin kad 1>/dev/null 2>&1
git add . 1>/dev/null 2>&1
git commit -m "master-worker" 1>/dev/null 2>&1
git push origin kad 1>/dev/null 2>&1
git push origin kad 1>/dev/null 2>&1
git push origin kad | grep "Everything up-to-date"
sleep 4s
```

1.5.3 Autres

- Outils graphiques : GraphViz¹¹, gnuPlot¹², R/RStudio¹³, LibreOffice Draw,
- Bureautique : TeXstudio <https://www.texstudio.org/>, Foxit : <https://www.foxitsoftware.com/>,
- Saros - Distributed Collaborative Editing and Pair Programming Eclipse plugin : <https://www.saros-project.org/>. Outils pour effectuer des développe-

¹¹<https://graphviz.org/>

¹²<http://www.gnuplot.info/>

¹³<https://www.rstudio.com/>

ment collaboratifs en temps réel. Plugin eclipse, Saros installé mais non utilisé. Voir aussi <https://vimeo.com/1195398>.

1.5.4 Environnement de compilation/exécution

Sur PC et serveurs

Les autotools¹⁴ bien connus sous GNU/Linux, ne sont pas utilisés pour compiler le code. A la place, toulbar2 utilise le système de build cmake¹⁵, avec son configurateur ccmake. En fonction de la plateforme cible ou lors d'ajout de bibliothèques ou options, les fichiers cmake doivent être modifiés, en particulier pour prendre en compte les bibliothèques Boost.serialization¹⁶ et Boost.MPI¹⁷ ou accéder à des bibliothèques qui ne se trouvent pas dans les répertoires standards.

La compilation d'un programme parallèle est effectuée via un *wrapper* : mpic++. De même l'exécution se fait via l'outil mpirun qui attribue les processus aux processeurs (cœurs).

Les langages de script bash, awk, les expressions régulières, voire Perl, sont utiles pour l'automatisation des tests.

Sur Cluster

Outre cmake, sur le cluster, la compilation et l'exécution, nécessite de charger, voire d'écrire des modules écrits en Tcl¹⁸ qui fixent les variables d'environnement nécessaires à l'exécution/compilation. C'est l'outil *module*¹⁹ qui a été choisi.

Enfin une maîtrise minimale du gestionnaire Simple Linux Utility for Resource Management (SLURM)²⁰ est nécessaire pour soumettre les jobs sur les nœuds du cluster.

1.5.5 Environnement de développement

Le choix d'un Environnement de développement ou Integrated Development Environment (IDE) adapté est essentiel dans un projet informatique. Parmi, les outils possibles plusieurs candidats ont été testés :

- Code : :Blocks : rapide à prendre en main mais trop limité. A réservé pour les petits développements.
- Visual Studio Code pour Linux : <https://code.visualstudio.com/Download>
- CLion : <https://www.jetbrains.com/clion/>. Payant mais licence académique gratuite limitée dans le temps. Utilise cmake comme système de build.
- vim + grep récursif pour l'analyse de code.

¹⁴texthttps://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html

¹⁵<https://cmake.org/>

¹⁶Permet de mémoriser, archiver tout ou partie de l'état d'un objet i.e. ses attributs

¹⁷Surcouche à la bibliothèque Message Passing Interface avec interface C++

¹⁸Tool Command Language

¹⁹<https://github.com/cea-hpc/modules/blob/master/doc/source/index.rst>

²⁰<https://slurm.schedmd.com/>

²¹http://bioinfo.genotoul.fr/index.php/faq/job_submission_faq/

- Eclipse for Scientific Computing pour les développement d'application parallèle en fortran, C et C++, comprend le plugin Parallel Tool Plateforme et un débogueur parallèle mais probablement ... bugués : <https://www.eclipse.org/downloads/packages/release/2019-06/r/eclipse-ide-scientific-computing>
- Débogueur parallèle utilisé, gdb via le script gdb_run : ./gdb_run 8 problem.wcsp

```
#!/bin/bash
# script to put e.g. in toolbar2/your_build_folder/bin/Linux to run t
echo "usage: $0 nproc file.wcsp [other options]"

mpirun -n \$1 xterm -e gdb -ex run --args ./toolbar2 -para \$*
```

Autres outils de débogages (source : Bull Atos Technology, internet) :

- Valgrind : <https://fr.wikipedia.org/wiki/Valgrind>
- TotalView : GUI-based dynamic source code defect analysis tool that allows you to control processes and thread execution and see program state and variable values while your application runs. Link with MemoryScape for advanced memory debugging. <https://www.roguewave.com/products-services/totalview>
- Allinea DDT (debugger) and MAP (profiler) for HPC applications <https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt>
- HPCToolkit : an integrated suite of tools for measurement and analysis of program performance on computers
- TAU : profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java and Python
- Scalasca : software tool that supports the performance optimization of parallel programs by measuring and analyzing their runtime behavior

Environnement sélectionné

Malgré les défauts d'Eclipse, c'est ce dernier qui a été choisi car il facilite grandement la lecture de code. Les plugins permettant l'utilisation de cmake avec Eclipse ne fonctionnent pas très bien. Le plugin Eclipse, cmake editor, peut être utile. La solution retenue a consisté à produire un fichier de configuration avec ccmake, puis de créer un Makefile avec cmake en ligne de commande, puis de créer un projet de type "Makefile project". On trouvera un mode opératoire en annexe [10.2](#).



2. Aspects gestion de projet

Résumé

La première section de ce chapitre reprend les éléments du planning prévisionnel tel qu'il était envisagé fin avril 2019, la seconde décrit les étapes effectivement suivies lors de ce stage qui s'est déroulé du 18 mars au 15 septembre 2019. Ce projet de fin d'étude (PFE) a suivi plusieurs itérations :

1. Bibliographie : ≈ 1.5 mois,
2. Exploration de la piste parallélisation naturelle : ≈ 1 mois,
3. Exploration de la piste Paradigme Master-Worker : ≈ 2 mois,
4. Autres : rédaction rapports et beamer, séminaire, mise en place environnement de développement, prise en main des outils, etc. : ≈ 1.5 mois.

2.1 Planning prévisionnel

Il est prévu de procéder par itérations successives. L'itération 1 visera à produire un code fonctionnel répondant au cahier des charges du projet concernant HBFS. Les itérations suivantes tenteront le cas échéant de trouver des solutions plus performantes pour le HBFS tirées des réflexions et de l'expérience acquise ou de l'analyse d'éventuels de goulets d'étranglement et de l'adapter au BTD-HBFS (Backtracking with Tree Decomposition-HBFS).

1. mars-juin : itération 1

- Bibliographie ([17], [1], [26], [14], [22], [25])
 - wcsp,
 - consistance d'arc souple,
 - propagation de contraintes souples,
 - parallélisation de recherches arborescentes,
 - Architectures logiciel : MPI, OpenMPI, OpenMP, Pthread,

- Architectures matériel du laboratoire : serveurs MIAT, cluster genotoul.
- prise en main de l'environnement de travail et du code,
- Recherche et analyse de stratégies de parallélisation,
- codage/benchmarks/analyse des performances, Les benchmarks sont disponibles à <https://forgemia.inra.fr/thomas.schiex/cost-function-library>.

2. juillet-août : itérations suivantes et autres selon temps disponible

- Comparaison avec un travail similaire mené dans le cadre des modèles graphiques([14]),
- Piste de recherche 1 : adaptation de la méthode à une variante proposée dans (Larrosa, 2016, [15]).
- Piste de recherche 2 : intégration dans une métaheuristique parallèle [22]
- rédaction rapport, beamer,
- rédaction rapport (suite),
- Présentation beamer,
- Fin codage/benchmarks. Les benchmarks sont disponibles à <https://forgemia.inra.fr/thomas.schiex/cost-function-library>.

2.1.1 Approche initiale pour traiter le sujet

Parmi les approches possibles, il sera exploré lors de la première itération l'approche Embarrassingly Parallel Search (EPS) décrite dans [26] puis le paradigme Master-Worker[24] lors de l'itération suivante.

2.2 Planning effectif du stage

- 1. Début de stage : 18 mars 2019 09 :00**
- 2. mars-avril : Etat de l'art**
 - Analyse et mise en place de l'environnement de développement,
 - Bibliographie : Formalisme Weighted Constraint Satisfaction Problems (WCSP),
 - Bibliographie : Parallélisation,
 - Rapport premier mois.
- 3. mai : Familiarisation avec le code**
 - Analyse du code,
 - Recherche et analyse de stratégies de parallélisation,
 - Premier codage : production des sous-problèmes par toulbar2.
- 4. juin : Itération 1**
 - Analyse de la piste Embarrassingly Parallel Search (EPS),
 - Benchmarks sur PC et serveurs,
 - synthèse écrite,
 - Présentation mi-stage à l'ENAC,
 - Présentation à l'INRA : Journée stagiaires.
- 5. juillet : Itération 2**
 - Analyse du Paradigme Master-Worker,
 - Mise au point algorithme Master-Worker,

- Analyse d'implémentation : choix d'utiliser Boost.MPI et intégration dans cmake,
- Analyse du code toulbar2 : choix d'expurger le code de la partie clusterisée **Backtracking with Tree Decomposition Hybrid Best-node First Search (BTD-HBFS)**,
- Codage du **Hybrid Best-First Search (HBFS)** parallèle,
- Premiers tests sur PC et serveurs, débogage.

6. août : Itération 3

- Analyse environnement matériel : cluster GenoToul,
- Résolution des problèmes de compilation et d'exécution sur le cluster,
- Codages complémentaires : nombres de backtracks, partage de solutions,
- benchmarks,
- rédaction du présent rapport.

7. septembre : Soutenance

- Rédaction présentation beamer,
- Tests complémentaires,
- Comparaison avec d'autres approches,
- **Soutenance le 10 septembre 2019 13 :30-15 :00**

8. Fin de stage : 15 septembre 2019



3. Présentation des wcsp

Résumé

Ce chapitre présente un Etat de l'art élémentaire sur les [Weighted Constraint Satisfaction Problems \(WCSP\)](#). Ce formalisme permet de calculer une borne inférieure w_\emptyset utilisée par la suite notamment pour initialiser l'algorithme [Hybrid Best-First Search \(HBFS\)](#). Un problème combinatoire est représenté par un hypergraphe dont les nœuds sont les variables du problème et les arêtes les contraintes. Résoudre un WCSP consiste à minimiser une fonction de coûts globale, combinaison de fonctions de coûts locales appelées également contraintes souples. Le formalisme WCSP généralise celui des CSP en permettant la combinaison de contraintes dures et souples. Cette fonction est transformée de manière à obtenir une borne inférieure notée w_\emptyset . Cette borne inférieure est utilisée dans un branch and bound qui permet d'élaguer l'arbre de recherche et de calculer une borne inférieure globale. Une borne supérieure est obtenue à chaque fois que l'on obtient une affectation de toutes les variables du problème c'est à dire lorsque la suite de décisions décrit un chemin qui se termine par une feuille dans l'arbre binaire de recherche.

Par construction, la valeur optimale se situe entre ces deux bornes dont l'une augmente et l'autre diminue durant l'exécution de l'algorithme. Si le problème possède au moins une solution, la recherche se termine lorsque les bornes inférieure et supérieure deviennent égales, ce qui fournit la preuve d'optimalité. L'affectation complète associée à la valeur optimale constitue la solution recherchée.

3.1 Définitions

3.1.1 WCSP

Les [Weighted Constraint Satisfaction Problems \(WCSP\)](#), problème de satisfaction de contraintes pondérées, généralisent les problèmes de type *Constraint Satisfaction Pro-*

bles (CSP) en permettant d'exprimer des contraintes dures comme souples ; Les CSP étant NP-difficiles en général, il en est de même pour les WCSP (démonstration par restriction). Les contraintes des CSP sont remplacées par des *fonctions de coûts*.

3.1.2 Fonctions de coûts

Les fonctions de coûts sont aussi connues sous les termes de *contraintes valuées* ou de *contraintes souples*. Une contrainte porte sur un ensemble de variables appelé support, ou scope, de la contrainte. Le cardinal du support constitue son *arité n* avec :

$$n = \text{card}(S) = |S|$$

Une Fonction de coûts de support S, $w_S(t)$, associe un coût¹ à toute combinaison² de valeurs affectées aux variables impliquées dans la contrainte. Elle peut avoir une représentation tabulaire ou analytique.

Exemple de fonction de coûts unaires

Les contraintes valuées unaires portent sur une seule variable. Ces fonctions de coûts permettent de modéliser une préférence pour certaines des valeurs que peuvent prendre les variables du problème d'où le terme de contraintes souples. Par exemple, une fonction de coûts telle que

$$w_{\{x\}}(b) = 0, \quad \text{avec } x \in \{a, b\}$$

modélise le fait que la valeur b pour la variable x est autorisée.

A contrario, la valeur b est interdite si on écrit la contrainte souple

$$w_{\{x\}}(b) = k$$

où k est un entier positif maximal fixé plus ou moins arbitrairement en fonction du problème. k est noté $+\infty$ dans le formalisme des **Valued Constraint Satisfaction Problem (VCSP)**.

Ainsi, les fonctions de coûts de valeurs 0 et k modélisent des contraintes dures : si une fonction de coûts est définie de manière tabulaire, et non analytique, le tableau ne contiendra que ces valeurs 0 ou k.

On aura compris que les valeurs comprises entre 1 et k-1 permettront de graduer les préférences liées à la nature du problème à modéliser. Par exemple, pour k = 3, choisir $w_x(a) = 1$ et $w_x(b) = 2$ signifierait que l'on n'interdit pas la valeur b pour x mais qu'on lui préfère la valeur a.

Exemple de fonction de coûts binaires :

Une contrainte souple binaire est une fonction de coûts d'arité 2. Son support est de la forme $S = \{x_i, x_j\}$. La contrainte souple est une fonction qui reçoit un tuple en entrée et renvoie un coût :

$$w_S : D_i \times D_j \mapsto E = \{0, \dots, k\}$$

¹coût, poids ou weight en anglais d'où le choix de la notation w.

²combinaison ou tuple de valeurs

On note X , l'ensemble des variables du problème tel que $S \subseteq X$.

Chaque variable dans le support S possède un domaine D_i . Le nombre de tuples possibles correspond au cardinal du produit cartésien des domaines :

$$\prod_{i=1}^n |D_i|$$

Ainsi, si $X = S$ et $D_1 = D_2 = \{a, b\}$, la taille de l'espace de recherche sera égale 4.

3.1.3 Structure de valuation pour les WCSP

Une structure de valuation V est un Quintuplet constitué des éléments suivants :

1. Un ensemble de coûts possibles : $E = \{0, \dots, k\}$ avec k élément *absorbant*,
2. Un opérateur d'agrégation $+_k$ tel que $\forall a, b \in E, a +_k b = \min(a + b, k)$,
3. Une relation d'ordre : $<$,
4. Un Coût min : $0 \rightarrow$ autorise un tuple de valeurs,
5. Un Coût max : $k \rightarrow$ interdit un tuple de valeurs.

La structure de valuation exprime les règles qui permettent notamment de combiner les contraintes locales en une valuation ou fonction de coûts globale.

3.1.4 Fonction de coûts globale

Les fonctions de coûts locales, i.e. qui ne concernent qu'un sous-ensemble de variables, sont agrégées pour obtenir une *Valuation* globale du problème, c'est à dire, une fonction de coûts globale en les variables du problème. Il s'agit alors de trouver le, ou les tuples, de valeurs à associer à toutes les variables du problème, c'est à dire de trouver *une affectation complète*, $x = (x_1, \dots, x_n) := (t_1, \dots, t_n) = t$ qui minimise la fonction de coûts globale ou valuation globale.

$$Val(t) = Val(t_1, \dots, t_n) = w_\emptyset + \sum_{i=1}^n w_i(t_i) + \sum_{w_{ij} \in C} w_{ij}(t_i, t_j)$$

Résoudre un WCSP revient donc à trouver une affectation de toutes les variables, un tuple complet, qui minimisent sa fonction de coûts globale.

3.1.5 Réseaux de fonctions de coûts

Un réseau de fonctions de coûts, ou Cost Function Network (**CFN**), est un quadruplet :

1. X : ensemble de variables
2. D : ensemble des domaines des variables de X
3. W : ensemble de fonctions de coûts locales
4. V : structure de valuation

On peut le représenter sous la forme d'un hypergraphe dont les nœuds sont les variables et les arêtes les fonctions de coûts.

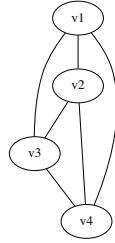


FIGURE 3.1 : Hypergraphe du problème des 4 reines : 4wqueens.wcsp. Le problème est représenté par un simple graphe car l’arité maximale des contraintes est égale à 2. Nous avons donc d’une part, le graphe représentant le problème, et d’autre part, l’arbre de recherche, ou arbre de décision, développé progressivement pour explorer l’espace d’états ici de taille $4^4 = 256$, valeur qui correspond au cardinal du produit cartésien des domaines des variables. Ici quatre variables : V1 à V4 de domaine identique $D = \{1, 2, 3, 4\}$. $V4 = 2$ signifie qu’on place une reine en colonne 4 et ligne 2. Dans ce type de graphe, la micro-structure n’apparaît pas ; les domaines ne sont pas représentés dans chaque nœud donc les fonctions de coûts unaires n’apparaissent pas non plus.

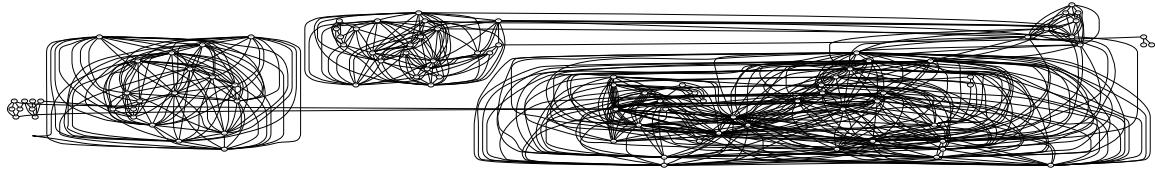


FIGURE 3.2 : Hypergraphe du problème 404.wcsp. Problème-type utilisé lors des développements de hybridSolvePara(). On note qu’il présente clairement une structure en clusters.

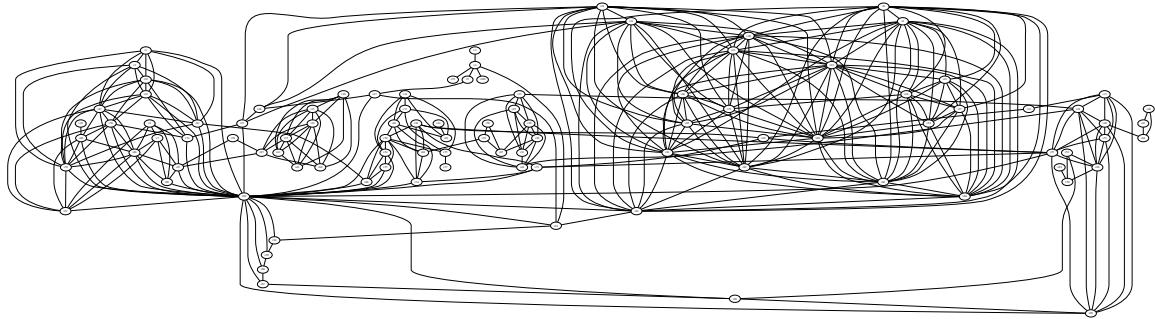


FIGURE 3.3 : Hypergraphe du problème scen06.wcsp

La micro-structure du réseau de fonctions de coûts fait apparaître, dans chaque nœud, les valeurs du domaine de la variable correspondante comme dans la figure 3.4.

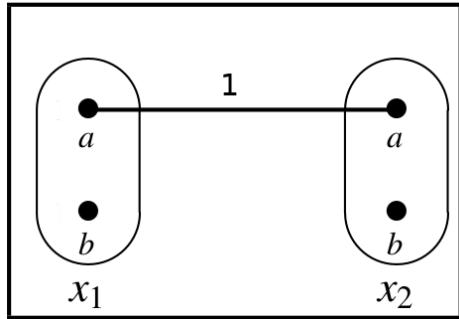


FIGURE 3.4 : Exemple de graphe avec micro-structure d'un problème simple à deux variables de même domaine $\{a, b\}$. Par convention, les contraintes de coût nul ne sont pas représentées. Ainsi la contrainte unaire w_1 sur x_1 est telle que $w_1(b) = 0$, la contrainte binaire w_{12} est telle que $w_{12}(a, a) = 1$ et $w_{12}(a, b) = 0$. Si $k > 1$, le graphe indique que l'on préfère éviter le cas (a, a) . Si $k = 1$, ce cas est interdit.

Les définitions ci-dessus peuvent maintenant être mises à profit pour détailler un peu plus le formalisme des WCSP.

3.2 Formalisme des WCSP

3.2.1 Description succincte

Le problème de satisfaction de contraintes valuées (VCSP) est décrit dans [27],[5]. C'est un formalisme général qui permet, contrairement aux problèmes de satisfaction de contraintes *dures*, d'exprimer des préférences entre variables et donc entre solutions ou de traiter des problèmes sur-contraints pour lesquels certaines contraintes peuvent éventuellement être violées. Le problème de satisfaction de contraintes pondérées (*Weighted Constraint Satisfaction Problem* (WCSP)) constitue un cas particulier de VCSP. Les réseaux de contraintes valuées ou réseaux de fonctions de coûts (CFN : Cost Function Network) définissent une classe de modèles graphiques. En effet, l'ensemble X des variables, l'ensemble D de leurs domaines respectifs, l'ensemble F des fonctions de coûts et d'une structure de valuation V peuvent être représentés sous forme d'un hypergraphe plus ou moins détaillé selon qu'on veuille capturer la structure globale du problème ou sa micro-structure qui intègre l'information sur les domaines des variables et les fonctions de coûts [5]. Un exemple de graphe décrivant la micro-structure d'un problème est donné dans [28].

3.2.2 Propagation de contraintes souples dans l'hypergraphe des fonctions de coûts

Des algorithmes de propagation de contraintes souples ont été développés par l'équipe SAB, en particulier **Existential Directional Arc Consistency** (EDAC) et implémentés dans toulbar2 qui se basent sur des opérations d'extensions et de projections , dites **Equivalent Preserving Transformations** (EPT), qui conservent la fonction de coûts globale, et qui permettent in fine d'obtenir une borne inférieure de cette fonction de coûts globale [10].

L'idée est de concentrer les coûts sur une seule variable ce qui peut déclencher une augmentation de w_\emptyset via une projection unaire. w_\emptyset est une fonction de coûts constante positive, i.e. d'arité nulle donc indépendante de toute variable. w_\emptyset est une borne inférieure de l'optimum car la fonction de coûts globale s'écrit comme la somme de w_\emptyset et d'une quantité positive Q : $Val = w_\emptyset + Q$. Il s'agit de trouver la bonne heuristique qui donnera la bonne séquence de projections et extensions pour maximiser w_\emptyset et donc se rapprocher de l'optimum. Les valeurs w_\emptyset sont calculées pour chaque noeud de l'arbre de recherche et recalculées pour les noeuds prélevés dans la file de priorité *open* dont il sera question dans le chapitre **Hybrid Best-First Search (HBFS)**. Dans [28] est présenté la notion de propagation de contraintes souples à l'aide de 3 types de transformations : projection, extension et projection unaire comme en figure 3.5.

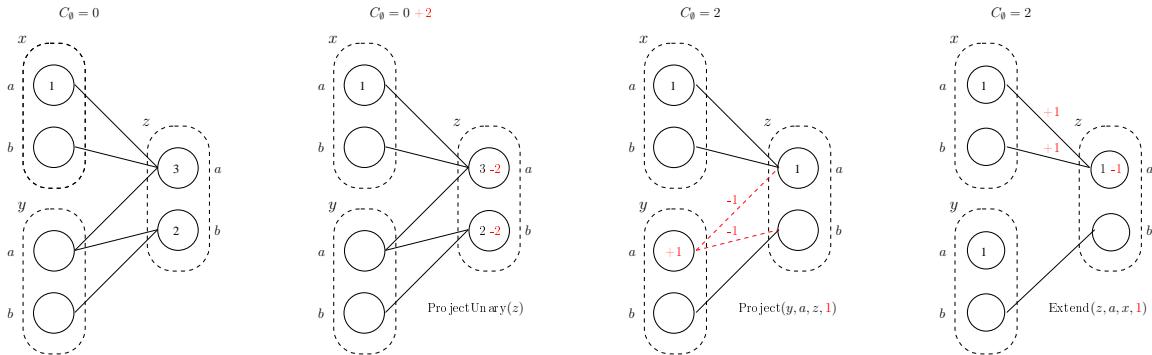


FIGURE 3.5 : Trois types de transformations de base. Initialement, figure de gauche, on a $w_\emptyset = C_\emptyset = 0$. La variable z peut prendre la valeur a dont le coût est 3 ou la valeur b dont le coût est 2. Cela donne une représentation "tabulaire". On peut donc opérer une projection unaire en transférant un coût égal à $2 = \min(2, 3)$ de la fonction de coûts unaire vers la fonction de coût d'arité nulle w_\emptyset .

Utilisation du minorant obtenu par propagation dans un branch and bound

Cette propagation par extension et projection dans l'hypergraphe fournit une borne inférieure de l'optimum en chaque noeud de l'arbre de recherche du B&B [3]. La valeur w_\emptyset est un minorant de l'optimum du sous-problème associé à un noeud. Dans le code de toulbar2, elle est notée *lb*.

Par ailleurs, dès qu'on obtient une affectation complète des variables du problème, i.e. on arrive à une feuille de l'arbre de recherche, on peut calculer une valeur v pour la fonction de coûts globale.

La borne supérieure *cub* du problème global qui constitue la meilleure valeur courante³, est mise à jour avec cette valeur v dans le cas où $cub \geq v$. Dès que $lb \geq cub$, on pourra élager l'arbre de recherche, i.e. supprimer le noeud en question, qui ne peut donner de solution. Ses descendants ne peuvent donner de solution car leurs bornes inférieures *lb* sont toutes supérieures ou égales au *lb* du noeud parent. Ainsi, l'algorithme HBFS, détaillé

³valeur courante ou valeur titulaire, ou incumbent value, associée à la solution fournie sous la forme d'un tuple complet.

ultérieurement dans le chapitre éponyme, extrait le nœud de *lb* minimum de la file de priorité et recalcule ce *lb* et, si $lb \geq cub$, il sera inutile de développer l'*arbre de décision* à partir de ce nœud. En ce sens, la recherche est partielle. On constate aussi que l'heuristique de calcul de $lb = w_\emptyset$ devra être la plus efficace possible.

3.3 Exemple de modélisation

Problème

Soit le problème à modéliser dont l'ensemble de variables $X = \{x_1, x_2, x_3\}$ de domaines identiques :

$$D_1 = D_2 = D_3 = \{a, b\} \quad a, b \in \mathbb{N}$$

La valeur a est préférée à la valeur b pour les 3 variables et nous avons la contrainte "dure" : $x_1 \neq x_2$

Modélisation

La valeur a préférée à b peut être modélisée par 3 contraintes souples unaires identiques définies par :

$$w_i(a) = 0, \quad w_i(b) = 1, \quad i \in \{1, 2, 3\} \quad E = \{0, \dots, k\} \quad \text{avec} \quad k > 1$$

La contrainte dure binaire $x_1 \neq x_2$ peut être modélisée par la fonction de coût binaire

$$w_{\{x_1, x_2\}}(t_1, t_2)$$

notée aussi

$$w_{12}(t_1, t_2)$$

On a $\prod_{i=1}^n |D_i| = 2 \times 2 = 4$ coûts possibles pour chaque tuple. Ainsi w_{12} est représentable par la table 3.1.

w_{12}	a	b
a	k	0
b	0	k

TABLE 3.1 : Représentation tabulaire de la contrainte dure binaire w_{12} .

3.4 Exemple de projection

Cette section présente une exemple didactique qui développe celui de l'article [5] afin d'illustrer sur un cas concret les **Equivalent Preserving Transformations (EPT)** permettant de transférer les coûts vers une contrainte constante donc d'arité nulle w_\emptyset qui sera utilisée dans l'algorithme **Hybrid Best-First Search (HBFS)**.

Soient deux variables x_1, x_2 avec les mêmes domaines :

$$D_1 = D_2 = \{a, b\}$$

et un ensemble de coûts :

$$E = \{0, \dots, k\}, \quad \text{avec} \quad k > 1$$

On décide qu'on préfère la valeur b à la valeur a , ce qui peut se traduire par :

$$w_1(a) = w_2(a) = 1$$

$$w_1(b) = w_2(b) = 0$$

et qu'on préfère éviter que nos deux variables prennent la même valeur b .

$$w_{12}(b, b) = 1$$

$$w_{12}(a, a) = w_{12}(a, b) = w_{12}(b, a) = 0$$

On représente la micro-structure de ce réseau de fonctions de coûts en figure 3.6.a avec la convention que les coûts unaires nuls et les arêtes de coût nul ne sont pas représentés.

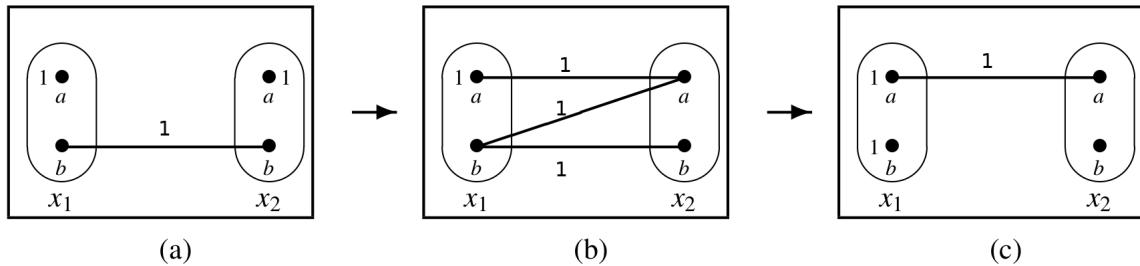


FIGURE 3.6 : Micro-structure d'un réseaux de fonctions de coûts.

La fonction de coût globale a pour expression :

$$Val(t_1, t_2) = w_\emptyset + \sum_{i=1}^n w_i(t_i) + \sum_{w_{ij} \in C} w_{ij}(t_i, t_j)$$

On calcule, par exemple, sa valeur pour le tuple $t = (a, a)$ dans le cas de la figure 3.6.a pour illustrer le fait que notre réseau de fonctions de coûts évolue mais la fonction de coût globale est préservée. On pourrait faire le même calcul avec les 3 autres tuples : $(a, b), (b, a), (b, b)$ pour vérifier la conservation de la valuation du problème par projection, extension et projection unaire.

$$Val(a, a) = w_\emptyset + w_1(a) + w_2(a) + w_{12}(a, a) = 0 + 1 + 1 + 0 = 2$$

De la figure 3.6 (a) à la figure 3.6 (b), on effectue une extension de la contrainte unaire w_2 vers la contrainte binaire w_{12} d'où le terme d'extension. Autrement dit, on transfert le coût $w_2(a) = 1$, qui devient alors nul, vers les coûts binaires tels que $w_{12}(a,a) = 1$ et $w_{12}(b,a) = 1$ de manière à préserver la fonction de coûts globale. Ainsi, on peut vérifier sur la figure 3.6.b que la valeur associée au tuple (a,a) est conservée :

$$Val(a,a) = w_\emptyset + w_1(a) + w_2(a) + w_{12}(a,a) = 0 + 1 + 0 + 1 = 2$$

De manière similaire, de 3.6.b à 3.6.c, une projection d'une contrainte binaire vers une contrainte unaire telle que sur $w_1(b)$ prend la valeur 1 tandis que $w_{12}(b,a) = 1$ et $w_{12}(b,b) = 1$ deviennent nulle conserve la valuation globale. La valeur pour la figure 3.6.c est préservée mais le réseau est modifié :

$$Val(a,a) = w_\emptyset + w_1(a) + w_2(a) + w_{12}(a,a) = 0 + 1 + 0 + 1 = 2$$

En figure 3.6.c, on constate qu'il est possible d'effectuer une projection unaire de la contrainte unaire w_1 vers la contrainte d'arité nulle w_\emptyset . On obtient le nouveau réseau de la figure 3.7 en soustrayant le min de $w_1(a) = 1$ $w_1(b) = 1$, qui égal à 1, de chacunes de ces valeurs, ce qui donne $w_1(a) = 0$ $w_1(b) = 0$ et en le transférant sur la contrainte constante w_\emptyset . Nous avons donc notre minorant $w_\emptyset = 1$ utilisable par exemple dans un branch and bound.

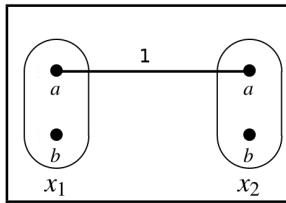


FIGURE 3.7 : Nouveau réseau obtenu par projection unaire à partir de la 3.6.c. La contrainte constante est désormais $w_\emptyset = 1$. Ce qui constitue un minorant de l'optimum de la fonction de coûts globale.

Ce dernier réseau préserve également la valuation :

$$Val(t_1,t_2) = w_\emptyset + \sum_{i=1}^n w_i(t_i) + \sum_{w_{ij} \in C} w_{ij}(t_i, t_j)$$

$$Val(a,a) = w_\emptyset + w_1(a) + w_2(a) + w_{12}(a,a)$$

$$Val(a,a) == 1 + 0 + 0 + 1 = 2$$

3.5 Exemple jouet : Le problème pondéré des 4 reines

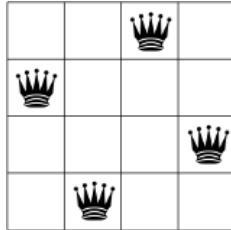


FIGURE 3.8 : Le problème des 4 reines.

Le problème des 4 reines, dont le graphe a été introduit figure 3.1, consiste à placer 4 reines sur un échiquier 4x4 telles qu’elles soient inoffensives les unes par rapport aux autres. On pourra consulter le site suivant : https://acrogenesis.com/or-tools/documentation/user_manual/manual/introduction/4queens.html. La version pondérée consiste à exprimer des préférences parmi les solutions possibles comme décrit dans le fichier wcsp commenté en détail ci-après.

PROBLEME DES 4 REINES - VERSION PONDÉRÉE

nom du problème nombre de variables taille maxi du domaine nombre de fonctions de coûts

Fichier : /home/toto/Bureau/4wqueens.wcsp

Page 1 sur 2

Arité de la fonction de coûts = contrainte binaire sur 2 variables.

4-WQUEENS-2-0 4 4 10 borne supérieure initiale UB du problème e.g = k
 4 4 4 4 taille des domaines des 4 variables d'index implicite 0 à 3 : wcsp est un format positionnel
 2 0 1 0 10 nombre de tuples avec un coût <> du coût par défaut = 10

0 0 5	Le coût par défaut est 0
0 1 5	
1 0 5	
1 1 5	
1 2 5	
2 1 5	
2 2 5	
2 3 5	
3 2 5	
3 3 5	
2 0 2 0 8	
0 0 5	
0 2 5	
1 1 5	
1 3 5	
2 0 5	
2 2 5	
3 1 5	
3 3 5	
2 0 3 0 6	
0 0 5	
0 3 5	
1 1 5	
2 2 5	
3 0 5	
3 3 5	
2 1 2 0 10	
0 0 5	
0 1 5	
1 0 5	
1 1 5	
1 2 5	
2 1 5	
2 2 5	
2 3 5	
3 2 5	
3 3 5	
2 1 3 0 8	
0 0 5	
0 2 5	
1 1 5	
1 3 5	
2 0 5	
2 2 5	
3 1 5	
3 3 5	
2 2 3 0 10	
0 0 5	
0 1 5	
1 0 5	
1 1 5	
1 2 5	
2 1 5	
2 2 5	
2 3 5	
3 2 5	
3 3 5	

1 0 0 2	1 : arité de la fonction de coûts. 0 : index de la 1ère variable donc la contrainte porte sur la variable X0 ; le support est le singleton {X0}. on la note w0. 0 : valeur du coût par défaut 2 : nombre de tuples, ici, singletons de coûts différents du coût par défaut.
1 1	
3 1	
1 1 0 2	
1 1	
2 1	

La contrainte binaire porte sur les variables X0 et X1
 Le scope ou support de la contrainte est {X0,X1}

10 tuples de coût 5 : la contrainte concerne les deux variables X0 et X1, on la note w01, et leurs domaines sont de même taille 4. On a donc $4 \times 4 = 16$ tuples possibles : 10 ont un coût = 5, et 6 un coût = 0. La fonction de coûts est donc parfaitement déterminée.

NB : la fonction de coût a pour valeurs soit 0, soit le coût max = k = 5. Cette fonction de coût exprime donc une contrainte DURE

Le problème des 4 reines ici modélisé utilise 4 variables X0, X1, X2, X3. Chaque index des variables correspond à une colonne dans le damier 4x4 sur lequel on doit placer 4 reines sans qu'elles puissent s'annexer mutuellement. Les valeurs des variables sont 0, 1, 2, 3. Elles correspondent aux lignes. Ainsi X0 = 1 et X1 = 0 signifie que la reine 1 est placée en case (0,0), la reine 2 en case (0,1). On interdit cette configuration en lui associant le coût max = 5 !

3 : index de la valeur prise par X0. Ici la taille du domaine est 4 ; les index des valeurs sont donc égaux aux valeurs.
 1 : coût associé si X0 prend la valeur indexée par 3, i.e. la valeur 3 : $w0(3)=1 \Leftrightarrow X0=3$ n'est pas interdite.
 X0 peut prendre les valeurs : 0, 1, 2, 3. On a $w0(0)=0$, $w0(1)=1$, $w0(2)=0$, $w0(3)=1$ puisque la coût par défaut est 0. Ceci exprime que le modélisateur, pour des raisons qui lui sont propres, préfère que X0 prennent les valeurs 0 ou 2. Il préfère que la reine de la première colonne soit placée sur la première ou la troisième ligne mais ce n'est pas une obligation.
 Ici, la fonction de coût exprime une CONTRAINTE SOUPLE !

```
1 2 0 2  
1 1  
2 1  
1 3 0 2  
0 1  
2 1
```

4 contraintes souples pour ce
weighted 4 queens problem.

Résolution du problème avec toulbar2 :

```
..../toulbar2 4wqueens.wcsp -s
```

```
version : 1.0.1-440-gc96322c-kad (1566679382), copyright (c) 2006-2019, toulbar2 team
```

```
loading wcsp file: 4wqueens.wcspRead 4 variables,  
with 4 values at most, and 10 cost functions, with maximum arity 2.
```

```
Cost function decomposition time : 2e-06 seconds.
```

```
Preprocessing time: 0.000282 seconds.
```

```
4 unassigned variables, 16 values in all current domains (med. size:4, max size:4) and  
6 non-unary cost functions (med. degree:3)
```

```
Initial lower and upper bounds: [0, 5] 100.000%
```

```
SEQUENTIAL HBFS MODE!!! ADD -para OPTION FOR PARALLEL MODE
```

```
New solution: 0 (0 backtracks, 1 nodes, depth 2)  
2 0 3 1
```

```
Node redundancy during HBFS: 0.000 %
```

toulbar2 trouve une valeur optimale nulle (minimale) pour la fonction de coût globale.
Cette valeur est associée à une affectation complète $(X_0, X_1, X_2, X_3) = (2, 0, 3, 1)$
ce qui correspond à la 1ère reine placée en 3ème ligne, la 2nd en 1ère ligne, etc.
on vérifie que toutes les contraintes dures sont respectées. Les 4 contraintes souples
le sont également mais cela aurait pu ne pas être le cas.

```
Optimum: 0 in 0 backtracks and 1 nodes ( 0 removals by DEE) and 0.000 seconds.
```

```
end.
```



4. Toulbar2

Résumé

Ce chapitre présente succinctement toulbar2, un solveur de contraintes *souples* qui comprend plusieurs algorithmes dont l'[Hybrid Best-First Search \(HBFS\)](#). Cet exposé se place tant du point de vue de l'utilisateur que celui du programmeur appelé à contribuer aux développements et désireux d'acquérir une vue synoptique. On trouvera des informations plus précises en consultant sa [documentation](#) ou l'article [ToulBar2, an open source exact cost function network solver](#).

4.1 Généralités

toulbar2 est un solveur de réseaux de fonctions de coûts dont la particularité est de pouvoir résoudre notamment des problèmes de satisfactions contraintes souples. D'autres informations générales sont intéressantes à connaître :

- Codé en C++,
- Développé à l'origine par les équipes INRA-MIAT à Toulouse et IIIA-CSIC¹ de l'Université Polytechnique de Catalogne (UPC) à Barcelone,
- toulbar : Contraction de Toulouse et Barcelone,
- toulbar version 1 codée en C,
- version 2 codée en C++,
- toulbar2 a bénéficié d'autres contributions,
- Maintenu par [Simon de Givry](#),
- Libre(free), gratuit(free) et open source,
- Financé partiellement par l'agence Nationale de la Recherche (ANR).

¹<http://www.iiia.csic.es>

4.2 Caractéristiques de toulbar2

Sur un plan technique, outre la documentation sur <https://github.com/toulbar2/toulbar2/tree/master/doc>, on pourra utilement consulter les quelques éléments ci-dessous :

- toulbar2 est compilable en tant qu'application ou en tant que bibliothèque²,
- Après compilation l'exécutable se trouve dans le dossier toulbar2/release/bin/Linux si on suppose que l'utilisateur a nommé son dossier de build : "release"/ Mettre les problèmes à résoudre dans ce dossier et lancer la résolution avec ./toulbar2 problem.wcsp
- Le build system permet de produire un package *.deb pour installation sur ubuntu : make package,
- Test de non regression : make test,
- Utilise le système de build cmake,
- Supporte les plateformes Linux, Mac et Windows,
- Comprend un ensemble d'algorithmes dont l'**Hybrid Best-First Search (HBFS)**,
- Implémente divers heuristiques pour la transformation des **Weighted Constraint Satisfaction Problems (WCSP)** qui conservent la fonction de coût globale tout en fournissant une borne inférieure correcte en un temps raisonnable : Virtual Arc Consistency for Weighted CSP[6], Existential Directional Arc Consistency(EDAC)[10] (consistance d'arc locale par défaut dans toulbar2)
- toulbar2 effectue des pré-traitements efficaces qui limitent la taille de l'intervalle dans lequel se trouve la valeur optimale ce qui accélère la phase de recherche de solution optimale, i.e. minimale.
- toulbar2 autorise de nombreuses options affichables par la commande ./toulbar2 sans options. quelques options utiles :
 - -v=3 : option qui augmente la verbosité des sorties de toulbar2. Associé à compilation en mode debug, elle permet de mieux superviser l'exécution du programme.
 - -s : affiche les solutions en plus des valeurs optimales trouvées,
 - -nopre : désactive les pré-traitements,
 - -A : utilise la consistance d'arc souple local VAC au lieu de EDAC,
 - si -option désigne une option activée par défaut -option suivie de ":" (-option :) désactive cette dernière. Exemple -solr est une option par défaut, -solr :; la désactive,
- L'ordre des options n'a pas d'importance : toulbar2 utilise SimpleOpt³

²Un exemple d'utilisation de la *librairie* toulbar2, libtb2.so, qui fournit une Application Programming Interface(API) se trouve dans toulbar2/src/toolbar2test.cpp

³SimpleOpt est utilisé pour lire les nombreuses options de toulbar2. Cf toulbar2/src/tb2main.cpp <https://github.com/brofield/simpleopt/>

4.3 Utilisation de toulbar2

L'utilisation de toulbar2, compilé en tant qu'application et non sous forme de bibliothèque, consiste à modéliser un problème et à le transcrire en un format supporté par le solveur puis à utiliser ce fichier : ./toulbar2 problem.wcsp -option-1 ... -option-n. Toulbar2 affiche alors en temps réel la progression de la recherche sous forme d'un gap d'optimalité qui désigne l'intervalle dans lequel se trouve la valeur optimale recherchée en l'occurrence $[clb, cub]$, où clb est la borne inférieure globale et cub la meilleure valeur courante, i.e. le coût global le plus faible, trouvé à ce stade de la recherche.

La commande ci-dessous lance la version parallèle boot.MPI de HBFS, sur 4 cpus, pour résoudre le problème 404.wcsp.

L'option -s, \textit{ou --show}, visualise la solution optimale et pas seulement la valeur optimale.

Le gap d'optimalité indique la progression de la recherche.

```
mpirun -np 4 ./toulbar2 404.wcsp -s -para
```

```
Preprocessing time: 0.006 seconds.
```

```
88 unassigned variables, 226 values in all current domains  
(med. size:2, max size:4) and 594 non-unary cost functions  
(med. degree:13)
```

```
Initial lower and upper bounds: [66, 158] 58.228%
```

```
PARALLEL HBFS MODE!!!
```

```
New solution: 122 (0 backtracks, 17 nodes, depth 18)
```

```
0 0 2 1 0 1 1 1 3 3 1 1 0 1 1 1 1 1 3 1 1 0 1 1 1 1 0 1 2 0 1 1  
1 0 1 1 0 1 1 1 1 1 0 1 1 1 1 3 1 1 1 1 1 1 1 3 1 3 1 1 1 1  
1 1 1 3 1 0 1 0 1 0 1 1 0 1 1 0 0 3 3 0 2 1 1 3 1 1 3 1 0 3 1  
1 1 3 3 1
```

```
.....
```

```
New solution: 114 (170 backtracks, 368 nodes, depth 16)
```

```
0 0 2 1 1 1 0 3 3 1 1 1 1 1 0 1 3 1 1 1 0 1 1 0 1 1 3 1 0  
3 1 1 0 0 1 1 0 1 1 1 1 0 1 1 1 1 1 3 1 1 1 0 1 1 3 3 1 3 1 1  
1 0 1 1 1 1 0 1 0 1 1 0 1 1 0 0 1 2 1 0 3 1 3 1 1 3 1 0  
3 1 1 1 3 3 1
```

```
Optimality gap: [85, 115] 26.087 % (4624 backtracks, 9607 nodes)
```

```
Optimality gap: [85, 114] 25.439 % (6694 backtracks, 13777 nodes)
```

```
Optimality gap: [86, 114] 24.561 % (24270 backtracks, 49131 nodes)
```

```
Optimality gap: [87, 114] 23.684 % (27365 backtracks, 55355 nodes)
```

```
Optimality gap: [88, 114] 22.807 % (72805 backtracks, 146665 nodes)
```

```
Optimality gap: [89, 114] 21.930 % (123145 backtracks, 247592 nodes)
```

```
Optimality gap: [90, 114] 21.053 % (252452 backtracks, 506713 nodes)
```

```
Optimality gap: [91, 114] 20.175 % (360518 backtracks, 723280 nodes)
```

```
Optimality gap: [92, 114] 19.298 % (485927 backtracks, 974600 nodes)
```

```

Optimality gap: [93, 114] 18.421 % (587870 backtracks, 1178905 nodes)
Optimality gap: [94, 114] 17.544 % (696440 backtracks, 1396820 nodes)
Optimality gap: [95, 114] 16.667 % (784642 backtracks, 1574130 nodes)
Optimality gap: [96, 114] 15.789 % (903232 backtracks, 1812364 nodes)
Optimality gap: [97, 114] 14.912 % (958291 backtracks, 1923538 nodes)
Optimality gap: [98, 114] 14.035 % (1009840 backtracks, 2028010 nodes)
Optimality gap: [99, 114] 13.158 % (1050466 backtracks, 2110700 nodes)
Optimality gap: [100, 114] 12.281 % (1069395 backtracks, 2149633 nodes)
Optimality gap: [101, 114] 11.404 % (1081933 backtracks, 2175637 nodes)
Optimality gap: [102, 114] 10.526 % (1091363 backtracks, 2195620 nodes)
Optimality gap: [103, 114] 9.649 % (1096809 backtracks, 2207552 nodes)
Optimality gap: [104, 114] 8.772 % (1101985 backtracks, 2219221 nodes)
Optimality gap: [105, 114] 7.895 % (1103827 backtracks, 2223803 nodes)
Optimality gap: [106, 114] 7.018 % (1106727 backtracks, 2230612 nodes)
Optimality gap: [107, 114] 6.140 % (1108582 backtracks, 2235893 nodes)
Optimality gap: [108, 114] 5.263 % (1109520 backtracks, 2239138 nodes)
Optimality gap: [109, 114] 4.386 % (1110212 backtracks, 2241956 nodes)
Optimality gap: [110, 114] 3.509 % (1110684 backtracks, 2244500 nodes)
Optimality gap: [111, 114] 2.632 % (1111033 backtracks, 2246767 nodes)
Optimality gap: [112, 114] 1.754 % (1111180 backtracks, 2248350 nodes)
Optimality gap: [113, 114] 0.877 % (1111228 backtracks, 2249481 nodes)
Optimality gap: [114, 114] 0.000 % (1111245 backtracks, 2250258 nodes)
Optimum: 114 in 1111245 backtracks and 2250258 nodes
( 6 removals by DEE) and 8.915 seconds.

```

Dans le cadre de ce PFE, seuls le format *positionnel* *.wcsp, et sa version compressée *.wcsp.xz⁴ pour les problèmes dont la taille se compte en Giga Octets, auront été utilisés.

On distinguera le problème **Weighted Constraint Satisfaction Problems (WCSP)** du format qui le représente. Une même modélisation d'un problème peut être représentée par différents formats. En effet, toulbar2 supporte une quinzaine de ces formats d'entrée dont un format de type json⁵ *.cfn⁶. Cf. documentation <https://github.com/toulbar2/toulbar2/blob/master/doc/CFNformat.pdf> et l'exemple ci-dessous :

```
{
problem: { name: "maximization_example", mustbe: ">-5.0" },
variables: { "v1": ["a", "b"], "v2": ["c", "d"] },
functions: {
"c0": {scope: [], costs: [-6.0]},
"c1": {scope: ["v1"], costs: [1.0, 0.5]},
```

⁴Compression gérée par la bibliothèque Boost

⁵Le format JavaScript Object Notation est utilisé, à l'origine, pour les communications entre navigateurs et serveurs. C'est un format texte qui enregistre les données sous forme de couples nom_de_la_variable : valeur_de_la_variable

⁶cost function network format

```

"c2": {scope: ["v2"], costs: [1.0, 0.5]},
"c12": {scope: ["v1", "v2"], costs: [-1.0, 0.5, -2.0, 5.5]}
}
}

```

Ainsi, un problème wcsp, i.e. un réseau de fonctions de coûts⁷, peut être représenté, au choix, par le format wcsp ou cfn. On peut exécuter toulbar2 sans options pour obtenir une liste complète des formats supportés.

4.4 Elément synoptique du code de toulbar2

On décrit ici succinctement des choix de conceptions de toulbar2 utiles à la compréhension du code.

4.4.1 Description succincte du code

1. Fichier principal du programme : toulbar2/src/tb2main.cpp :
 - Lit les options passées à toulbar2 avec SimpleOpt,
 - Charge les fichiers d'entrées,
 - Crée l'objet solver,
 - Lance la méthode solve()
2. Fichiers essentiels du solver : toulbar2/src/search/tb2solve.cpp et tb2solve.hpp
3. La classe Solver dérive de la classe WeightedCSPSolver qui représente un solver wcsp
4. Attributs principaux de la classe Solver qui contient à la fois des types de base et d'autres classes ou objets :
 - Un pointeur vers un objet wcsp de type WeightedCSP : représente un problème wcsp. On accède en lecture ou écriture à la borne inférieure globale *clb* ou à la borne supérieure globale *cub* via cet objet wcsp : *wcsp->getUb()*. Idem pour les méthodes : *wcsp->setSolution(...)*, *wcsp->updateUb(...)*,
 - Classe OpenNode : représente un nœud ouvert avec un coût de type Cost qui définit la borne inférieure locale. Cost est un type personnalisé dans toulbar2 correspondant à un *long long int*. D'autres types, telle que Value sont en fait des entiers.
 - Classe OpenList : c'est en fait une file de priorité qui contiendra les nœuds classés selon leur borne inférieure locale croissante.
 - Structure ChoicePoint : représente un point de choix, une décision dans l'arbre de recherche (branching)
 - Classe CPstore : vecteur de points de choix qui mémorise de manière économique les décisions prises pour pouvoir les "rejouer",
 - Classe Work : classe ajoutée pour passer des messages entre processus avec la bibliothèque boost.MPI. Utilisée dans la méthode hybridSolvePara().

⁷Un réseau bayésien et un Markov Random Field (MRF) pourront être représentés par des fichiers au format uai.

5. Principales méthodes :

- pair<Cost, Cost> Solver : :hybridSolve(Cluster *cluster, Cost clb, Cost cub) : Effectue le HBFS séquentiel avec ou sans prise en compte de la structure du problème sous forme d'arbres de clusters : [Backtracking with Tree Decomposition Hybrid Best-node First Search \(BTD-HBFS\)](#),
- pair<Cost, Cost> Solver : :hybridSolvePara(Cost clb, Cost cub) : version parallélisée du HBFS,
- recursiveSolve(...) et binaryChoicePoint(...) : Effectuent récursivement le branch and bound en profondeur d'abord (DFBB),
- bool Solver : :solve() : initie la recherche,
- void Solver : :restore(...) : Restaure un nœud en rejouant la séquence de décisions dans l'arbre de recherche lorsque ce nœud est prélevé de la file open,
- pair<Cost, Cost> Solver : :hybridSolveSeq(Cost clb, Cost cub) : version HBFS séquentielle expurgée du BTD-HBFS.

4.4.2 Fonction général de toulbar2

Un objet "solver" de wcsp est créé qui possède un état caractérisé par les valeurs de ses attributs.

Le solveur comprend également un certain nombre de méthodes qui modifient les attributs du solveurs. Des méthodes permettent de sauvegarder ou de restaurer l'état du solveur.

Les attributs importants de la recherche tels que *clb* se trouvent dans l'objet wcsp accédé par adresse et qui représente un problème wcsp.

Les méthodes modifient donc les variables *globalement*, la portée étant celle de la classe Solver.

D'autres variables sont utilisées qui, elles, sont globales au niveau du programme, notamment les variables préfixées par "Toulbar2 :" qui contrôlent toulbar2 notamment via les options passées en arguments au programme et lues par SimpleOpt.

La conséquence est qu'il serait probablement difficile et inefficace d'utiliser une parallélisation en mémoire partagée due à la nécessité de se prémunir des problèmes liés aux accès concurrents. L'utilisation de Pthreads⁸ et openMP⁹ n'a pas été explorée.

Après cet aperçu du code et du fonctionnement de toulbar2, il est temps de s'intéresser à l'algorithme [Hybrid Best-First Search \(HBFS\)](#) lui-même.

⁸Pthread ou POSIX threads API de bas niveau pour la programmation multithreadée. Permet un contrôle précis du parallélisme : https://en.wikipedia.org/wiki/POSIX_Threads

⁹OpenMP (Open Multi-Processing) API de plus haut niveau que pthread pour le calcul parallèle sur architecture à mémoire partagée. Plus portable et facile à utiliser que Pthreads : <https://fr.wikipedia.org/wiki/OpenMP>. A ne pas confondre avec openMPI, qui est une des implémentations, comme MPICH, d'un standard de parallélisation appelé, Message Passing Interface, destiné aux architectures à mémoire distribuées.



5. Hybrid Best-First Search

Résumé

Ce chapitre présente en détail l'algorithme HBFS séquentiel. Il combine une file de priorité *open* qui contient les nœuds ouverts, c'est à dire les nœuds dont le sous-arbre n'a pas encore été totalement développé, et un Branch and Bound avec parcours en profondeur d'abord, borné par un nombre de backtracks¹ fixé dynamiquement pour essayer de trouver le meilleur compromis entre *propagation* et *diversification*.

Dans le cadre de ce PFE, on considère que l'algorithme HBFS décrit ci-après prend en entrée un problème wcsp, au format *.wcsp, qui représente une réseau de fonction de coûts, c'est à dire un hypergraphe, qu'il s'agit d'*analyser* à l'aide d'un arbre de décisions ou de recherche développé au fur et à mesure que des décisions sur les domaines des variables sont prises. On restera à un niveau abstrait sans décrire les problèmes pratiques qu'ils modélisent qui relèvent de domaines variés : biologie, etc.

5.1 Definitions

5.1.1 Depth-First Search

Désigne une recherche dans un graphe où les nœuds sont explorés jusqu'à atteindre la profondeur maximale avant d'effectuer un retour arrière et de rechercher à nouveau la profondeur maximale. Cf. par exemple : https://en.wikipedia.org/wiki/Depth-first_search.

5.1.2 Breadth-First Search

Désigne une recherche dans un graphe où les nœuds sont explorés dans l'ordre déterminé par leur profondeur : tous les nœuds de profondeur 1 sont d'abord explorés, puis ceux

¹Un backtrack désigne un retour arrière dans l'arbre de recherche.

de profondeur 2, etc. Une implémentation non récursive d'une telle recherche utilise une pile. Cf. par exemple : https://en.wikipedia.org/wiki/Breadth-first_search. Cette pile contient l'ensemble des nœuds situés sur la *frontière* qui sépare les nœuds explorés de ceux qui ne le sont pas encore.

5.1.3 Best-First Search

Désigne une recherche dans un graphe où les nœuds les plus prometteurs sont explorés en premier. Une heuristique en détermine le caractère prometteur. Une implémentation non récursive d'une telle recherche utilise une file de priorité. Cf. par exemple : https://en.wikipedia.org/wiki/Best-first_search.

5.1.4 Nœud ouvert

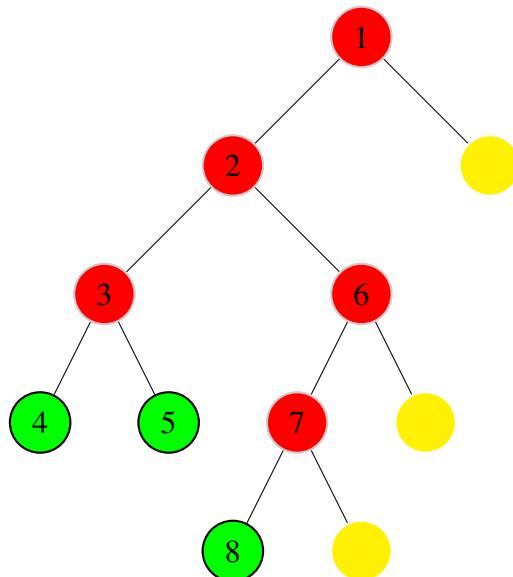


FIGURE 5.1 : Arbre binaire de décision partiellement exploré par un DFS avec une limite de backtrack, $Z = 3$. Les nœuds verts sont les feuilles de l'arbre donnant lieu à une affectation complète des variables. Les nœuds jaunes non cerclés sur les branches droites de l'arbre sont les nœuds ouverts i.e. le sous arbre associé n'a pas encore été exploré. Ces nœuds sont placés dans la file *open* par le DFS lorsqu'il atteint la limite $Z = 3$. En rouge, ce sont les nœuds fermés. Les nombres indiquent l'ordre de parcours DFS de l'arbre.

5.1.5 Backtrack adaptatif

Désigne une heuristique qui permet de trouver un compromis entre deux objectifs contradictoires, la diversification et les propagations répétées :

1. Diversification : le nombre de backtracks doit être suffisamment faible pour diversifier la recherche. En effet, l'arrêt du DFS donne la possibilité de choisir un nouveau nœud dans la file *open* plus prometteur situé à un endroit différent dans la frontière de

l'arbre de recherche. Cela permet donc de reconsidérer les choix faits précédemment et d'augmenter la probabilité de trouver une nouvelle meilleure solution.

2. Propagation : Le nombre de backtracks doit être suffisamment élevés car à chaque fois qu'un nœud est prélevé dans la file *open* deux opérations doivent être effectuées :
 - (a) Le rejet des $v.\delta$ décisions qui ont conduit à ce dernier,
 - (b) Le re-calcul de w_\emptyset à partir de la racine (Soft Arc consistency re-enforcement)

5.1.6 Hybrid Best-First Search

La recherche hybride en meilleur nœud d'abord (HBFS) combine deux approches : un **Depth First Search with B&B (DFBB)** et une **Breadth First Search (BFS)**.

Les différentes étapes de l'algorithme de recherche dans l'arbre de décision associé au graphe du problème à résoudre sont développées ci-dessous. La figure 5.2 et l'algorithme 1 illustrent également son fonctionnement décrit dans l'article *Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP* [3] dont on reprend les notations pour en faciliter une éventuelle lecture.

En particulier, un nœud v possède deux attributs qui caractérise son *état* :

1. $v.\delta$: la séquence de décisions, ou points de choix, qui ont conduit à ce nœud dans l'arbre de recherche ; un nœud est caractérisé par un chemin dans cet arbre. Il est associé à une affectation partielle des variables : A_v , voire à une affectation complète si le nœud est une feuille.
2. $v.lb$: la borne inférieure locale au nœud utilisée dans le HBFS pour élaguer l'arbre et pour calculer la borne inférieure globale clb . Cette dernière étant égale au plus grand des $v.lb$ des nœuds *frontière*. Son calcul est aisément puisque les nœuds sont classés par lb décroissant dans la file de priorité *open*.

Le gap d'optimalité est l'intervalle noté $[clb, cub]$. cub est la meilleure valeur de la fonction de coûts globale courante trouvée, clb est la borne inférieure globale décrite ci-dessus.

1. Initialisation :

- Le nœud racine est initialisé avec $v.\delta = \emptyset$ et $v.lb = clb = w_\emptyset$; en effet, à ce stade, la borne inférieure locale au nœud lb est aussi la valeur globale. La meilleure valeur courante est initialisée par consistance d'arc locale. Dans HBFS, la méthode par défaut pour calculer cette borne inférieure w_\emptyset est l'**Existential Directional Arc Consistency (EDAC)**[10].
 - Le nœud racine est placé dans la file de priorité *open*.
 - La borne supérieure cub , qui est la meilleure valeur courante pour la fonction de coûts globale, est fixée à k^2 ou à un coût très grand.
2. Le nœud est retiré de la file pour être traité par le DFBB, i.e. le Branch and Bound avec parcours DFS,
 3. Après un certain nombre de backtracks³, le DFBB est arrêté,

² k est ce qu'on peut appeler l'élément absorbant dans la structure de valuation du problème wcsp. Avant la recherche, c'est la meilleure valeur que l'ont ait.

³le DFBB effectue une recherche bornée par le nombre de backtracks autorisés Z. Z n'est pas constant

4. L'algorithme DFBB place alors dans la file *open* les nœuds ouverts⁴. Ce sont les nœuds qu'il n'a pas eu le temps d'explorer,
5. La file de priorité se charge de classer les nœuds selon leur borne inférieure croissante ou selon leur profondeur décroissante en cas d'égalité. Le premier nœud de la file est censé être le plus prometteur. En tout cas, il permet de calculer efficacement la borne inférieur globale *clb*. Le second critère du tri lexicographique avec profondeur décroissante sélectionne le nœud le plus "profond" davantage susceptible d'aboutir à une affectation complète donc à un nouvel UB inférieur à *clb*, donc à des possibilités potentiellement accrues d'élagage,
6. Si le DFBB parvient à une feuille de l'arbre de recherche⁵, une affectation complète des variables du problème est trouvée et permet de calculer une nouvelle valeur pour la fonction de coûts globale. Si cette valeur est meilleure i.e. inférieure à la valeur courante alors elle est mise à jour i.e. l'*incumbent value is updated*,
7. Le meilleur nœud est retiré de la file *open*,
8. L'état du nœud est re-calculé (restauration) ainsi que sa borne inférieure avant d'être traité par le DFBB qui ajoute à la file *open* des nœuds ouverts s'il n'a pas eu le temps de traiter l'ensemble du sous-arbre associé,
9. L'opération se poursuit tant que la file *open* n'est pas vide,
10. Les nœuds tels que la borne inférieure locale $lb \geq cub$ ne sont pas traités. On dit que l'arbre de recherche est élagué (pruning).

mais adaptatif. L'heuristique d'adaptation joue un rôle fondamental dans les performances de HBFS. En outre, en cas de manque de mémoire, hbfs passe en mode pure DFS jusqu'à ce que la pénurie cesse.

⁴Un nœud ouvert peut être vu comme un sous-problème ou un sous-arbre qu'il reste à explorer par opposition à un nœud fermé.

⁵La séquence de décisions δ est telle quelle détermine une affectation complète.

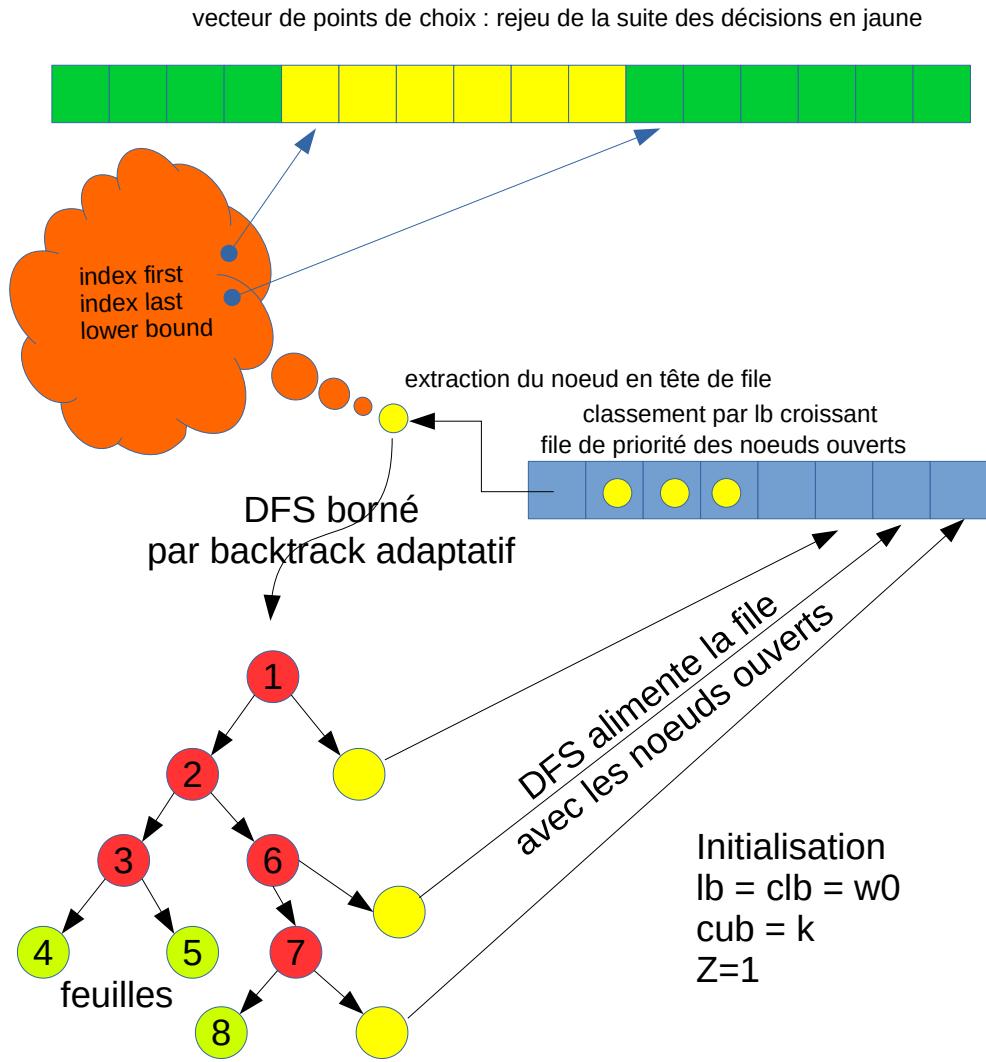


FIGURE 5.2 : Illustration de l'algorithme : un nœud est retiré de la file *open*, donné au DFS qui retourne des nœuds ouverts dans la file *open*, qui les classe par *clb* croissants. Un vecteur de décision, i.e. de points de choix, optimise l'espace mémoire en évitant les redondances. En effet, deux nœuds peuvent avoir en commun un même chemin dans l'arbre de décisions. Il est plus économique en mémoire d'utiliser deux index, *first* et *last*, qui pointent sur le chemin du nœud que de mémoriser dans chaque nœud le chemin complet.

Algorithm 1: Hybrid Best-First Search. Initial call : HBFS(w_\emptyset, k) with $Z = 1$.

```
/* clb : borne inf. globale courante, cub : borne sup. globale
courante */  
/* [clb,cub] : gap d'optimalité */  
1 Function HBFS(clb, cub) : pair(integer,integer)  
    /* initialisation du nœud racine */  
2   open := v( $\delta = \emptyset, lb = clb$ ) ;  
    /* condition d'arrêt  $clb \geq cub$  ou open vide */  
3   while (open  $\neq \emptyset$  and clb  $< cub) do  
4     v := pop(open) ;           /* on pop le nœud le plus "prometteur" */  
5     Restore state v. $\delta$ , leading to assignment  $A_v$ , maintaining local consistency ;  
6     NodesRecompute := NodesRecompute + v.depth ;  
7     cub := DFS( $A_v, cub, Z$ )/* puts all right open branches in open */ ;  
8     clb := max(clb, lb(open)) ;           /* calcul du clb global courant */  
9     /* heuristic Z adaptatif */  
10    if (NodesRecompute > 0) then  
11      if (NodesRecompute/Nodes >  $\beta$  and  $Z \leq N$ ) then  $Z := 2 \times Z$ ;  
11      else if (NodesRecompute/Nodes <  $\alpha$  and  $Z \geq 2$ ) then  $Z := Z/2$ ;  
12    return (clb, cub);$ 
```



6. Stratégies de parallélisation

Résumé

Ce chapitre présente quelques notions et éléments de vocabulaire en usage dans la communauté du calcul haute performance (HPC). Plusieurs pistes de parallélisation de l'algorithme **Hybrid Best-First Search (HBFS)** sont considérées. Deux sont retenues :

1. L'[Embarrassingly Parallel Search \(EPS\)](#),
2. Le mécanisme de coordination Master-Worker.

6.1 Introduction

Les techniques de parallélisation constituent un domaine important tant sur le plan théorique que pratique. Depuis l'avènement des architectures parallèles dans les années 2000, il est devenu essentiel de modifier la façon de concevoir et d'implémenter les algorithmes au besoin en les simplifiant pour mieux exploiter par force brute les processeurs multi-coeurs que l'on trouve dans les PC grands public et les smartphones, les fermes de calcul (clusters), les ressources du cloud, les grilles de calcul¹ ou les superordinateurs devenus endémiques.

Dans le domaine de la recherche opérationnelle, et en particulier en optimisation linéaire en nombres mixtes ([MILP](#)), les progrès importants effectués sur les algorithmes séquentiels ont ralenti la recherche sur les algorithmes parallèles. En outre, la sophistication des programmes séquentiels les rend difficiles à paralléliser. Cf. le rapport technique *Parallel Solvers for Mixed Integer Linear Optimization* [24].

En particulier, ces derniers dépendent beaucoup de l'ordre de traitement et la parallélisation a tendance à casser ces séquences d'exécution pourtant sélectionnées pour leur efficacité. Enfin, le parallélisme a un coût et les résultats obtenus ne sont pas toujours à la hauteur des attentes. La force brute ne remplace pas, par exemple, l'exploitation de la

¹Grid computing

structure d'un problème mais peut la compléter pour résoudre des instances plus difficiles. Ces considérations s'appliquent également à un solveur de réseaux de fonctions de coûts comme toulbar2.

On explore dans ce qui vient à la fois le vocabulaire du monde parallèle et les options de parallélisation de l' [Hybrid Best-First Search \(HBFS\)](#).

6.2 Définitions

Dans cette section, on trouvera quelques définitions de termes appelés à être utilisés par la suite.

6.2.1 Processus

Programme en cours d'exécution, qui bénéficie de ressources attribuées par le système d'exploitation, en particulier d'un espace mémoire privé et protégé. Il possède un état, un contexte qui évolue au cours du temps. Il peut être en cours d'exécution donc en partie chargé dans les caches et les registres du processeur, dans la file d'exécution, en attente de ressources ou d'événements, ou encore à l'état zombi si le processus parent ne le termine pas. Il subit les *décisions* de l'ordonnanceur du système d'exploitation tel que son éviction du processeur avec mémorisation de son contexte. Un processus peut recevoir des signaux et réagir en conséquence. On parle aussi de processus lourds car la sauvegarde du contexte est chronophage par opposition à un processus léger ou thread.

6.2.2 Thread

Processus léger qui s'exécute en parallèle avec d'autres threads au sein d'un processus lourd. Chaque thread possède sa propre pile d'exécution mais partage l'espace mémoire du processus lourd avec les autres threads. Les commutations de contexte des threads sont rapides mais il faut gérer l'accès concurrent aux ressources ce qui ajoute des parties de code essentiellement séquentielles. Les outils les plus connus pour gérer les exclusions et la synchronisation sont les mutex, les sémaphores et les moniteurs de Hoare. Des programmes hybrides combinent le multi-threading utilisant Pthreads (POSIX threads) et OpenMP (Open Multi-Processing) dans le cadre du modèle de mémoire partagée avec OpenMPI (Message Passing Interface), API (Application Programming Interface) pour échanger des messages entre processus dans le cadre de mémoire distribuée.

6.2.3 Socket

Désigne la puce, le *mille-patte*, le circuit intégré et son boîtier visible sur les cartes mères. Dans les architectures actuelles, un *socket* accueille plusieurs processeurs qu'on nommera également cœurs ou cpu (Central Processing Unit).

6.2.4 Parallelism

Le parallélisme dont il est question ici concerne celui accessible au programmeur qu'on peut désigner sous le terme Task-parallelism en référence à sa granularité de traitement. On

distribue, en effet, des sous-problèmes à chaque processeurs. Il existe d'autres parallélisation avec une granularité plus fine : l'ILP et le TLP.

6.2.5 Instruction-Level Parallelism

L'*Instruction-Level Parallelism* (ILP) concerne la parallélisation au niveau des *instructions* compréhensibles pour le processeur, ce qui correspond à une ligne du code en langage Assembleur par opposition aux instructions d'un langage de haut niveau comme le C ; en C, la simple incrémentation d'un entier *i* n'est pas *atomique*. Elle correspond à une seule instruction en C mais à plusieurs en langage assembleur. A ce niveau de granularité, deux approches sont utilisées : le *pipelining* et le *Multiple Issue*. La première utilise le principe du travail à la chaîne, la seconde multiplie les unités de traitement en entrée, additionneur par exemple, pour que le processeur puisse recevoir plusieurs instructions à la fois. Dans ce dernier cas, si l'ordonnancement des instruction est réalisé dynamiquement et pas seulement à la compilation, on parle d'architecture *superscalaire*. Les processeurs actuels combinent *Pipelining* et *Multiple Issue*. Les processeurs *vectoriels* concernent, quant à eux, un autre type d'architecture qui permet de manipuler des vecteurs comme de simples nombres. Si l'addition d'un vector de taille 10 prend 10 cycles d'horloge sur une architecture scalaire, elle n'en requerra qu'un seul sur un processeur vectoriel.

6.2.6 Thread-Level Parallelism

Le *Thread-Level Parallelism* possède une granularité plus grossière que l'ILP puisqu'un thread est constituer d'un ensemble d'instructions processeur qui peut être élevé. L'idée générale se résume dans le terme *Multithreading* qui désigne le fait de "virer" un thread du processeur si ce dernier ne travaille pas au lieu de chercher à paralléliser le dit thread.

6.2.7 Hyperthreading

L'hyperthreading (HT) est une technologie propre à Intel qui utilise un type de Multithreading particulier, le Simultaneous Multithreading (SMT), qui cherche à exploiter les unités fonctionnelles multiples que l'on trouve dans les processeurs à architecture superscalaire actuels pour exécuter plusieurs threads à la fois sur le même cœur. On peut observer un gain de 15-20% des performances, dû à l'hyperthreading. L'hyperthreading peut être désactivé à des fins de *benchmark* par exemple. Avec L'HT activé, le système d'exploitation (OS) voit le nombre de cœurs logiques (commande linux/bash : *nproc*). A titre d'illustration, la sortie commentée de la commande *lscpu* est indiquée ci-dessous. La terminologie est cependant trompeuse : utilisation du terme *processeur(s)* au lieu de *thread*. Le terme *socket*, sous-entendu *Chip Socket*, désigne le support, généralement blanc présent sur la carte mère qui accueille le boîtier noir (Chip) protégeant le circuit intégré multi-cœurs.

Architecture :	x86_64
Mode(s) opératoire(s) des processeurs :	32-bit, 64-bit
Boutisme :	Little Endian
Address sizes:	39 bits physical, 48 bits virtual

Processeur(s) :	8 => 8 THREADS OU COEURS LOGIQUES
Liste de processeur(s) en ligne :	0-7 => DE RANG 0 À 7
Thread(s) par cœur :	2
Cœur(s) par socket :	4 => 4 cœurs REELS
Socket(s) :	1 => UN SEUL SUPPORT DE CPU
Nœud(s) NUMA :	1 => UN SEUL NOEUD MEMOIRE
DONC LES 8 COEURS LOGIQUES AURONT LE MEME TEMPS D'ACCES MEMOIRE	
<=> ACCES UNIFORME À LA MEMOIRE.	
Identifiant constructeur :	GenuineIntel
Famille de processeur :	6
Modèle :	158
Nom de modèle :	Intel (R) Core (TM) i7-7700HQ CPU @

6.2.8 Non Uniform Memory Access

Le terme *Non Uniform Memory Access*(NUMA) décrit l'architecture en mémoire partagée d'un système où le temps d'accès à la mémoire RAM varie selon la position relative des processeurs. Ainsi, la sortie de la commande *numactl –hardware* présentée ci-dessous indique, qu'il n'existe qu'un nœud mémoire de rang 0 et que tous les coeurs logiques sont affectés à ce nœud. Le temps d'accès sera donc uniforme et le programmeur, sur cette architecture, n'aura pas à se soucier, pour des raisons de performances, de la position de ses données en mémoire².

```
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 7854 MB
node 0 free: 2990 MB
node distances:
node    0
```

6.2.9 Algorithme parallèle

Généralement, désigne la partie de l'algorithme qui contrôle le parallélisme tels qu'affection de tâches aux processeurs, déplacement des données entre processus et non le *programme parallélisé* dans sa globalité.

6.2.10 Algorithme séquentiel sous-jacent

Ensemble de sous tâches associées à l'algorithme purement séquentiel telles que les opérations de branching (séparation) et du calcul des bornes (évaluation) dans le B&B. Ces éléments constituent souvent des boîtes noires pour l'algorithme parallèle.

²Cela peut expliquer des performances décevantes sur une architecture supposée plus puissante et une partie de la variance des temps d'exécution.

6.2.11 Unité d'exécution (UE)

Terme générique qui désigne indifféremment un processus ou un thread.

6.2.12 Tâche

Un programme, une tâche ou Task en anglais, décrit ce qui doit être fait. C'est un concept statique contrairement à la notion de processus, essentiellement dynamique. En soit, une tâche ne fait rien. C'est une simple séquences d'instructions. Le traitement réel est effectué par un processus qui est une "tâche" chargée en mémoire et prise en charge par le système d'exploitation. On distinguera aussi le processus du processeur physique, ou cœur, ou cpu, sur lequel il s'exécute. Le processus utilise un processeur qui est une des ressources à sa disposition.

6.2.13 Worker

Processus qui effectue un traitement sur demande en s'exécutant sur un processeur physique.

6.2.14 Master

Processus principal qui distribue le travail, reçoit et synthétise les résultats. En programmation MPI, on lui attribue le rang 0 pour le distinguer clairement des workers.

6.2.15 Communicateur

Autre terme qui désigne un processus dans le contexte du standard Message Passing Interface (MPI). Les entités qui communiquent entre elles sont effet des processus qui s'échangent des messages. C'est MPI qui se charge d'affecter les processus aux cpu. MPI ne permet pas d'affecter un processus de rang i à un processeur de rang j particulier. La plupart du temps, la taille de l'ensemble des communicateurs est fixée à une valeur égale au nombre de coeurs. Généralement, sur une machine à 24 coeurs réels, on fixe le nombre de processus également à 24 via la commande :

```
mpirun -np 24 ./toulbar2 404.wcsp
```

Avec *mpirun*, commande qui exécute le programme parallèle et *mpic++*, commande (wrapper) qui compile le programme parallèle. La différence entre processus et processeur devient inutile à moins que le système d'exploitation ne s'amuse à permuter les processus sur les processeurs où un processus i qui libère de gré ou de force un cœur j reprendrait son traitement sur un coeur k .

6.2.16 Wallclock time

Temps global d'exécution d'un algorithme. C'est le temps chronométré entre le lancement du programme et sa terminaison. C'est le temps donné par la ligne *real* de la commande *time* sous Linux. On le nomme également Elapsed time, Elapsed real time, real time, wall-clock time, ou **wall time**.

6.2.17 Speedup

Souvent définie comme le rapport du temps d'exécution de l'algorithme séquentiel sur le temps nécessaire à l'algorithme parallélisé pour résoudre le même problème. Un algorithme peut être plus performant mais moins *scalable* qu'un autre. Cf. la notion d'[échelonnabilité](#).

6.2.18 Efficacité parallèle

Désigne le speedup par nombre de cœurs. Speedup S et Efficacité E caractérisent la performance globale du programme. D'autres métriques ou critères peuvent être utilisées par exemple la capacité d'un programme à résoudre des instances difficiles.

6.2.19 Fraction séquentielle

Désigne la proportion ρ du code d'un programme parallélisé qui s'exécute séquentiellement.

6.2.20 Amdahl's law

Cette loi indique que le speedup théorique de Amdahl S_a est majoré par l'inverse de la fraction séquentielle x . En effet, on a :

$$S_a = \frac{1}{x + \frac{1-x}{c}} \leq \frac{1}{x}$$

Ainsi, si on exécute un programme parallélisé sur un cluster équipé de 3000 cœurs et si $x = 0.01$, le speedup ne pourra être supérieur à 100[23].

nb : la loi de Gustafson fournit une formule qui prend en compte le fait qu'une machine parallèle permet de traiter des problèmes plus gros mais ne prend pas en compte l'augmentation des coûts de communications. Si c désigne le nombre de cœurs disponibles, la loi de Gustafson s'écrit :

$$S_g = c + (1 - c)x$$

6.2.21 Scalabilité ou échelonnabilité

Capacité d'un algorithme à tirer partie d'une augmentation des ressources, principalement du nombre de cœurs. On cherchera, en général, à obtenir des performances croissant linéairement avec le nombre de processeurs.

6.2.22 Ramp-up, Ramp-down

Un algorithme parallélisé se déroule en 3 phases :

1. *Ramp-up phase* ou phase de montée en puissance (décollage, montée),
2. *Primary phase* ou phase principale (régime de croisière),
3. *Ramp-down phase* ou phase de terminaison (descente, atterrissage).

La phase de ramp-up débute avec le programme et se termine lorsque tous les processeurs/workers se sont mis au travail. La phase de ramp-down débute lorsqu'au moins un worker devient inoccupé de manière permanente et se termine lorsque la solution globale est affichée.

6.2.23 Overhead ou surcoût

Quantité de travail mesuré en unité de temps nécessaire à la parallélisation qui ne serait pas effectuée dans l'algorithme séquentiel. On peut distinguer l'overhead :

1. de communication : temps passé à déplacer les données entre processeurs,
2. d'inactivité due aux phases de ramp-up et ramp-down e.g. où tous les processeurs ne sont pas alimentés en travail ou à un load balancing défaillant,
3. dû à la latence associée à l'attente de données ou au temps d'allocation de mémoire,
4. dû au travail redondant : temps perdu à répéter les mêmes tâches alors que cette redondance n'existerait pas dans le programme séquentiel.

Une parallélisation efficace doit contrôler l'overhead mais il n'existe pas de solution standard car il y a une forte dépendance aux propriétés de l'algorithme et à l'architecture matérielle. En particulier, l'overhead de communication dépendra du type de réseaux entre processeurs physique³, du nombre de cœurs par socket, du nombre de threads par cœur. Une machine à 8 processeurs peut comporter 1 socket, 4 cœurs, 2 threads par cœur (hyperthreading).

6.2.24 Granularité

Quatre principaux types de granularités, classée ici de la plus grossière à la plus fine, peuvent caractériser les algorithmes parallèles. La granularité définit la quantité de travail atomique de l'algorithme.

1. *Tree parallelism* : Le parallélisme porte sur la recherche en parallèle dans plusieurs arbres avec des stratégies différentes ; le même espace d'état est parcouru au même moment. C'est l'approche portfolio qui vise à déterminer l'algorithme le plus efficace pour une instance donnée ; une course est lancée entre les solutions ou les stratégies.
2. *Subtree parallelism* : plusieurs sous-arbres sont explorés simultanément mais indépendamment sans échange d'informations. Contexte de l'[Embarrassingly Parallel Search \(EPS\)](#).
3. *Node parallelism* : Plusieurs nœuds sont traités en parallèle avec un contrôle centralisé et des échanges d'informations. Le mécanisme de coordination Master-Worker constitue une implémentation immédiate. Le master distribue les nœud à traiter présents dans une file Q et collecte les résultats et les nœuds, dits ouverts ou actifs, à traiter produits par le worker. C'est le mode de parallélisation le plus courant. Par rapport à la granularité subtree, on constate qu'un sous arbre est aussi un nœud, puisqu'un nœud représente un sous arbre en voie de développement.
4. *Subnode parallelism* : parallélisation des traitements effectués au sein de chaque nœud. Ces traitements varient en fonction des problèmes et des solutions utilisées.

³Un processeur physique, au sens de boîtier accueillant un circuit intégré composé éventuellement de plusieurs cœurs est nommé *socket* sous linux. Cf. commande linux *lscpu*.

6.2.25 Adaptivity

Capacité d'un programme à effectuer un traitement différent en fonction du contexte. Par exemple, capacité à changer la granularité du traitement, en fonction des 3 phases : ramp-up, principale et ramp-down.

6.2.26 Partage de connaissances

Données générées durant la recherche qui sont utiles aux autres workers : bornes, solutions réalisables produites par un worker utiles à d'autres

6.2.27 Déterminisme

capacité d'un algorithme à produire la même solution quel que soit le nombre d'exécution. Le Déterminisme "fort" impose un ordre des opérations toujours identique mais ceci est difficile à garantir notamment dans le cas de l'hyperthreading.

6.2.28 Synchronisation

Méthode, en programmation concurrente, qui permet aux workers de travailler sur un état commun pour assurer la cohérence et/ou l'efficacité du programme. Implémentée par exemple sous forme de barrières où les processus s'attendent en un point donné. Elle est nécessaire pour agréger des résultats intermédiaires ou finaux et permet de renforcer le déterminisme. Inconvénient : les workers les plus rapides doivent attendre les plus lents donc plus il y a de workers plus il y a perte de temps et la scalabilité est alors mauvaise. **La synchronisation est donc à éviter dans la mesure du possible.**

Coordination

La coordination des actions effectuées par un algorithme introduit un compromis à trouver entre l'augmentation de l'overhead et l'amélioration des performances de l'algorithme séquentiel sous-jacent. La coordination ne nécessite pas forcément une stricte synchronisation. Un exemple de schéma de coordination est le load balancing.

Topologie

Agencement global du réseau. La latence dépend de la topologie. Le cluster du data center genotoul de l'INRA Toulouse possède une architecture Fat Tree : https://en.wikipedia.org/wiki/Fat_tree.

Nœud

Un cluster est un ensemble d'ordinateurs homogènes connectés par un réseau selon une certaine topologie. Chacun de ses ordinateurs constitue un nœud. On voit l'image d'un graphe dont les arêtes sont les connexions réseau et les ordinateurs les nœuds.

Latence

Temps de transmission d'un message vide entre deux nœuds.

Framework

Lorsque le programme parallèle qui gère le parallélisme et le programme séquentiel sous-jacent sont suffisamment découpés il est possible d'utiliser un framework dont le rôle sera de gérer le parallélisme d'un algorithme séquentiel. Par exemple, la fonction de branching du branch and bound pourrait faire partie de l'interface du framework sous forme de signature, charge à l'utilisateur de ce framework d'implémenter le code séquentiel correspondant dans cette fonction. Un framework peut se présenter sous forme d'un ensemble de classes de base qu'il s'agit de dériver et compléter les méthodes. L'avantage est d'éviter d'avoir à implémenter soi-même des stratégies de load balancing et autres coordinations nécessaires aux parallélisme.

6.2.29 Compromis ou Trade-off

Le partage de données entre processus telle que la meilleure solution courante améliore l'efficacité du traitement mais augmente l'overhead. Ceci d'autant plus que la taille des données échangées est conséquente et le réseau lent, avec en outre une latence importante. Un compromis, ou trade-off, doit être déterminé. A noter, que sur le plan pratique, lors de l'utilisation de Boost.MPI, la taille des données transmises n'ont pas altéré les performances sur PC et sur serveur. Sur le cluster, même avec un réseau InfiniBand à 56 GB, c'est une autre histoire

6.2.30 Load balancing

L'équilibrage de charge désigne la distribution du travail utile de manière à minimiser les temps d'inactivité des workers. Il est caractérisé par :

1. la fréquence à laquelle il est effectué,
2. par les informations qui doivent être déplacées,
3. par sa nature statique, load balancing effectué une seule fois au début
4. ou par sa nature dynamique, load balancing effectué durant tout le programme.

Static load balancing

Le rapport technique [24] explicite quatre méthodes statiques d'équilibrage de charge :

1. Root initialization : La plus utilisée pour sa facilité d'implémentation. Une processus traite le nœud racine et crée les nœuds enfants. Ces enfants sont distribués à tous les workers et cette technique est répétée itérativement jusqu'à ce que tous les workers soient occupés. Cette méthode est plus efficace si Le temps de calcul d'un nœud est faible et le branchement est élevé de manière à mettre au travail les workers le plus rapidement possible. Elle est moins efficace si le nombre workers est important.
2. Enumerative initialization : le nœud root est transmis à tous les workers qui développent chacun le même arbre. Il faut donc que le programme soit déterministe. Lorsque le nombre de nœuds enfant est égale au nombre de workers, le worker i sélectionne le nœud i et efface le reste.
3. Selective initialization : La racine est broadcastée à chaque worker qui génère un seul chemin. Les chemins de chaque worker doit être distincts d'où la nécessité d'un

schéma sophistiqué pour assurer que tous les worker travaillent sur des chemins différents.

4. Direct initialization : pas de création explicite de l'arbre. A la place, ce sont les worker qui créent directement un nœud à partir d'une certaine profondeur.
5. Autres méthodes implémentées dans des framework : racing ramp-up dans UG, spiral et root initialization à deux niveaux dans le solveur CHiPPS.

Dynamic load balancing

Les schémas de load balancing sont fortement dépendants du type d'algorithme. Ils sont caractérisés par le degré de centralisation et par l'initialisateur de l'équilibrage [24] : processus dédiés (work-sharing), fournisseur ou receveur de travail (work-stealing).

Quelques modèle de Dynamic load balancing :

1. *Asynchronous Round Robin* : exemple avec 8 workers de rang 0 à 7. Le worker 0 maintient une variable *target* qui pointe sur le worker auquel demander du travail dès qu'il sera inoccupé. Exemple : Initialement target = $(i+1) \bmod 8 = (0+1) \bmod 8 = 1$. Si le worker 0 passe en mode idle, il envoie donc une requête au worker 1 et incrémenté modulo 8 sa variable target. target := target++ mod 8 = 2, etc.
2. *Nearest neighbor* : A chaque worker est associé un ensemble de "voisins". Si un worker manque de travail, il envoie une requête à ses "voisins". Avantage : communications localisées. Inconvénient : possible temps long pour atteindre le load balancing global.
3. *Random Polling Scheme* : Un worker envoie une requête à un worker choisi au hasard. équiprobabilité de choix d'un worker. Très simple et assez efficace dans certaines applications.

Le travail transmis entre workers fait partie des "connaissances" échangées ; il existe donc de nombreux compromis à trouver. Un load balancing agressif réduit le temps d'inoccupation des workers et le travail redondant mais augmente l'overhead. Les solveurs modernes emploient des stratégies plus sophistiquées et adaptatives[24].

6.3 Mémoire partagée contre mémoire distribuée

6.3.1 Solveurs existants

Les solveurs commerciaux Xpress, Gurobi, Cplex sont parallélisés en mémoire partagée. Les solveurs libres CBC, Glpk, Lpsolve, SYMPHONY, DIP et SCIP possèdent pour certains des algorithmes parallèles.

6.3.2 Choix pour toulbar2

La conception de toulbar2 basée sur la création d'un objet "solver" dont les attributs globaux sont modifiés en cours de résolution du problème rend difficile l'utilisation d'une mémoire partagée⁴. La gestion des accès concurrents aboutirait de toute façon à augmenter la part

⁴En outre, toutes les méthodes de toulbar2 ne sont pas forcément réentrant et/ou thread-safe : <https://fr.wikipedia.org/wiki/R%C3%A9entrance>.

séquentielle du programme parallélisé au niveau des sections critiques⁵ avec une perte en terme de performances. De ce fait, les pistes explorées portent uniquement sur les modèles à mémoire distribuée même sur les architectures matérielles intrinsèquement à mémoire partagée tel qu'un PC de bureau classique. Ce constat oriente de fait le choix technique vers le standard **Message Passing Interface (MPI)**, standard le plus couramment utilisé dans la communauté du calcul haute performance (HPC). Les implémentations libres les plus connues sont OpenMPI et MPICH.

Après avoir exposé quelques notions sur le parallélisme et éliminé l'option "mémoire partagée", la suite de ce chapitre décrit les stratégies envisagées.

6.4 Stratégies envisagées

6.4.1 Embarrassingly parallel search

L'**Embarrassingly Parallel Search (EPS)**⁶ désigne une méthode de parallélisation *naturelle* adaptée aux programmes décomposables en tâches indépendantes. Les problèmes qui peuvent être décomposés en parties totalement indépendantes se prêtent donc particulièrement bien à son utilisation. L'EPS consiste à produire un nombre de sous-problèmes important, par exemple sur une machine à k processeurs, on produira $30k$ sous-problèmes. Ils sont alors placés dans une file où chaque worker viendra se servir lorsqu'il aura besoin de travail. On doit, cependant, éviter de produire des sous-problèmes qui auraient été éliminés par le processus de propagation du solveur. On pourra consulter les références suivantes pour une description détaillée de la méthode [26, Régin et al. 2013] de [25, Régin et al. 2018].

En outre, les architectures de type cluster ou grille de calculs constituent un terrain de jeu naturel pour les programmes compatibles EPS car constitués de nœuds connectés par des réseaux qui ne sont pas forcément très rapides⁷ surtout en comparaison de communications internes à un nœud de la grille de calcul. Comme les seuls échanges de données se déroulent en début et fin de programme, la bande passante reste secondaire alors que le programme peut exploiter un grand nombre de processeurs.

Cette technique de parallélisation est réalisable en trois étapes :

1. Etape de génération qui décompose le problème en sous-problèmes indépendants en nombre plus élevés que le nombre de coeurs disponibles,

⁵En programmation concurrente, une section critique est une portion de code dans laquelle il doit être garanti qu'il n'y aura jamais plus d'un thread simultanément. Il est nécessaire d'utiliser des sections critiques lorsqu'il y a accès à des ressources partagées par plusieurs threads. Une section critique peut être protégée par un mutex, un sémaphore ou d'autres primitives de programmation concurrente. Puisqu'à un moment donné, jamais plus d'un thread ne peut être actif dans une section critique, le thread la détenant doit la libérer le plus vite possible pour éviter qu'elle ne devienne un goulot d'étranglement. sources : https://fr.wikipedia.org/wiki/Section_critique

⁶Le terme humoristique "Embarrassingly Parallel Computation"(EPC) a été inventé par Geoffrey Fox en 1995. La méthode EPS est basée sur l'EPC.

⁷L'internet par exemple.

2. Etape d'affectation des tâches aux processus,
3. Etape de réduction qui consiste à récupérer les résultats de chaque processus et à les synthétiser.

Remarque : La granularité de l'EPS se situe au niveau du sous-arbre (subtree parallelism) selon la définition donnée dans le rapport technique [24].

Avantages

- Simplicité d'implémentation,
- Probable possibilité de paralléliser des algorithmes de toulbar2 autres que l'**Hybrid Best-First Search (HBFS)**

Inconvénients

- Les sous-problèmes (SP) dans l'arbre de recherche sont indépendants - un SP est résolvable indépendamment des autres - mais l'absence de partage d'informations entre processus, en l'occurrence de la borne supérieure UB, grèvera les performances en empêchant certains élagages de l'arbre de recherche.
- La diversification sera limitée aux sous-problèmes puisque le DFS n'aura accès qu'aux SP présent dans sa file de priorité *open*.
- La phase de réduction devra attendre que le processus le plus lent ait terminé son traitement : effet barrière.

6.4.2 Work stealing

Dans ce modèle, quand un worker a fini d'explorer son sous-arbre, il demande à un autre worker s'il a du travail à lui donner. Si la réponse est positive le worker sollicité scinde dynamiquement son problème en deux et en transmet la moitié au solliciteur. Si la réponse est négative, le worker effectue la même requête auprès d'un autre, jusqu'à ce qu'il arrive à ses fins ou qu'il ait fait le tour des autres workers sans succès. En quelque sorte, le worker affamé *vole* ses collègues[25].

Avantages

- Méthode qui donne de bons résultats en pratique,
- Plus difficile à implémenter que l'EPS.

Inconvénients

- On doit empêcher le vole d'un même problème par deux workers,
- En fin de traitement, les voleurs sont plus nombreux que les travailleurs et il faut gérer ce problème d'overhead de communication qui réduit les performances.

6.4.3 Utilisation d'un framework

Un *framework* constitue une interface entre le solveur et la machine parallèle. On a vu qu'un programme parallélisé était constitué d'un code parallèle qui gère le parallélisme et d'un code séquentiel *sous-jacent*. Or, les paradigmes efficaces de parallélisation ne sont pas forcément simples à implémenter et déboguer. Ainsi, s'il est possible de déléguer le

parallélisme à un framework pour se concentrer sur le découpage du code séquentiel, il serait possible d'aboutir à un gain en performances à peu de frais.

Une liste de framework-candidats potentiels est présentée succinctement ci-dessous à titre d'illustration :

- Ubiquity Generator framework (UG) est composé d'un ensemble de classes C++ qui forment une interface personnalisable permettant la prise en compte de tout solver MILP ou MILNP (Non Linear) : <https://ug.zib.de/>. Le solveur **SCIP**(Solving Constraint Integer Programs) a utilisé UG avec succès.
- COIN-OR High Performance Parallel Search (**CHiPPS-ALPS**) permet d'implémenter des recherches arborescentes parallèles de type B&B. Langage : C++, Granularité : Subtree Parallelism, Paradigme : Master-Hub-Worker[24]. CHiPP parcours l'ensemble du sous-arbre jusqu'à ce qu'un certain critère soit rempli : temps, nombre de noeuds, etc. Utilise MPI. CHiPPS est basé sur Abstract Library for Parallel Search (ALPS), une implémentation abstraite de recherche arborescente : <https://coral.ie.lehigh.edu/~ted/files/talks/CHiPPS-CPAIOR10.pdf>.
- Branch-Cut-Price (BCP) permet d'implémenter des algorithmes parallèles de type B&B pour résoudre des problèmes mixtes en nombres entiers. Granularité : Node parallelism, Paradigme : Master-Worker, <https://github.com/coin-or/Bcp>.
- BiCePs-ALPS : surcouche de ALPS qui apporte le support nécessaire à la résolution de B&B basés sur une relaxation du problème.
- Bobpp : framework de parallélisation pour résoudre des problèmes combinatoires([20], [20]). Ne semble plus maintenu et le code source non publié.
- IBobpp : amélioration du framework Bobpp[19]. Sources non trouvés.

Avantage

- Bénéficier d'implémentation de mécanismes de coordination sophistiqués.

Inconvénients

- Le code de toulbar2 ne serait plus maîtrisé de bout en bout,
- Un temps de prise en main serait nécessaire sans garantie de succès,
- Pas d'acquisition de vraies compétences en parallélisation à moins d'en analyser précisément le code.

6.4.4 Paradigme Master-Worker

Le mécanisme de coordination master-workers est caractérisé par des communications effectuées uniquement à l'occasion de la distribution du travail ou pour recevoir le produit de ce travail. Autrement dit, le master ne dérange pas les workers qui travaillent et les workers ne parlent pas lorsqu'ils sont occupés. Il ne peut pas

L'algorithme simplifié ci-dessous :

- Un processus *master* distribue les sous problèmes (SP) aux *workers*,
- Les processus workers s'exécutent sur un processeur⁸. *Les workers au travail ne*

⁸On utilisera aussi les termes cœur et cpu pour désigner cette ressource calculatoire physique gérée par le système d'exploitation.

communiquent plus avec le master,

- Dès qu'un worker a terminé, il renvoie un ensemble de SP qui peut être vide s'il ne trouve pas de solution,
- Le master range les SP, ou nœuds, dans sa file de priorité *open* et place le worker dans sa file des processus inoccupés Q (idle queue),
- Le master retire de la file Q, le worker situé en tête,
- Le master retire un SP de la file *open* et le transmet au worker.

Remarque : La granularité du procédé de coordination Master-Worker se situe au niveau du nœud (Node parallelism). On utilise ici la définition donnée dans le rapport technique [24].

Inconvénients

- Problème de scalabilité : Les communications sont centralisées, le master deviendra un goulot d'étranglement lorsqu'il ne parviendra plus à distribuer le travail suffisamment rapidement ; les workers trop nombreux vont s'entasser dans la file Q d'où des performances appelées à plafonner. Il n'y a plus de plein emploi.

Avantages

- Simple à implémenter,
- Le load balancing est intrinsèque à ce paradigme car si un worker reçoit un *petit* SP et les autres des *gros* SP, le master lui donnera du travail immédiatement (problème de scalabilité mise à part),
- Mécanisme de coordination le plus employé,
- Semble être un bon choix pour parallélisé l'algorithme **Hybrid Best-First Search (HBFS)** dont la granularité naturelle est le nœud puisqu'il y a communication de nœuds entre le **Breadth First Search (BFS)** et le **Depth First Search with B&B (DFBB)**. Dans HBFS, un nœud est un sous-problème ou un sous-arbre qu'il s'agit de développer.

6.4.5 Autres paradigmes

D'autres mécanismes de coordination ont été considérés. Pour rester synthétique, en voici une liste tirée de [24]. Les communications sont plus complexes. Ces paradigmes tentent de résoudre le problème de goulot d'étranglement du Master-Worker.

- Superviseur-Worker,
- Multiple-Masters-Worker,
- Master-Hub-Worker,
- Self Coordination.

6.5 Bilan

A la suite de cette étude, il a été décidé de suivre les pistes **Embarrassingly Parallel Search (EPS)** et Master-Worker qui vont faire l'objet des deux derniers chapitres de ce rapport.



7. Embarrassingly Parallel Search

Résumé

Ce chapitre résume l'exploration de la piste EPS implémentée en utilisant l'algorithme HBFS séquentiel pour la génération de sous-problèmes de bonne qualité et GNU parallel pour lancer les diverses instances de toulbar2 appliquées à ces sous-problèmes. L'approche EPS, dans ce contexte précis, n'a pas donné des résultats satisfaisants. Le paradigme Master-Worker, de granularité plus fine, et objet du prochain chapitre, semble plus prometteur.

7.1 Généralités sur la méthode de parallélisation embarrassante

Le terme *pittoresque* d'Embarrassingly Parallel Computation (EPC) est dû à Geoffrey Fox et date de 1995. L'[Embarrassingly Parallel Search \(EPS\)](#) s'inspire de ce principe[26]. L'EPS désigne une parallélisation naturelle qui consiste à décomposer un programme, si ce dernier le permet, en tâches indépendantes et à les assigner à des "workers", c'est à dire à des processus, qui s'exécutent sur des coeurs ou processeurs disponibles¹. Les workers résolvent les sous-problèmes (SP) indépendamment, c'est à dire sans échanger d'informations. L'EPS possède une granularité de niveau *sous-arbre* selon la définition donnée en page 14 du rapport technique de l'Université de Lehigh [24].

7.2 Implémentation de l'EPS

Une recherche embarrassante consiste en 3 phases :

¹En général, on affecte un seul processus à un seul worker. De ce fait, le processus worker peut désigner le processeur physique sur lequel s'exécute ce worker.

1. Génération : La phase de génération des sous-problèmes a été implémentée en utilisant une méthode de génération *a priori* d'une part et une génération qui utilise la file *open* de l'algorithme HBFS, d'autre part,
2. Assignation : Cette phase, où les sous-problèmes générés sont assignés aux workers, est déléguée à **GNU Parallel**²,
3. Réduction : Cette phase de calcul de la meilleure solution obtenue par les workers est assurée par des scripts **bash**.

7.3 Génération a priori des sous-problèmes

Une génération a priori ne prend en compte que l'ensemble des variables et leurs domaines. Le problème n'est pas pris en compte. Il s'agit donc de générer tous les tuples possibles à une profondeur donnée de l'arbre. Ce faisant, on ne bénéficie pas d'heuristiques de choix de variables qui peuvent améliorer les résultats.

Génération par produit cartésien

Un programme C++ a été créé qui produit les assignations partielles donnant ainsi un nombre de sous-problèmes tel que

$$|A_{k-1}| < q \leq |A_k|$$

où q est la valeur cible du nombre de sous-problèmes désirés tandis que $|A_k|$ correspond au cardinal du produit cartésien des domaines de k variables[26].

Le nombre de sous-problèmes effectivement générés ne peut pas être quelconque puisqu'on doit obtenir une *partition* du problème global pour couvrir tout l'espace de recherche.

Illustration pratique :

Ci-dessous, on cherche à générer environ 250 sous-problèmes dans le fichier *job_file* et on utilise GNU parallel pour une exécution sur 24 processeurs :

```
./generate 404.wcsp job_file 250
cat job_file | parallel -j24 --eta -k ./toulbar2 404.wcsp {}
```

Le programme *generate*, dont on trouvera le listing en annexe 10.1, génère uniquement des décisions sous forme d'affectation de variables³ avec un ordre des variables par rang croissant : x_0, x_1, \dots

Ainsi, avec le problème 404.wcsp à 100 variables, si on demande 250 sous-problèmes on aboutit à un fichier *job_file* avec $4^4 = 256$ sous-problèmes car les 4 premières variables ont un domaine identique $D = \{0, 1, 2, 3\}$,

²Un script bash *parallel.sh* a été testé mais a donné lieu à des performances décevantes : <https://github.com/sublimino/parallel.sh>. On peut aussi envisager d'utiliser la commande *xargs* avec l'option *-P*.

³Quatre types de décisions de branchement dans l'arbre de recherche sont utilisées dans *toulbar2* et notées $=, \#, <, >$: affectation d'une valeur ($=$), retrait d'une valeur ($\#$), réduction du domaine aux valeurs inférieures strictement à un certain seuil ($<$) et supérieures ($>$).

Ci-dessous, 256 sous-problèmes du problème 404.wcsp sont générés correspondant aux 4 premières variables x_0 à x_3 . L'option $-x= \dots$ de toulbar2 lui permet de construire et résoudre le sous problème fourni.

```
-x=",0=0,1=0,2=0,3=0"  
-x=",0=0,1=0,2=0,3=1"  
-x=",0=0,1=0,2=0,3=2"  
-x=",0=0,1=0,2=0,3=3"  
-x=",0=0,1=0,2=1,3=0"  
-x=",0=0,1=0,2=1,3=1"  
-x=",0=0,1=0,2=1,3=2"  
...  
...
```

Bilan de la génération a priori

Les résultats obtenus sur les problèmes au format wcsp : 404.wcsp et scen06.wcsp sont décevants. Le temps d'exécution est largement majoré par rapport au programme séquentiel. C'est, a priori, une approche trop naïve qui ne bénéficie pas des heuristiques de choix de variables de toulbar2. Comme la génération des sous-problèmes avec HBFS donnait de meilleurs résultats, cette piste n'a pas été explorée davantage.

7.4 Génération des sous-problèmes avec HBFS

L'algorithme HBFS de Toulbar2 utilise une file de priorité, nommée *open*, pour stocker les noeuds ouverts qui constituent les sous-problèmes restant à résoudre, donc naturellement, en fixant une taille maximale pour *open* qui lorsqu'elle est atteinte ou dépassée aboutit à un *dump*⁴ et à la terminaison du programme, on obtient une série de sous-problèmes de *bonne qualité* au sens où ils ont pu bénéficier des optimisations de toulbar2.

En outre de manière à obtenir un load balancing correct les sous-problèmes doivent être *petits* donc suffisamment nombreux. Pour ce faire, on se fixe une valeur N égale par exemple à 30 fois le nombre de cœurs[26]. Lorsque la taille de la file *open* dépasse cette valeur, on extrait les noeuds et on reconstitue la séquence de décisions associée.

Toulbar2 produit un fichier contenant la liste des sous-problèmes. Ces derniers sont transmis à GNU Parallel qui les exécutent sur les processeurs disponibles. Par exemple, deux cents sous-problèmes seront exécutés en parallèle sur 24 processeurs, GNU parallel gérant la file de problèmes et leur affectation aux processeurs. On pourra consulter l'annexe 10.4 qui explicite la description synthétique ci-dessus.

Les courbes suivantes décrivent certains des résultats obtenus pour le problème 404.wcsp. La figure 7.1 montre l'existence d'un minimum du temps d'exécution⁵ en fonction du

⁴Un dump désignera ici une copie des sous-problèmes dans un fichier

⁵temps d'exécution chronométré ou *wall time* ou Elapsed time : c'est le temps chronométré entre le début du programme et sa terminaison.

nombre de sous-problèmes. La figure 7.2 est un zoom sur la figure précédente afin de préciser le valeur du minimum situé vers 250. La figure 7.3 exhibe un speedup maximum égal à 4.9 sur une machine à 8 cœurs logiques⁶, avec un nombre de sous-problèmes par cœur entre 28 et 33⁷.

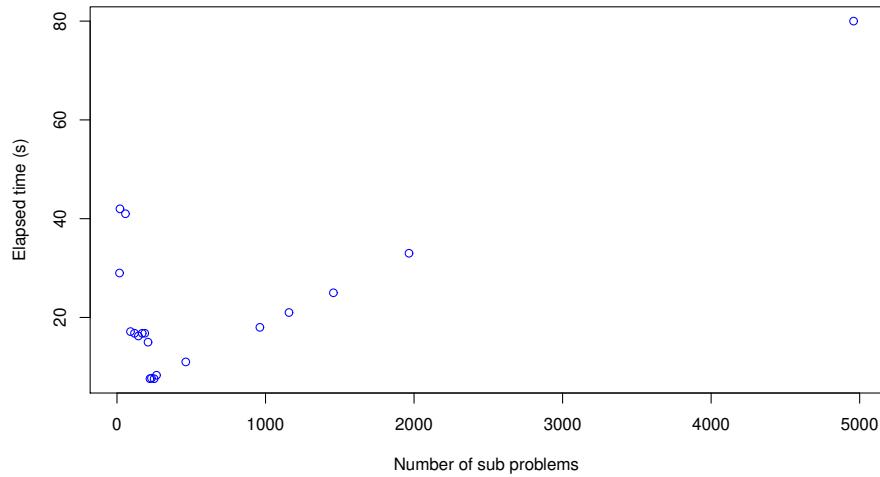


FIGURE 7.1 : Un temps minimum d'exécution obtenu pour une partition en 200 sous-problèmes.

⁶Quatre cœurs réels, 8 logiques avec l'hyperthreading

⁷On retrouve le résultat de Regin et al. [25]. La valeur *moyenne* de 30 problèmes/cœur est probablement architecture et problème dépendant. Ce résultat a été contredit sur les serveurs à 24 cœurs réels, 12 par sockets. L'utilisation de GNU parallel joue aussi un rôle. Les résultats ne sont pas forcément comparables.

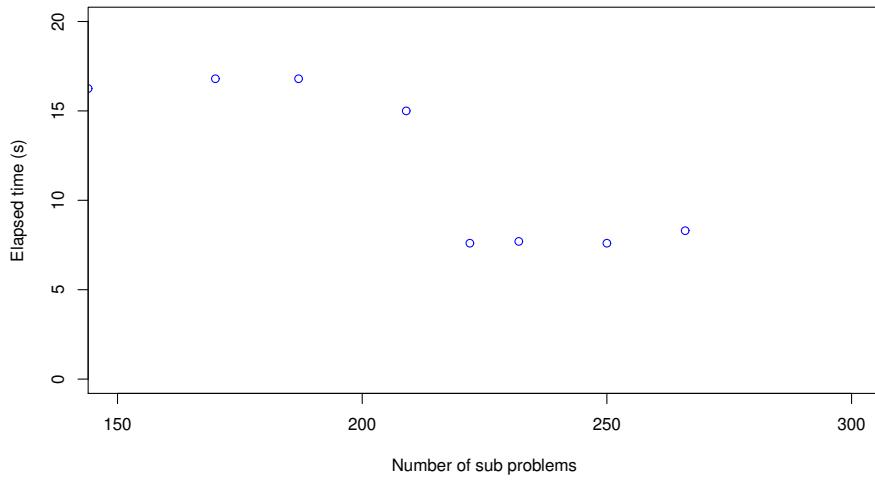


FIGURE 7.2 : Zoom de la figure 7.1 : un temps minimal obtenu entre 230 et 260 sous-problèmes.

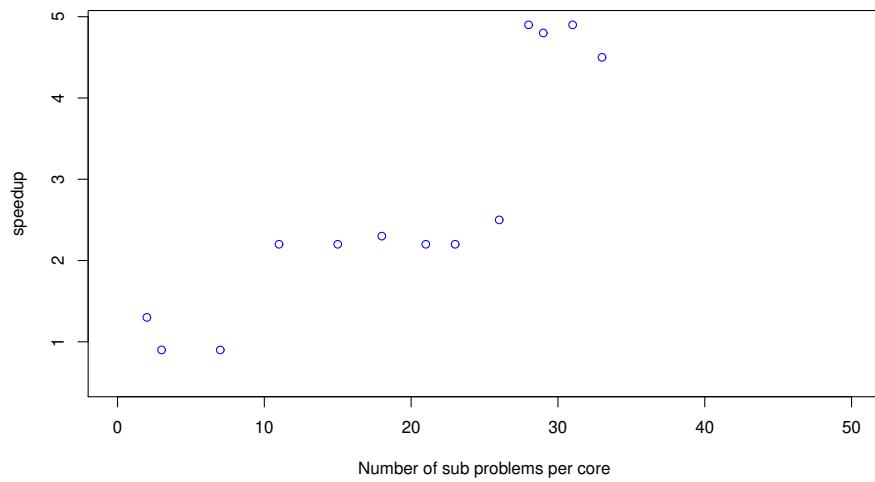


FIGURE 7.3 : Maximum speedup observé pour des valeurs entre 28 et 33 sous-problèmes par cœur.

7.5 Quelques expérimentations

7.5.1 Influence de la borne supérieure *cub*

On reprend ici la notation utilisée dans le code de toulbar2 : *cub* désigne la *Current Upper Bound* autrement dit la meilleure valeur courante de la fonction de coûts globale, c'est à dire le plus petit coût trouvée jusque là. En effet, comme en Economie, on cherche à minimiser un coût. Le *tuple complet* associé constitue la meilleure solution courante, ou *incumbent* solution, qui a permis de calculer ce *cub*.

Ainsi, lors de la génération des sous-problèmes, toulbar2 fournit également le *cub* obtenu au moment du *dump*. Le *dump* des nœuds désigne simplement l'opération d'écriture des sous-problèmes dans un fichier où chaque ligne correspond à un nœud représenté sous la forme d'une séquence de décisions.

résultats

On observe, comme prévu, une amélioration des résultats en utilisant le *cub* du *dump* en le passant aux instances de toulbar2 dans la phase parallèle via l'option -ub :

```
./toulbar2 404.wcsp -ub=cub -x=",44=0,12#0,32#0,13=0"
```

On parlera de passage d'information statique car effectué une seule fois en début de programme.

7.5.2 Problème du nombre de problèmes à générer

Le nombre N de sous-problèmes par cœur impacte de façon importante sur les performances. Ce nombre semble varier en fonction de la plateforme et du type de problèmes à résoudre et le trouver par tâtonnement expérimental n'est pas très pratique.

7.5.3 Problème de la distribution des temps de résolution

Les temps de résolution sont très disparates avec des problèmes résolus en une fraction de seconde et d'autres qui nécessitent près de 10s comme on le voit sur la figure 7.4. Comme le temps minimal de résolution est conditionné par le processus le plus lent, il serait intéressant de pouvoir générer des sous-problèmes de difficulté homogène. Pour ce faire, il faudrait pouvoir prédire la difficulté des sous problèmes générés à partir de leur complexité. S'il y a corrélation, un sous-problème détecter comme difficile pourrait être décomposé à nouveau. Cependant la complexité elle-même pose problème.

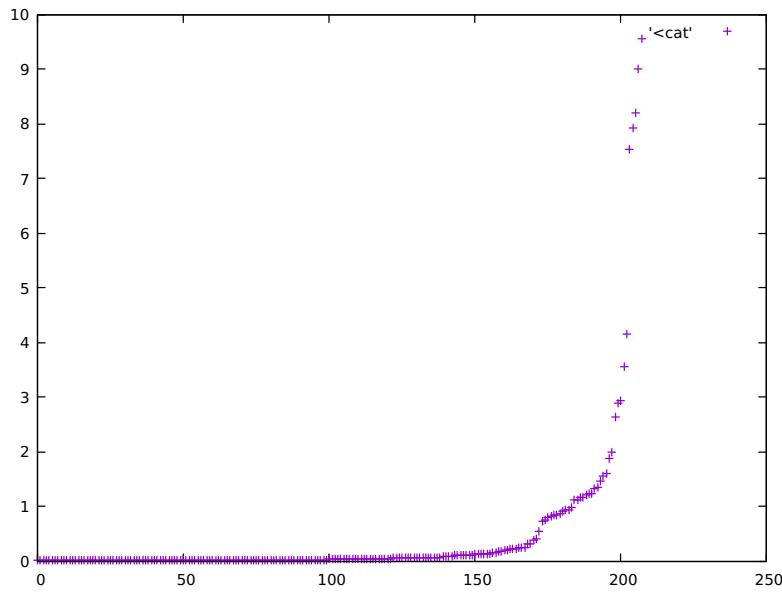


FIGURE 7.4 : Temps de résolution en secondes pour les 208 sous-problèmes du problème 404.wcsp classés par temps de résolution croissant. La grande majorité des processus terminent en une fraction de seconde. Les processus les plus gourmands terminent en un peu moins de 10s. Ici, le *cub* n'a pas été utilisé.

Distribution en complexité des sous-problèmes

Les complexités sont évaluées sous la forme du cardinal des produits cartésiens des domaines des variables non encore concernées par des décisions de type : =, #, <, >. Si les domaines sont réduits par propagation de contraintes au moment de l'extraction des sous-problèmes alors ces complexités seront sur-évaluées. A contrario, si les variables déjà concernées par un décision, le sont via une simple affectation par exemple, la complexité du sous-problème sera sous-évaluée et même proche de celle du problème global.

L'idée est donc de calculer un critère de complexité et de voir s'il est corrélé au temps de résolution en évitant de tirer des conclusions hâtives basées sur une pseudo complexité.

Les figure ci-dessous 7.5 montre une grande disparité des complexités théoriques avec une majorité de problèmes complexes de l'ordre de 10^{37} à 10^{38} . Ce qui est cohérent avec les temps de résolution très différents observés en figure 7.4.

404: Sub Problem Complexity

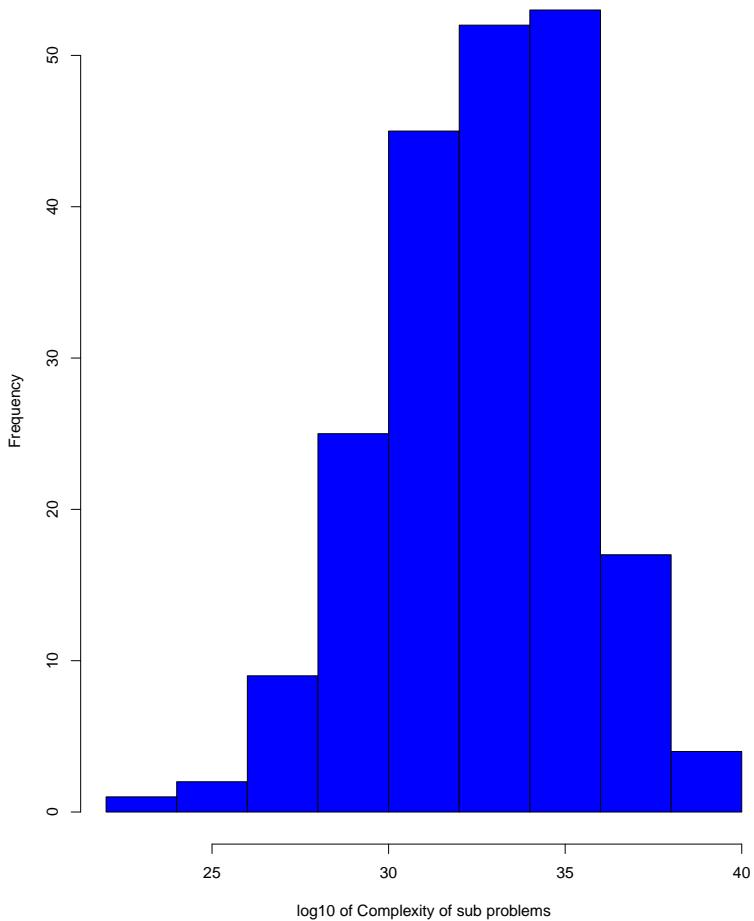


FIGURE 7.5 : Distibution du \log_{10} des critères de complexité. Le type *long double* a dû être utilisé pour éviter le dépassement de capacité des *long long int*. C'est pourquoi le critère est un flottant.

Le figure 7.6 ci-dessous indique que pour les complexités faibles, inférieures à 33 en échelle logarithmique le temps de résolution est faible. Par contre, au delà, les temps d'exécution des sous-problèmes s'allongent et la prévision des temps associés est hasardeuse.

On peut éventuellement inférer une application pratique pour la génération des sous-problèmes à savoir :

1. Calculer le critère de complexité théorique des sous-problèmes,
2. Remettre dans la file *open* ceux dont les critères dépassent un certain seuil pour les décomposer davantage.

Encore faut-il pouvoir déterminer ce seuil et trouver le bon compromis entre la durée de la phase séquentielle de génération des SP et celle de la phase parallèle. On risque également d'obtenir un nombre de sous-problèmes non optimal.

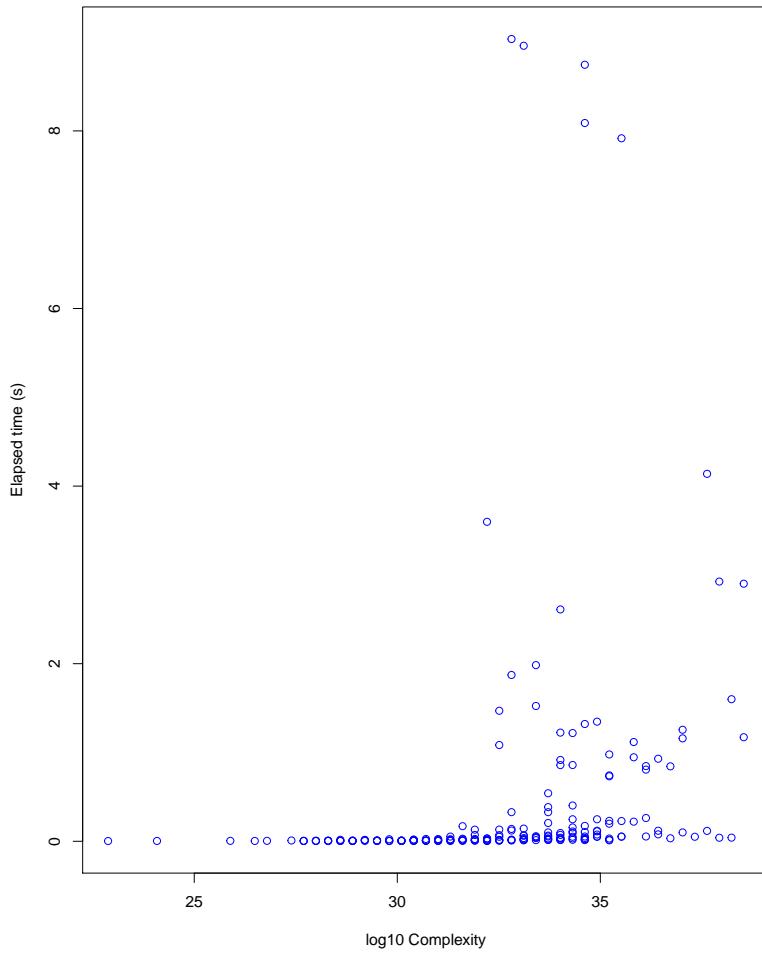


FIGURE 7.6 : Temps d'exécution en fonction du critère de complexity des sous-problèmes. En dessous du seuil 33, on peut inférer que le problème est facile mais au-dessus on ne peut rien conclure.

7.5.4 Performances globales de la parallélisation

Un échantillon de problème a été sélectionné et des mesures de speedup effectuées sur PC de bureau et sur serveur avec 24 cœurs. Le tableau 7.5.4 présente les résultats obtenus sur serveur.

wcsp file	Generation Time	Serial Time	Parallel time	speedup	Total Speedup	Number of sub problems
1PGB	2.1	4.3	5.1	0.83	0.59	52
graph11	1.7	325.0	1294.1	0.3	0.25	488
capmp1	61.2	266.0	101.9	2.61	1.63	172
scen06	0.6	945.0	437.0	2.16	2.16	175
capmo1	4.6	14.2	3.2	4.46	1.82	85
pedigree18	0.3	6.2	4.4	1.41	1.32	124
nug12	1.0	207.3	49.5	4.19	4.11	716
nug12	0.9	207.0	35.9	5.77	5.63	150
404.wcsp	0.4	36.7	20.9	1.8	1.73	699
404.wcsp	0.3	37.1	7.0	5.28	5.09	141

TABLE 7.1 : Ce tableau présente les temps de génération en secondes des sous problèmes sur un serveur 24 cœurs ainsi que les temps séquentiels et parallèles. Le speed-up total tient compte du temps de génération. Les problèmes nug12 et 404 montrent l'influence du nombre sous problèmes sur les speed-ups.

Sur les serveurs, les résultats se sont avérés décevants en choisissant un nombre de sous-problèmes cible N égal à $30k = 30 * 24 = 720$. La détermination du nombre optimal de sous problèmes nécessite des expérimentations nombreuses. La valeur $N = 150$ a donné des résultats plus intéressants ce qui correspond à environ 6 sous-problèmes par cœur.

Commentaires

- La scalabilité n'est pas bonne. On le voit en comparant les speed-ups obtenus sur PC (4.9s pour 4 cœurs réels) et sur serveur (5.09s pour 24 cœurs).
- Les problèmes avec un temps de génération important exhibent des speed-ups faibles.
- Le problème graph11 est résolu avec une borne inférieure locale lb calculée par Virtual Arc Consistency (VAC : option toulbar2 -A) plutôt que par l'heuristique par défaut : [Existential Directional Arc Consistency \(EDAC\)](#).
- Le problème pedigree est résolu avec les options -A -O=-3 -p=-8

7.6 Bilan

HBFS génère des sous-problèmes de bonne qualité avec en outre une borne supérieure cub améliorée au moment du *dump* utilisable pour booster la phase parallèle.

Une tentative de load balancing *a priori*⁸ qui consiste à produire des sous problèmes de complexité comparable n'a pas abouti. En outre, le load balancing dynamique est censé être géré par GNU parallel. Ainsi, si ces sous-problèmes sont très disparates en terme de temps de résolution, le vrai problème est qu'on ne connaît pas *a priori* le nombre optimal de SP à générer qui dépendent de la plateforme utilisée⁹ et du sous-problème lui-même comme le montre le tableau 7.5.4. On constate, en effet, un nombre de SP générés très variables pour

⁸Affecter N sous-problèmes de mêmes tailles à N processeurs assure le load balancing mais c'est une condition nécessaire mais non suffisante. En effet, le load balancing a pour objet de tenir occupés tous les workers durant l'exécution du programme. On peut donc avoir 10 processus de 1 secondes affectés à un cpu et un autre de 10s affecté à un autre cpu et sur cette machine à 2 processeurs le load balancing serait assuré.

⁹Constat expérimental entre deux architecture PC et serveur.

optimiser les performances mais aussi parce qu'on ne peut pas imposer un nombre précis de SP.

Enfin, creuser davantage la *piste EPS* aurait consisté à programmer la phase parallèle pour se passer de tout programme externe. Ceci aurait en fait conduit à implémenter le paradigme Master-Worker pour assurer le *load balancing*, la distribution du travail et le partage dynamique des connaissances entre processus. Il était donc temps de *backtracker*, d'élaguer l'arbre de recherche du stage et de repartir de la racine, d'autant plus que nous étions déjà le lundi 24 juin 2019.

Une approche plus dynamique devait donc être trouvée, avec une granularité plus fine et une maîtrise du code de bout en bout. Ce constat ne remet pas en cause l'EPS, qui a déjà été utilisée avec succès, mais dans d'autres contextes. L'EPS est un paradigme très adapté au calcul sur cluster ou sur grille de calcul.¹⁰.

De ce fait, le chapitre suivant portera sur l'étude et l'implémentation d'une autre méthode, le Master-Worker déjà cité, en espérant qu'il donne des résultats satisfaisants. La suite nous le dira.

¹⁰A contrario, des tests sur cluster tendent à montrer que l'algorithme HBFS parallélisé gagnerait davantage à fonctionner sur [supercomputer](#)



8. Paradigme Master-Worker

Résumé

L'implémentation du mécanisme de coordination Master-Worker utilise les bibliothèques openMPI et Boost.MPI. L'algorithme est d'abord détaillé puis des expérimentations sont exposées. Enfin la comparaison avec d'autres approches exhibe des résultats intéressants qui dépassent parfois les performances de solveurs perfectionnés tel que cplex qui activement maintenu par IBM.

8.1 Description du paradigme Master-Worker

Le paradigme Master-Worker est très connu et largement utilisé dans la communauté du calcul haute performance (HPC). Dans son implémentation la plus simple, un seul processus master coordonne le travail d'un ensemble de processus workers. Les workers n'échangent qu'avec le master et les échanges avec le master se limitent à recevoir le travail et à restituer les résultats, rien entre les deux. Comme on l'a déjà vu, un worker ne *parle* pas en travaillant et le master ne le dérange pas lorsqu'il est occupé. Le rôle principal du master est d'assurer que les processeurs, utilisés par les workers, soient occupés la plus grande partie du temps. De part sa conception, le master assure donc le *load balancing*. Mais il peut être en charge d'autres opérations.

Cependant, comme toutes les communications passent par le master, on peut prévoir un phénomène de goulot d'étranglement du fait d'une augmentation linéaire des échanges avec le nombre de workers. Ceci a pour conséquence de limiter l'échelonnabilité (*scalability*) du programme concrétisée par une *efficacité* décroissant avec le nombre de workers ; par construction le paradigme Master-Worker, implanté dans sa forme la plus simple, présentera un *speedup* par processeur décroissant. Cette limitation peut cependant être dépassée par exemple si le master demande aux workers d'envoyer directement le travail non terminé à ses *collègues*.

En outre, la granularité de niveau nœud du master-worker peut être modifiée dynamiquement pour passer à une granularité de niveau *sous-arbre* où le worker travaillera en toute indépendance tant que le master ne sera pas disponible. La centralisation de l'état de la recherche permet aussi de conserver dans une certaine mesure l'ordre de la recherche du programme séquentiel. On retrouve la nécessité de trouver un compromis entre échanges d'informations qui vont permettre d'éviter aux workers de faire un travail qu'ils n'auraient pas fait en séquentiel et l'overhead de communication décrit au chapitre 6 : Stratégies de parallélisation.

Malgré ses défauts potentiels et la simplicité du son principe, cette approche a été utilisée avec succès pour résoudre des instances difficiles non encore résolues de problèmes MILP¹, en utilisant le solveur CPLEX², de la même manière que les sous-problèmes étaient générés par toulbar2 dans le chapitre précédent, pour générer les sous-problèmes qui sont ensuite traités par une grille de calcul [24].

¹On appelle programme linéaire à variables mixtes, ou Mixed Integer Linear Programming, un programme linéaire contenant à la fois des variables continues donc dans \mathbb{R} et des variables discrètes donc dans \mathbb{Z} . Comme cette catégorie de problèmes généralise les programmes linéaires à variables entières, un MILP est un problème NP-difficile. Toutefois, dans de nombreux cas pratiques, ils sont faciles à résoudre à l'aide de solveurs entiers basés sur des B&B.

²CPLEX est un solveur commercialisé et maintenu par IBM. Son nom fait référence au langage C et à l'algorithme NP-hard du simplexe. Il est composé d'un exécutable et d'une bibliothèque de fonctions interfacées avec divers langages dont python, C, C++ et java : <https://www.ibm.com/analytics/cplex-optimizer>

8.2 Description de l'algorithme

8.2.1 Vue globale

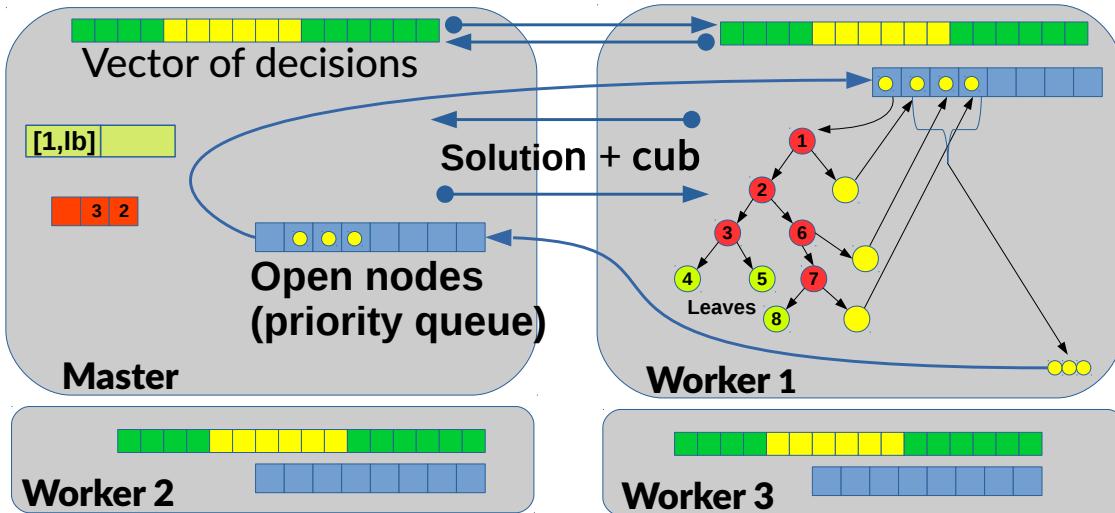


FIGURE 8.1 : HBFS Parallélisé : le master fournit le nœud 1 au worker qui lance un B&B dans le sous-arbre binaire de recherche associé à ce nœud. Les bornes inférieures sont propagées dans les nœuds enfants et servent à l’élagage de l’arbre. Le worker transmet au master les noeuds ouverts dès que le nombre maximum de backtracks, ici égal à 3, est atteint.

8.2.2 Algorithm Master

Algorithm 2: Parallel Hybrid Best-First Search : Master

```
/* INITIALISATIONS */  
1 cp := Ø ; /* cp = δ vecteur de choice points. */  
2 open := v(cp,lb = clb) ; /* initialisation du nœud racine */  
3 idleQ := {1,2,...} ; /* Au début, tous les workers sont inactifs */  
4 activeWork := Ø ; /* map avec les rangs des workers actifs et le lb min */  
*/  
/* Tant que clb < cub et qu'il reste du travail à distribuer ou en cours */  
/* TRAITEMENTS */  
5 while (clb < cub and (open ≠ Ø or activeWork ≠ Ø)) do  
  /* Tant qu'il reste du travail et des workers pour le faire */  
  6 while (open ≠ Ø and idleQ ≠ Ø) do  
    /* le master envoie un nœud, cub et la solution au worker */  
    7 isend(node,ub,masterSol) ; /* isend() non bloquant */  
    8 pop(idleQ) ; /* le worker devient actif */  
    9 activeWork[worker] := lb ; /* mémorise le lb du worker */  
   10 v := pop(open) ; /* on pop le nœud avec lb minimum */  
    /* Le master attend le message d'un worker quelconque */  
   11 recv(MPI_ANY_SOURCE,tag0,work2) ; /* recv() bloquant */  
    /* A la réception d'une réponse, le master se remets au travail */  
   12 cub := min(cub,cub(worker)) ; /* mets à jour le cub et la solution */  
   13 cp := cpWorker ; /* maj avec le vecteur de décisions du worker */  
   14 open := openWorker ; /* maj avec la file de priorité du worker */  
   15 activeWork.erase(worker) ; /* efface la paire [worker,lb] */  
    /* calculate the min of the lb among the active workers */  
   16 minLbWorkers = min(activeWork(worker));  
   17 idleQ.push(worker) ; /* le worker devient inactif */  
   18 clb := max(clb,min(lb(open),minLbWorkers)) ; /* calcule le clb global */  
   19 showGap(clb,cub) ; /* affiche le gap d'optimalité */  
  20 print(UB) ; /* affiche la valeur optimale si elle existe */  
  /* termine tous les processus workers */  
 21 return (clb,cub) ; /* retourne le gap au hbfs parallel */
```

8.3 Implémentation de l'algorithme

8.3.1 Bibliothèques utilisées

L'implémentation du Master-Worker a été effectuée en utilisant le standard MPI via son implémentation openMPI et l'interface C++ Boost associée : https://www.boost.org/doc/libs/1_71_0/doc/html/mpi.html.

8.3.2 Utilisation du hbfs parallèle

Il suffit d'ajouter l'option -para à la ligne de commande de toulbar2 :

```
./toulbar2 -para problem.wcsp
```

8.3.3 Description synoptique

Les deux fichiers principaux ci-dessous sont accessibles sur GitHub :

- <https://github.com/toulbar2/toulbar2/blob/kad2/src/search/tb2solver.cpp>
- <https://github.com/toulbar2/toulbar2/blob/kad2/src/search/tb2solver.hpp>

Dans le fichier.hpp, la class `Work` définit les messages échangés entre master et workers. Un constructeur permet de formater les messages envoyés par le master aux workers, un autre se charge des communications dans l'autre sens.

Dans le fichier.cpp, la méthode `pair<Cost, Cost> Solver::hybridSolvePara(Cost clb, Cost cub)` contient l'implémentation parallèle avec Boost.MPI de l'algorithme HBFS obtenu à partir de l'algorithme séquentiel de la méthode `pair<Cost, Cost> Solver::hybridSolve(Cluster *cluster, Cost clb, Cost cub)`.

Le master comme les workers envoient les messages en mode non-bloquant via la fonction `isend()`³ et les reçoivent en mode bloquant via `recv()`.

Le master comme les workers possèdent leur propre file *open*. Le master distribue les nœuds et la solution courante, y compris la valeur courante *cub*, aux workers qui se chargent d'effectuer le B&B avec parcours DFS borné par le nombre de backtracks (DFBB). Les workers renvoient les nœuds de leur propre file *open* au master qui les rangent dans la sienne.

Le master comprend deux boucles while imbriquées. La boucle externe assure la terminaison du programme ce qui n'est pas trivial dans le cas de la programmation parallèle. La boucle interne permet au master de distribuer le travail aux workers.

Pour des raisons d'optimisation spatiale un vecteur de *choice points*, ou vecteur de décisions, noté *cp*, est utilisé. Les nœuds en tant qu'objets possèdent comme attributs, outre le coût local *lb*, deux index qui pointent sur une partie du vecteur *cp*. La séquence de décisions associée au nœud n'est donc pas directement mémorisée dans le nœud, ce qui conduirait à des duplications de données. En effet, dans l'arbre de recherche, certains nœuds sont appelés à partager une partie de leur chemin.

³isend() correspond à un *immediate send* donc non bloquant. La fonction recv(), non préfixée par la lettre i, est bloquante ; le processus attend de recevoir un message pour poursuivre l'exécution du code.

8.3.4 Algorithme Worker

Algorithm 3: Parallel Hybrid Best-First Search : Worker

```
/* boucle infinie */  
1 while (1) do  
2   cpWorker := Ø ;    /* vecteur de choice points ou décisions du worker.  
   */  
3   openWorker := Ø ;      /* initialisation de la PQ du master */  
4   recv(master,tag0,work);  /* Le worker attend le travail du master */  
   /* A la réception d'un nœud, le worker se mets au travail */  
5   cub := min(cub,cub(master)) ;  /* mets à jour le cub et la solution */  
6   cpWorker := cp[first,last] ; /* maj avec les décisions associées au nœud  
   */  
7   openWorker.push(node) ;      /* maj sa file de priorité avec le nœud du  
   master */  
   /* le DFS B&B mets à jour openWorker avec les nœuds ouverts  
   produits */  
8   cub :=DFS(Av,cub,Z);  
9   NodesRecompute := NodesRecompute + v.depth ;  
10  clb := max(clb,lb(open)) ;      /* calcul du clb global courant */  
11  if (NodesRecompute > 0) then  
   /* heuristic Z adaptatif */  
12  if (NodesRecompute/Nodes > β and Z ≤ N) then Z := 2 × Z;  
13  else if (NodesRecompute/Nodes < α and Z ≥ 2) then Z := Z/2;  
   /* Le worker envoie les nœuds produits, son ub, la nouvelle  
   solution si elle est améliorante et son identifiant. isend() ->  
   envoi non bloquant */  
14  isend(cpWorker,openWorker,newWorkerUb,workerRank,workerSol)
```

8.4 Expérimentations

8.5 Comparaison entre EPS et Master-Worker

Le paradigme Master-Worker est meilleur dans tous les cas comme le montre la table 8.5.

Instance	<i>n</i>	<i>d</i>	S M-W	E M-W (%)	S EPS	E (%) EPS
graph11	340	44	2.22	9.24	0.25	1.05
capmp1	400	200	2.10	8.76	1.63	6.79
scen06	100	44	21.20	88.34	2.16	9.01
pedigree18	1184	5	11.18	51.03	1.32	5.50
nug12	12	12	20.14	83.92	5.63	23.46
404	100	4	16.25	67.70	5.09	21.21

TABLE 8.1 : Speed-up S et Efficacité E pour le Master-Worker(M-W) et l'*Embarrassingly Parallel Search* (ESP). Expérimentations faites sur serveur 24 cœurs. Le nombre de variables *n* et la taille max. des domaines *d* donnent une idée de la complexité du problème.

Comparaison de toulbar2 avec HBFS séquentiel et HBFS parallèle

Les expérimentations ont été réalisées sur des serveurs 24 cœurs (Intel Xeon E5-2680/87 at 2.50/3GHz and 256 GB) ou sur la platefome GenoToul (64-core nodes of Intel Xeon E5-2683 at 2.10GHz).

Les premiers résultats présentés dans le tableau 8.2 montrent des performances très variables en fonction du problème mais globalement bonnes et toujours meilleures que celles de toulbar2 séquentiel.

Quand ce n'est pas le cas, on peut invoquer une fraction séquentielle du programme non négligeable comme pour le problème capmp1 et invoquer le speed-up théorique de Amdahl pour interpréter les résultats faibles dus à la durée de pré-traitement et de chargement du programme.

Cependant, un autre aspect semble influer sur les performances, celui des temps d'attente du master et des workers. On observe des performances plus faibles en terme de speed-up si les temps d'attente du master comme des workers sont élevés. Ainsi, pour scen06 le temps d'attente des workers est de l'ordre de 0.1s tandis que le master attend les réponses des workers durant 37 secondes. Pour graph11, les workers attendent entre 7 et 25s et le master 122s pour un temps d'exécution de 125s. Le master est donc bien disponible. Les temps de communications ou les temps d'accès et d'écriture dans la mémoire pourraient être invoqués pour expliquer des speed-ups faibles.

Instance	Serial Time (s)	Parallel time (s)	speedup	Efficacité (%)	pre time
graph11	323.12	125.55	2.57	10.72	1.90
capmp1	173.27	82.23	2.11	8.78	6.80
scen06	1026.48	39.22	26.18	109.07	0.16
pedigree18	6.76	0.57	11.86	49.42	0.10
nug12	201.40	8.75	23.02	95.90	0.004
404.wcsp	34.66	1.30	26.66	111.09	0.005

TABLE 8.2 : Exécutions sur serveur 24 cœurs. Le temps CPU est utilisé pour le calcul des speed-ups. La colonne Parallel time indique le temps CPU total. La colonne **pre time** indique la durée du préprocessing.

Eléments nécessaires à la reproduction des résultats

- Le problème graph11 est exécuté avec l'option -A pour utiliser la méthode d'arc cohérence : [Virtual Arc Consistency \(VAC\)](#),

- Options toulbar2 pour le problème pedigree18 : -A -O=-3 -p=-8,
- Exécution en parallèle : mpirun -np 24 ./toulbar2 -para problem.wcsp,
- Exécution séquentielle : ./toulbar2 problem.wcsp,
- Exécution faite sur serveur 24 cœurs de l’unité MIAT : sullo et enfer avec tous les processeurs libres,
- Efficacité = $100 * (\text{speedup}/24)\%$.

Dans les Tables 8.3 and 8.4, on trouvera les temps en secondes nécessaires à la résolution et à la preuve d’optimalité d’instances de type Warehouse et Conception de protéines par ordinateur (CPD). Les tests ont été automatisés sur les serveurs 24 cœurs. Les exécutions sont arrêtées au bout de 1 heure. On notera que la virgule est le séparateur des milliers et non décimal.

On constate dans la table 8.3 une amélioration des performances dans presque tous les cas de figure pour le HBFS parallèle (PHBFS). Les speed-ups restent cependant faibles.

Instance	<i>n</i>	<i>d</i>	Time (sec.)		Speed-up
			HBFS	HBFS-24	
capmo1	200	100	10.92	5.14	2.12
capmo2	200	100	1.80	2.04	0.88
capmo3	200	100	6.09	3.73	1.63
capmo4	200	100	4.36	3.21	1.36
capmo5	200	100	2.69	2.58	1.04
capmp1	400	200	172.57	80.64	2.14
capmp2	400	200	95.15	61.35	1.55
capmp3	400	200	75.04	56.25	1.33
capmp4	400	200	107.86	78.4	1.38
capmp5	400	200	81.15	52.9	1.53
capmq1	600	300	679.43	412.52	1.65
capmq2	600	300	841.80	503.64	1.67
capmq3	600	300	647.67	431.47	1.5
capmq4	600	300	1,093.55	514.42	2.13
capmq5	600	300	1,388.41	701.61	1.98

TABLE 8.3 : Benchmark warehouse [16] où *n* désigne le nombre de variables du problème et *d* la taille du plus grand domaine de ces variables.

Dans la table 8.4, on voit que PHBFS a résolu 3 instances non résolues par HBFS.

Instance	<i>n</i>	<i>d</i>	Time (sec.)		Speed-up
			HBFS	HBFS-24	
1xaw	107	412	721.43	568.50	1.27
3lf9	120	416	407.28	407.92	1
5dbl	130	384	122.84	171.82	0.71
5e10	133	400	147.73	198.23	0.75
5e0z	136	420	148.26	193.41	0.77
5eqz	138	434	3,366.11	1,049.11	3.21
1dvo	152	389	622.03	463.29	1.34
4bxp	170	439	327.46	395.16	0.83
1is1	185	431	-	2,545.82	-
2gee	188	397	797.22	863.64	0.92
5jdd	263	406	-	2,758.98	-
3r8q	271	418	1,605.30	1,294.97	1.24
1f00	282	430	-	2,140.40	-

TABLE 8.4 : Benchmarks *Computational Protein Design* (CPD) [22]. Les tirets indiquent que le problème n'a pu être résolu en moins d'une heure.

8.5.1 Comparaison avec des travaux similaires

Des expérimentations sur d'autres instances difficiles de conception de protéines ont montré des speed-up intéressants.

Dans la table 8.5, on compare les solveurs cplex, daoopt et HBFS sur le benchmark *Linkage*. Les résultats du solveur daoopt sont obtenus sur un cluster de dual core 2.67 GHz Intel Xeon X5650 6-core CPUs et 24 GB de RAM. Ici, la meilleure scalabilité de PHBFS permet d'obtenir de meilleurs résultats sur cluster sur les instances pedigree31 et 44 mais PHBFS est dominé par cplex sur les instances pedigree19 et 51.

Le solveur daoopt est loin derrière mais avec des speed-ups et efficacités élevés.

	pedigree19	pedigree31	pedigree44	pedigree51
<i>n</i>	793	1,183	811	1,152
<i>d</i>	5	5	4	5
cplex	790	59.30	6.35	36.23
//10	191 (4.14)	9.00 (6.59)	2.48 (2.56)	9.43 (3.84)
//30	75 (10.53)	7.17 (8.27)	2.69 (2.36)	5.34 (6.78)
daoopt	375,110	16,238	95,830	101,788
//20	27,281 (13.75)	1,055 (15.39)	6,739 (14.22)	6,406 (15.89)
//100	7,492 (50.07)	201 (80.79)	1,799 (53.27)	1,578 (64.50)
HBFS	3,126	4.34	39.72	1,608
//10	434.27 (7.20)	1.51 (2.87)	6.08 (6.53)	179.22 (8.97)
//20	227.02 (13.77)	1.39 (3.12)	3.18 (12.49)	72.30 (22.24)
//100	119.43 (26.17)	0.97 (4.47)	1.64 (24.22)	31.40 (51.21)

TABLE 8.5 : Benchmark *linkage* [8] avec divers nombre de coeurs. Entre parenthèses, sont indiqués les speed-ups.

On voit dans la table 8.6 que HBFS parvient à résoudre toutes les instances contrairement à CPLEX et que PHBFS, utilisé avec 10 coeurs, améliore encore les résultats.

Instance	n	d	cplex	cplex-10	HBFS	HBFS-10	Speed-up
1UBI	13	198	-	-	1,023	214.02	4.78
2DHC	14	198	-	-	8.2	5.83	1.41
2DRI	37	186	-	-	135.5	30.00	4.52
1CDL	40	186	-	-	392.6	54.95	7.14
1CM1	42	186	-	6,177	6.6	6.11	1.08
1BRS	44	194	-	-	555.3	107.86	5.15
1GVP	52	182	-	-	596.1	185.75	3.21
1RIS	56	182	-	-	129.7	36.23	3.58
3CHY	74	66	-	5,259	88.7	20.71	4.28

TABLE 8.6 : Autre benchmark CPD [2]. Un tiret indique que la méthode n'est pas parvenue à prouver l'optimalité en moins de 9000s.

8.6 Bilan

La version Master-Worker de PHBFS produit des résultats intéressants qui améliorent la plupart du temps les performances du HBFS séquentiel mais surtout dépassent dans certains cas des solveurs perfectionnés tel que cplex⁴ ou daoopt⁵.

⁴Solveur activement développé par IBM : <https://www.ibm.com/analytics/cplex-optimizer>

⁵<https://github.com/lotten/daoopt>



9. Conclusion

L'objectif de ce stage, qui s'est déroulé du 18 mars au 15 septembre 2019 à l'INRA Toulouse, a consisté à paralléliser HBFS un algorithme séquentiel de recherche arborescente implémenté en C++ dans [toulbar2](#), un solveur de réseaux de fonctions de coûts représentables sous la forme d'hypergraphes.

Après une première étape bibliographique (1.5 mois), de prise en main des outils et de [toulbar2](#) (0.5 mois), la piste EPS a été explorée durant 1 mois comprenant l'analyse du code, sa modification pour produire les sous-problèmes et les expérimentations et tests associés. La piste Master-Worker a pris environ 2 mois comprenant l'analyse du code, le choix de solutions techniques, les tests, les expérimentations et comparaisons avec d'autres approches utilisées dans des solveurs tels que [cplex](#). La rédaction de ce rapport et d'autres documents a nécessité 1 mois.

La démarche utilisée a consisté à explorer des pistes de parallélisation, à en choisir deux : la piste EPS et la piste Master-Worker et à les explorer c'est à dire à réaliser les développements, les tests, les expérimentations, la comparaison et l'analyse des résultats.

HBFS a finalement été parallélisé avec succès en donnant des résultats probants qui feront l'objet d'une présentation à la 25ème conférence CP 2019¹ sur la programmation par contraintes.

Des améliorations pourraient cependant être apportées :

- En terme d'échelonnabilité avec l'aide d'autres paradigmes déjà cités dans ce rapport² : *Superviseur-Worker, Multiple-Masters-Worker, Master-Hub-Worker*
- En terme de speedup, en parallélisant les pré-traitements si toutefois c'est possible,
- En parallélisant l'algorithme : Backtracking Tree Decomposition HBFS (BTD-HBFS).

¹<https://cp2019.a4cp.org/>

²Le présent rapport est publié sur <https://github.com/kad15/SandBoxToulbar2/blob/master/kad/rapport.pdf>



10. Annexes

10.1 Annexe A : Génération a priori des sous problèmes

Le code ci-après constitue une implémentation en C++ qui, notamment, calcule les ensembles issus de produits cartésiens d'un ensemble de variables x_i de domaine D_i , utilisée dans la production de sous problèmes a priori.

```
/** /brief generate subproblems such that Ak-1 < q <= Ak
*/
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <vector>
#include <cassert>
#include <algorithm>
using namespace std;

/** \brief function that read a wcsp file and return
a vector of long containing the domain size of variables

*/
vector<long> readDomains(const string & fic)
{
    ifstream file(fic,ios::in);
    string line;
    if(file)
    {
        getline(file, line);
        getline(file, line); //get second line with domains of vars
    }
    else
    {
        cout << "ERREUR: no file" << endl;
    }

    // Vector of string to save tokens
    vector <long> domain;

    // stringstream class check1
    stringstream check1(line);
    string token;

    // Tokenizing w.r.t. space ' '
    while(getline(check1, token, ' '))
        domain.push_back(stoi(token));

    return domain;
}

/** \brief function that compute the cardinal of cartesian product of the domains
between two indexes min and max, max excluded
*/
long cardinalProd(const vector<long> & dom, const long min, const long max)
{
    assert(min <= max);
    assert(max < (long) dom.size());
    long prod = 1;
    for(long i = min; i<max; i++)
        prod *= dom[i];

    return prod;
}

/** \brief compute the complexity of a sub problem(sp)
which is the cartesian product of all the variables
divided by the cardinal of cartesian prod of the assigned vars of the sp
*/
long subPbComplexity(const vector<long> & subDomain, const long globaleCpx )
{
    return globaleCpx / cardinalProd(subDomain, 0, (long) subDomain.size());
}
```

```

}

/** \brief Compute the number of vars k in assignment Ak such that
the provide number q is in (card(Ak-1), card(Ak)] */

long sizeOfAk(const vector<long> & dom, const int q)
{
    long k = 0;
    long AkMinusOne = dom[0];
    assert(q>=dom[0]);
    if(q==dom[0]) return 1;
    long Ak = AkMinusOne * dom[1];
    if(q>AkMinusOne && q<= Ak) return 2;
    for(long i=2 ; i< (long) dom.size(); i++)
    {
        AkMinusOne = Ak;
        Ak *= dom[i];
        // cout << "AkMinusOne = "<< AkMinusOne<<endl;
        // cout << "Ak = "<< Ak<<endl;

        if(q==AkMinusOne)
        {
            k=i;
            break;
        }
        if(q>AkMinusOne && q<= Ak)
        {
            k=i+1;
            break;
        }
    }
    //cout << "nb of subproblems to produce = " << cartesianProd(dom,0,k)<<endl;
    cout << "nb of subproblems generated = " << Ak<<endl;
    return k;
}

ostream& operator<<( ostream &flux, const vector<vector<long>> &vv)
{
    for(size_t i = 0; i < vv.size(); i++) {
        for (size_t j = 0; j < vv[i].size(); j++) {
            cout << vv[i][j] << " ";
        }
        cout << std::endl;
    }
    return flux;
}

ostream& operator<<( ostream &flux, const vector<long> &v)
{
    for(size_t i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << std::endl;
    return flux;
}

/** \brief compute the cartesian product of a vector of vectors
domains : set of domains dm represented by c++ vectors
*/
vector<vector<long>> cartProd (const vector<vector<long>>& domains)
{
    vector<vector<long>> vv = {{}}, tmp; // init vector of vectors
    for (const vector<long> & dm : domains) { // for each domain dm in the set of domains dm
        tmp= {};
        for (const vector<long> & v : vv) { // for each sub vector v in vv (v is {} at first)
            for (const long value : dm) { // for each value in domain dm

                tmp.push_back(v); // add domain v in tmp here v ={} at first
                // cout << "x ="<<x << endl;
                tmp.back().push_back(value); // add value to the last domain in tmp, here v={} tmp
                ={{0},{1}} ...
            }
        }
    }
}

```

```

        }
    }

    vv.swap(tmp); // vv become tmp and tmp become vv
}
//cout<< tmp << endl;
return vv;
}

vector<vector<long>> convertVV(vector<long> dom, int k)
{
    // convert vector dom in a vector of vectors
    vector<vector<long>> vv;
    vector<long> v;
    for (long i =0; i<k; i++ )
    {
        for ( long j =0; j<dom[i]; j++)
        {
            v.push_back(j);
        }
        vv.push_back(v);
        v.clear();
    }
    // cout<< vv << endl;
    return vv;
}

void writeJobs(vector<vector<long>> cp, const string & fic)
{
    ofstream file(fic);
    if(file) // if ok
    {
        int k;
        for(size_t i = 0; i < cp.size(); i++)
        {
            k=0;
            file << "-x=\\"";
            for (size_t j = 0; j < cp[i].size(); j++)
            {
                file << ","<< k<<"="<< cp[i][j] ;
                k++;
            }
            file << "\\"<<endl;
        }
    }
    else
    {
        cout << "File error "<< fic << endl;
    }
    file.close();
}

int main(int argc, char *argv[])
{
    if(argc !=4)
    {
        cout << "usage: "<< argv[0] << " " << "name of input wcsp file, space, name of output
subproblems file,space, approximate number of sub problem"<< endl;
        cout << "Example: "<< argv[0]<< " 404.wcsp job_file 250"<< endl;
        cout << "Parallel command : cat job_file | parallel -j+0 --eta -k ./toulbar2 404.wcsp
    }
}
```

```
{}"<<endl;
}

vector<long> dom = readDomains(argv[1]); // domain of vars

long k = sizeOfAk(dom, atoi(argv[3])); // nb of vars to try to match q variables
cout << "nb of corresponding pre-assigned variables = "<< k << endl;
vector<vector<long>> vv = convertVV(dom, k);

vector<vector<long>> cp = cartProd(vv);
writeJobs(cp, argv[2]);

return 0;
}
```

10.2 Annexe B : Utilisation de toulbar2 avec Eclipse

On trouvera ci-dessous une proposition de mode opératoire dont l'objectif est d'obtenir rapidement un environnement de développement efficace.

HowTo.use.toulbar2_with_eclipse.cpp

```
1# TUTORIAL VERSION 3 : juillet 2019
2# AUTHOR : KAD
3
4PREREQUISITES:
5sudo apt-get update
6
7 - Connecting to git without login :
8create an ssh key without pass phrase.
9cd ~/.ssh && ssh-keygen -t ed25519 -C "toto@free.fr"
10cat id_ed25519.pub, copy the output, use it to create a new ssh key on github. see
    Settings of your account.
11git clone git@github.com:toulbar2/toulbar2.git
12
13 - sudo apt install cmake cmake-data cmake-curses-gui libboost-all-dev libboost-graph-
    dev zlib1g-dev liblzma-dev libjemalloc-dev libgmp3-dev htop sudo libomp-dev openmpi*
14
15 - sudo apt install texstudio doxygen graphviz texlive-latex-recommended texlive-
    fonts-recommended
16
17 - Install last version Eclipse IDE for Scientific Computing which support mpi,
    openmpi; Parallel Tools Platform (PTP)
18 for instance via https://www.eclipse.org/downloads/packages/
19
20 1 - SHELL PHASE
21
22 In toulbar2 folder create a build folder.
23 cd build
24 Do cmake .. to configure the build ( c to configure, g to generate, q to quit)
25 nb : do sudo apt-get install libboost-all-dev to install boost_mpi
    boost_serialize ...
26
27 Do cmake .. to create the Makefile ( cmake .. has to be done each time new files are
    added to the project)
28 Do make -j8 where 8 is the number of cores on your computer
29
30 nb : re-execute cmake .. in build directory to update the Makefile
31 if you add files in the code of toulbar2
32
33 nb : cmake .. means execute cmake (cmake configure cmake tool which in turn
    produce a makefile)
34 with the CMakeLists.txt file in the parent directory.
35
36 nb : build folder can be named as you wish : for instance you can
37 create 2 folders : one folder named "debug" for the debug version,
38 the other named "release" for the release version.
39
40 IMPORTANT IF YOU CAN UPDATE GIT :
41 put the following lines and others in .gitignore to avoid polluting the repo with
    build folder and eclipse project files.
42 release/
43 debug/
44 *.cproject
45 *.project
46
47 prerequisites : two following "parallel" folders :
48 1 - toulbar2/src
49 2 - toulbar2/release build folder has to be created if it does not exist
50
51
```

HowTo.use.toulbar2_with_eclipse.cpp

```
52 - cd build (where build designate debug or release folder or whatever)
53
54 - Configure cmake with ccmake tool: ccmake .. (where the 2 point designate the parent
      directory namely toulbar2)
55 select the option you need e.g. replace Release by Debug, etc ...
56 type c to configure cmake then g to generate conf files.
57
58 - Run cmake .. (cmake with only one c this time).
59 this will generate make file and other stuff.
60
61 - make -j8 to compile on an 8 cores machine which produce the exe file
62 toulbar2/debug/bin/Linux/toulbar2
63
64
65
66 2 - ECLIPSE PHASE
67
68 Eclipse does not seem to deal properly with existing cmake project: plugin are more
      or less obsolete, ...
69 so we import a makefile project to use make -j8 through eclipse.
70 Open eclipse and create a workspace named e.g. toulbar2_eclipse_workspace for
      instance in your home directory.
71 note that toulbar2_eclipse_workspace and toulbar2 are distinct folders. Eclipse uses
      the former one to save its metadata,
72 the latter is used by eclipse through .project and .cproject files to write the
      workbench parameters, etc
73 the workbench designate the actual "workspace" where you write your code.
74
75 - Create the toulbar2 eclipse project :
76 File -> Import -> C/C++ -> Existing code as Makefile project
77 nb : the version used here is eclipse scientific computing 2019-06 (eclipse cdt with
      parallel tools)
78
79 - enter a name for the project : toulbar2_debug for instance
80 - enter the path to toulbar2 folder
81 - choose linux GCC as a toolchain
82 - validate with finish button
83
84=> the project is created in eclipse
85
86 - select project folder on the upper left and go to menu Project -> properties -> C/
      C++ build
87 uncheck use default build command and add type make -j8 ( j = jobs : where 8 is
      your number of processors, more than 8 is also possible with only 8 cores)
88 this will trigger a faster parallel compilation. (alternatively, go in Behavior tab
      and choose enable parallel build)
89 add release to build directory i.e. ${workspace_loc:/toulbar2_release}/release
90 to tell eclipse where to build toulbar2.
91 clic Manage configurations > rename Default with release then create
92 a debug configuration if you have compiled and created a debug folder too.
93 then in eclipse Projet menu you will be able to set the default configuration.
94
95 toulbar2 executable will be created in release/bin/Linux directory.
96
97 To use the toulbar2 you just build, and not a possibly other version installed in
      your system,
98 go to release/bin/Linux directory and use ./toulbar2 file.wcsp put your wcsp files
      in this current folder.
99
```

HowTo.use.toulbar2_with_eclipse.cpp

```
100
101- in C/C++ general -> code analysis check use project settings and uncheck all to
      avoid false warning and errors
102
103- in formatter choose a format for your code that you like. Normally toulbar2 uses
      K&R format style
104
105- to have the code formatted when saving the file : windows -> Preferences -> C/C++ -
      > editor -> Saves Actions -> select Format source Code -> format for all lines
106- mannually format the code with ctrl+shift+F
107
108- To change font in editor : Windows -> Preferences -> General -> Appearance ->
      Color and font -> C/C++ -> C/C++ editor text font -> button Edit
109
110
111- To execute toulbar2 from eclipse with arguments : Menu Run -> Run configuration ->
      C/C++ Application -> toulbar2 release default
112 add toulbar2/build/bin/Linux/toulbar2 the path to the program toulbar2
113 then in tab Argument add the argument to use with toulbar2 e.g. a wcsp file name and
      other toulbar2 options.
114 Choose your working folder e.g. ${workspace_loc:toulbar2}/release/bin/Linux so that
      to avoid to place wcsp files in toulbar2 folder.
115 You can also run a parallel run configuration.
116
117
118
119
120
121
```

10.3 Annexe C : Compilation sur le cluster genotoul

On trouvera ci-dessous un fichier tcl (Tool Command Language) utilisé par l'utilitaire "module" et un fichier cmake modifié pour compiler toulbar2 sur le cluster genotoul. Pour compiler toulbar2, on peut utiliser le fichier *.tcl pour configurer l'environnement puis suivre la "SHELL PHASE" du mode opératoire en annexe [10.2](#).

```
module load -f fichier.tcl
```

Le fichier cmake doit être renommé en CMakeList.txt et placé dans le dossier toulbar2. Il faut alors créer un répertoire de build : release dans le dossier toulbar2, aller dans release et lancer le configurateur cmake via la commande : ccmake .. ce qui permet à ccmake d'accéder au fichier CMakeList.txt situé dans le répertoire parent toulbar2.

```
#%Module1.0#####
# description : module to load environment on SLURM cluster
# kad version 1.0
# usage: module load -f path_to/my_own_module
# To unload one module: module unload bioinfo/bowtie2-2.2.9
# To unload all module and specific variable: module purge

#limit coredumpsizer 0
#module purge
module load compiler/gcc-7.2.0
module load mpi/openmpi-2.1.2
module load compiler/cmake-3.12.3
#module load compiler/intel-2018.0.128

setenv BOOST_ROOT /tools/libraries/Boost/boost_1_70_0_openmpi-2.1.2
setenv BOOST_INCLUDEDIR /tools/libraries/Boost/boost_1_70_0_openmpi-2.1.2/include
setenv BOOST_LIBRARYDIR /tools/libraries/Boost/boost_1_70_0_openmpi-2.1.2/lib

setenv MPI_INCLUDE_PATH /tools/cluster/mpi/openmpi/2.1.2/gcc-4.5.8/include
setenv JEMALLOC_ROOT /home/allouche/work/kad/jemalloc

prepend-path PATH /tools/cluster/mpi/openmpi/2.1.2/gcc-4.5.8/bin
prepend-path LD_LIBRARY_PATH /tools/libraries/Boost/boost_1_70_0_openmpi-2.1.2/lib
prepend-path LD_LIBRARY_PATH /home/allouche/work/kad/jemalloc/lib
```

```
# modif kad aout 2019 : pour compilation sous cluster
# avec boost mpi 1.70 et jemalloc recompilée en local
# jemalloc installée dans /home/allouche/work/kad/jemalloc
# pour compiler toulbar2 sur genologin faire :
# cd ....../toulbar2
# module purge
# module load -f ./toulbar2_module.tcl
# cd vers le répertoire de build désiré
# ccmake .. vérifier les options touche t pour avoir tout
# c pou config puis g pour generate
# cmake ..
# make -j8
# COPIER UN PB DANS BIN LINUX OU SE TROUVE le binaire toulbar2 compilé
# cd bin/Linux pour tester srun ./toulbar2 404.wcsp
# srun -N 3 --ntasks-per-node=32 -n 96 --time=00:04 --exclusive=user ./toulbar2
404.wcsp -para

#####
# cmake file for toulbar2 framework building , test and packaging
# version 0.9
# David allouche 17/10/10
#####

cmake_minimum_required(VERSION 2.6)
set (CMAKE_EXPORT_COMPILE_COMMANDS ON)
set (My_cmake_script "${CMAKE_CURRENT_SOURCE_DIR}/cmake-script") # location of
cmake script needed
set (My_Source src)      # source location
set (doc_destination "share/doc")           #path of doc installation
set (My_misc_source misc/src/)
list(APPEND CMAKE_MODULE_PATH "${My_cmake_script}/Modules")

set (MAINTAINER "David Allouche <david.allouche@inra.fr>") #used in packaging
set (CONTACT "https://github.com/toulbar2/toulbar2") #used in packaging

#####
# PROJECT NAME
#####

project("toulbar2")

MESSAGE(STATUS "#####")
MESSAGE(STATUS "project :${PROJECT_NAME} toolkit compilation cmake file version
0.99")
MESSAGE(STATUS "source:${CMAKE_CURRENT_SOURCE_DIR} ")
MESSAGE(STATUS "MAKE_BUILD_TYPE : ${CMAKE_BUILD_TYPE}")
MESSAGE(STATUS "#####")

include(CMakeDependentOption)

#####
# cmake option definition
#####

#      OPTION(TOULBAR2 "toulbar2 solver compilation [default: on]" ON)
#      OPTION(MENDELSOFT "mendelsoft compilation and packaging option
[default:off]" OFF)
OPTION(WIN32 "toulbar2 cross compilation flag [default: off]" OFF)
OPTION(ALL_APP "dedicated applications and toulbar2 solver compilation [default:
off]" OFF)
OPTION(MENDELSOFT_ONLY "mendelsoft compilation and packaging ONLY ==> remove other
exe and lib compilation [default:off]" OFF)
OPTION(TOULBAR2_ONLY "toulbar2 solver compilation ONLY [default: on]" ON)
OPTION(LIBTB2 "lib toulbar2 compilation [default: off]" OFF)
OPTION(ILOG "ilog solver binding [default: off]" OFF)
#      OPTION(LIBTB2INT " lib toulbar2 compilation INT mode required with ilog
```

```

and windows [default: off]" OFF)
OPTION(XML "add a reader for the (W)CSP xml input format [default: off]" OFF)
##      OPTION(CPLEX "encode global cost functions into a linear program solved by
CPLEX [default: off]" OFF)
OPTION(Boost "boost graph binding [default: on]" ON)
OPTION(MPI "MPI [default: off]" OFF)
OPTION(HBFS_MPI "HBFS parallelization using MPI [default: on]" ON)
OPTION(WIDE_STRING "use wide string to encode long domains in n-ary cost
functions/separators [default:on]" ON)
OPTION(LONG_COSTS "use long long to encode costs [default:on]" ON)
OPTION(LONG_PROBABILITY "use long double to encode probabilities [default:on]" ON)
OPTION(BUILD_API_DOC "build and install HTML documentation with doxygen
[default:off]" OFF)
OPTION(BUILD_API_DOC_LATEX "build and install LaTeX PDF documentation with doxygen
[default:off]" OFF)
OPTION(STATIC "static compilation flag [default: off]" OFF)
OPTION(verbose "verbose mode [default:on]" ON)
OPTION(COVER_TEST "cover test [default:on]" ON)
OPTION(BENCH "benchmarking [default:off]" OFF)
OPTION(WITH_MEM_JEMALLOC "Enable malloc replacement (http://www.canonware.com/jemalloc) [default on]" ON)
mark_as_advanced(WITH_MEM_JEMALLOC BUILD_API_DOC_LATEX)

#####
#Default profile for compilation
#####
IF(NOT CMAKE_BUILD_TYPE)
    SET(CMAKE_BUILD_TYPE Release CACHE STRING
        "Choose the build type, options are: None Debug Release RelWithDebInfo
MinSizeRel."
        FORCE)
ENDIF(NOT CMAKE_BUILD_TYPE)

SET(CMAKE_CXX_FLAGS "-Wall -std=c++17" )

IF(CPLEX)
    SET(CPLEX_LOCATION "/opt/ibm/ILOG/CPLEX_Studio126")
    SET(CPLEXFLAGS "-fPIC -fexceptions -fno-strict-aliasing -DIL0GCPLEX -DIL_STD -
DIL0STRICTPOD -pthread -I${CPLEX_LOCATION}/cplex/include -I${CPLEX_LOCATION}/
concert/include")
    SET(CPLEXLIB "-L${CPLEX_LOCATION}/cplex/lib/x86-64_linux/static_pic -L${
CPLEX_LOCATION}/concert/lib/x86-64_linux/static_pic")
    SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${CPLEXFLAGS}")
    SET(CMAKE_EXE_LINKER_FLAGS " ${CPLEXLIB}")
    SET(all_depends ${all_depends} "ilocplex")
    SET(all_depends ${all_depends} "cplex")
    SET(all_depends ${all_depends} "concert")
    SET(all_depends ${all_depends} "m")
    SET(all_depends ${all_depends} "pthread")
    SET(STATIC off)
ENDIF(CPLEX)

#####
# Compile Options on recent 64 bytes Macs.
# thanks to alex rudnick
#####
IF(APPLE)
    SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -m64" )
    SET(CMAKE_SHARED_LINKER_FLAGS "-Wall -m64" )
    SET(STATIC off)
ENDIF(APPLE)

#####
# default OPTION for toolbar2 test phase
#####
SET(FOPT "test-opt.cmake" CACHE string "filename containing local options used for

```

```
validation")
SET(Default_test_option CACHE STRING "Define toulbar2 option used in command line
for testing: cf toulbar2 command line")
SET(Default_test_timeout 300 CACHE INTEGER "default test timeout")
SET(Default_Nb_cpu 4 CACHE INTEGER "default ncpus number for mpirun ")
SET(Default_validation_dir "validation" CACHE string "default location of
validation files")
SET(Default_cover_dir "cover" CACHE string "default location of cover test files")
SET(Default_regex "end." CACHE string "default regex searched in test output")

SET(Default_BenchDir "benchmarks" CACHE string "default location of benchmark
files")
SET(Default_BenchFormat "wcsp" CACHE string "default format extension for
benchmark files search")
SET(Default_bench_timeout 30 CACHE INTEGER "default timeout used for benchmarking
")
SET(Default_bench_option "TOULBAR2_OPTION" CACHE STRING " option used in command
line for benchmarking.")
SET(Default_bench_regex "test ok" CACHE string "default regex searched in bench
output")

MARK_AS_ADVANCED(FORCE LIBTB2 verbose Boost FOPT CMAKE_INSTALL_PREFIX
Default_test_timeout WIDE_STRING LONG_PROBABILITY LONG_COSTS)
MARK_AS_ADVANCED(FORCE BUILD_SHARED_LIBS GMP_LIBRARY)

IF (MPI)
    SET(Boost ON)
    SET(STATIC OFF)
ENDIF(MPI)

#####
# static building
#####

IF(STATIC)
    MESSAGE(STATUS "STATIC COMPILATION ON (warning: jemalloc, xml, and MPI options
not compliant with static link)")
    SET(WITH_MEM_JEMALLOC OFF)
    SET(XML OFF)
    SET(MPI OFF)
    SET(BUILD_SHARED_LIBS OFF)
    set(CMAKE_FIND_LIBRARY_SUFFIXES ".a")
    set(CMAKE_EXE_LINKER_FLAGS "-static -static-libgcc -static-libstdc++")
ELSE()
    SET(BUILD_SHARED_LIBS ON) #shared library building
ENDIF(STATIC)

if(verbose)
    set (CMAKE_VERBOSE_MAKEFILE ON)
endif(verbose)

#####
# OPTION DEPENDANCES
#####
SET(Toulbar_NAME "toulbar2") # default project name

IF(ALL_APP)
    MESSAGE(STATUS "#####")
    MESSAGE(STATUS " TOULBAR2 and MENDELSOFT COMPILEMENT AND PACKAGING ")
    MESSAGE(STATUS "#####")
    SET(TOULBAR2_ONLY OFF)
    SET(MENDELSOFT_ONLY OFF)
    SET(MENDELSOFT ON)
    SET(TOULBAR2 ON)
    SET(ILOG OFF)
```

```
SET(LIBTB2 ON)
SET(XML ON)
SET(CPLEX OFF)
SET(boost ON)
set (Toulbar_NAME "toulbar2-all")
project(${Toulbar_NAME})
#description used in the package building
    SET(Toulbar_PACKAGE_DESCRIPTION "${Toulbar_NAME} is an open source C++
exact solver and library for graphical model optimization. It can solve MAP/Markov
Random Fields or Cost Function Networks/Weighted CSP.")
        SET(Toulbar_PACKAGE_SUMMARY "${Toulbar_NAME} exact solver for graphical
models.")

ENDIF(ALL_APP)

IF(MENDELSPORT_ONLY)
MESSAGE(STATUS "#####
MESSAGE(STATUS " MENDELSPORT COMPILATION AND PACKAGING ONLY")
MESSAGE(STATUS "#####

SET(MENDELSPORT ON)
SET(TOULBAR2 OFF)
SET(TOULBAR2_ONLY OFF)
SET(ILOG OFF)
SET(LIBTB2 OFF)
SET(XML OFF)
SET(CPLEX OFF)
SET(Boost OFF)
SET(MPI OFF)

#basename for packaging and versioning
set (Toulbar_NAME "mendelsoft")
#description used in the package building
SET (Toulbar_PACKAGE_DESCRIPTION
    "MendelSoft is an open source software which detects marker genotyping
incompatibilities (Mendelian errors only) in complex pedigrees using weighted
constraint satisfaction techniques. The input of the software is a pedigree data
with genotyping data at a single locus. The output of the software is a list of
individuals for which the removal of their genotyping data restores consistency.
This list is of minimum size when the program ends.")
    SET(Toulbar_PACKAGE_SUMMARY "${Toulbar_NAME} is an open source software which
detect Mendelian errors in complex pedigrees using weighted constraint
satisfaction techniques")

ENDIF(MENDELSPORT_ONLY)

IF(TOULBAR2_ONLY)
MESSAGE(STATUS "#####
MESSAGE(STATUS " TOULBAR2 Solver Compilation and Packaging Only")
MESSAGE(STATUS "#####

SET(MENDELSPORT OFF)
SET(TOULBAR2 ON)
SET(ILOG OFF)
SET(LIBTB2 OFF)
#basename for packagin and versionning
SET(Toulbar_NAME "toulbar2")
#description used in the package building
    SET(Toulbar_PACKAGE_DESCRIPTION "${Toulbar_NAME} is an open source C++
exact solver and library for graphical model optimization. It can solve MAP/Markov
Random Fields or Cost Function Networks/Weighted CSP.")
        SET(Toulbar_PACKAGE_SUMMARY "${Toulbar_NAME} exact solver for graphical
models.")
ENDIF(TOULBAR2_ONLY)
```

```
#####
IF(ILOG)
  set(LIBTB2INT ON)
ENDIF(ILOG)

SET(EXECUTABLE_OUTPUT_PATH bin/${CMAKE_SYSTEM_NAME})
SET(LIBRARY_OUTPUT_PATH lib/${CMAKE_SYSTEM_NAME})

#####
IF(WIN32)
  # option used for cross compilation
  set( EXE ".exe")
  set(COST INT_COST)
  set(LIBTB2INT OFF)
  set(XML OFF)
  set(CPLEX OFF)
  set(Boost OFF)
  set(MPI OFF)
  set(ILOG OFF)
  set(LIBTB2 OFF)

  MESSAGE STATUS "WIN32 on ."
  MESSAGE STATUS "COST ==> int."
  
# mingW32 env setup
include(${My_cmake_script}/mingw32-config.cmake)

ELSE(WIN32)

  SET(COST LONGLONG_COST)
  MESSAGE STATUS "COST ==> long long"

ENDIF(WIN32)

#####
# find opt libs
#####
#if(WITH_MEM_JEMALLOC)
#  find_package(Jemalloc)
#  if(JEMALLOC_FOUND)
#    link_directories(${JEMALLOC_LIBPATH})
#    SET(all_depends ${all_depends} "jemalloc")
#  endif()
#endif()

if(WITH_MEM_JEMALLOC)
  set( JEMALLOC_FOUND 0 )

if ( UNIX )
  FIND_PATH( JEMALLOC_INCLUDE_DIR
    NAMES
      jemalloc/jemalloc.h
    PATHS
      /home/allouche/work/kad/jemalloc/include
      $ENV{JEMALLOC_ROOT}
      $ENV{JEMALLOC_ROOT}/include
      ${CMAKE_SOURCE_DIR}/externals/jemalloc
  DOC
    "Specify include-directories that might contain jemalloc.h here."
  )
  FIND_LIBRARY( JEMALLOC_LIBRARY
    NAMES
      jemalloc libjemalloc JEMALLOC
    PATHS
      /home/allouche/work/kad/jemalloc/lib
```

```
$ENV{JEMALLOC_ROOT}/lib
$ENV{JEMALLOC_ROOT}
DOC "Specify library-locations that might contain the jemalloc library here."
)

if ( JEMALLOC_LIBRARY )
if ( JEMALLOC_INCLUDE_DIR )
set( JEMALLOC_FOUND 1 )
message( STATUS "Found JEMALLOC library: ${JEMALLOC_LIBRARY}")
message( STATUS "Found JEMALLOC headers: ${JEMALLOC_INCLUDE_DIR}")
else ( JEMALLOC_INCLUDE_DIR )
message(FATAL_ERROR "Could not find jemalloc headers! Please install
jemalloc libraries and headers")
endif( JEMALLOC_INCLUDE_DIR )
endif( JEMALLOC_LIBRARY )

mark_as_advanced( JEMALLOC_FOUND JEMALLOC_LIBRARY JEMALLOC_EXTRA_LIBRARIES
JEMALLOC_INCLUDE_DIR )
endif(UNIX)

endif()

MESSAGE(STATUS "search for GMP library")
INCLUDE(FindPkgConfig)
include(${My_cmake_script}/FindGmp.cmake)
SET (all_depends ${all_depends} "gmp")
INCLUDE_DIRECTORIES(${GMP_INCLUDE_DIR})

#CMAKE_DEPENDENT_OPTION(ILOG "ILOGLUE COMPILED" OFF "LIBTB2INT" OFF)
#####
INCLUDE(FindPkgConfig)
INCLUDE(FindGit)

# list of files used for compilation are included in source_file.cmake
# new file need to be added to this list
# you can also define your own list and add it to the wall list
# for example: SET (source_files ${source_files} ${my_file_2add})

include(${My_cmake_script}/source_files.cmake)

#####
IF(Boost)
include(${My_cmake_script}/Boost_option.cmake)
link_directories(${Boost_LIBRARY_DIR})
INCLUDE_DIRECTORIES(${Boost_INCLUDE_DIR})
ENDIF(Boost)

#####
IF(MPI)
# SET(CMAKE_C_COMPILER mpicc)
# SET(CMAKE_CXX_COMPILER mpicxx)
# SET(CMAKE_CXX_COMPILER /usr/mpicc/gcc/openmpi-1.10.3rc4/bin/mpicxx)
# SET(CMAKE_SYSTEM_PREFIX_PATH /usr/mpicc/gcc/openmpi-1.10.3rc4)
find_package(MPI REQUIRED)
include_directories(${MPI_INCLUDE_PATH})
SET (mpiflag OPENMPI)
ENDIF(MPI)

#####
# FLAG XML ON ==> xmlcps supported
#####
include(${My_cmake_script}/xmlcsp.cmake)

#####
# build executable
#####
```

```

INCLUDE_DIRECTORIES ( ${CMAKE_CURRENT_SOURCE_DIR}/${My_Source} )

IF(TOULBAR2)
    add_executable(toulbar2${EXE} ${source_files})
    ADD_CUSTOM_TARGET(gen_version ALL /bin/sh ${CMAKE_CURRENT_SOURCE_DIR}/cmake-
script/genVersionFile.sh)
    SET_SOURCE_FILES_PROPERTIES(ToulbarVersion.hpp PROPERTIES GENERATED 1)
    ADD_DEPENDENCIES(toulbar2${EXE} gen_version)
    IF(MPI)
        TARGET_LINK_LIBRARIES(toulbar2${EXE} ${all_depends} ${MPI_LIBRARIES})
        IF(MPI_COMPILE_FLAGS)
            set_target_properties(toulbar2${EXE} PROPERTIES COMPILE_FLAGS "$
{MPI_COMPILE_FLAGS}")
        ENDIF(MPI_COMPILE_FLAGS)
        IF(MPI_LINK_FLAGS)
            set_target_properties(toulbar2${EXE} PROPERTIES LINK_FLAGS "$
{MPI_LINK_FLAGS}")
        ENDIF(MPI_LINK_FLAGS)
    ELSE(MPI)
        TARGET_LINK_LIBRARIES(toulbar2${EXE} ${all_depends})
    ENDIF(MPI)

    IF(HBFS_MPI)
        SET(CMAKE_CXX_COMPILER mpic++)
    ENDIF(HBFS_MPI)

# BOOST
#   set(BOOST_LIBRARYDIR /usr/lib64/mpich/lib) # to test boost mpich on fedora
#   set(BOOST_LIBRARYDIR /usr/lib/x86_64-linux-gnu) # boost openmpi on ubuntu
#   set(BOOST_LIBRARYDIR /tools/libraries/Boost/boost_1_70_0_openmpi-2.1.2/lib)
#   include_directories(/tools/libraries/Boost/boost_1_70_0_openmpi-2.1.2/include)
# -I/tools/libraries/Boost/boost_1_70_0_openmpi-2.1.2/include
# find_package(Boost 1.70.0 REQUIRED) # Fail with error if Boost is
not found
# find_package(Boost 1.70.0 REQUIRED mpi serialization)
# include_directories(${Boost_INCLUDE_DIR})
# set(Boost_NO_SYSTEM_PATHS ON) #Set to ON to disable searching in locations not
# specified by these hint variables. Default is OFF.

# MPI
find_package(MPI 2.1.2 REQUIRED)
include_directories(${MPI_INCLUDE_PATH})
set(MPI_CXX_LIBRARIES /tools/cluster/mpi/openmpi/2.1.2/gcc-4.8.5/lib)
set(MPI_CXX_INCLUDE_PATH /tools/cluster/mpi/openmpi/2.1.2/gcc-4.8.5/include)
include_directories(${MPI_INCLUDE_PATH})

set(MPI_CXX_HEADER_DIR /tools/cluster/mpi/openmpi/2.1.2/gcc-4.8.5/include)

    TARGET_LINK_LIBRARIES(toulbar2${EXE} ${all_depends} ${MPI_LIBRARIES} $
{Boost_LIBRARIES}
"/tools/libraries/Boost/boost_1_70_0_openmpi-2.1.2/lib/libboost_serialization.so"
"/tools/libraries/Boost/boost_1_70_0_openmpi-2.1.2/lib/libboost_mpi.so" "/tools/
libraries/Boost/boost_1_70_0_openmpi-2.1.2/lib/libboost_iostreams.so" "/home/
allouche/work/kad/jemalloc/lib/libjemalloc.so")
set(MPI_COMPILE_FLAGS "-I/tools/libraries/Boost/boost_1_70_0_openmpi-2.1.2/
include")
    IF(MPI_COMPILE_FLAGS)
        set_target_properties(toulbar2${EXE} PROPERTIES COMPILE_FLAGS "$
{MPI_COMPILE_FLAGS}")
    ENDIF(MPI_COMPILE_FLAGS)
    IF(MPI_LINK_FLAGS)
        set_target_properties(toulbar2${EXE} PROPERTIES LINK_FLAGS "$
{MPI_LINK_FLAGS}")
    ENDIF(MPI_LINK_FLAGS)
    ELSE(HBFS_MPI)
        TARGET_LINK_LIBRARIES(toulbar2${EXE} ${all_depends})
    ENDIF(HBFS_MPI)

```

```
    INSTALL( TARGETS toulbar2${EXE} DESTINATION bin)
ENDIF(TOULBAR2)

#####
# mendelsoft compilation
#####
IF(MENDELSPORT)
    add_executable(mendelsoft${EXE} ${source_files})
    ADD_CUSTOM_TARGET(gen_version ALL /bin/sh ${CMAKE_CURRENT_SOURCE_DIR}/cmake-
script/genVersionFile.sh)
    SET_SOURCE_FILES_PROPERTIES(ToulbarVersion.hpp PROPERTIES GENERATED 1)
    TARGET_LINK_LIBRARIES(mendelsoft${EXE} ${all_depends})
    install( TARGETS mendelsoft${EXE} DESTINATION bin)
ENDIF(MENDELSPORT)

# we must delete INT_COST
#####
# LIBTB2INTCOST GENERATION
#####
IF(LIBTB2)
    IF(WIN32)
        MESSAGE(STATUS "LIBTB2 not compliant with win32 in cost=LONG / cost type is
now int .....")
    ENDIF(WIN32)
    # INCLUDE_DIRECTORIES ( ${CMAKE_CURRENT_SOURCE_DIR}/${My_Source} )
    ADD_CUSTOM_TARGET(gen_version ALL /bin/sh ${CMAKE_CURRENT_SOURCE_DIR}/cmake-
script/genVersionFile.sh)
    SET_SOURCE_FILES_PROPERTIES(ToulbarVersion.hpp PROPERTIES GENERATED 1)

    LINK_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR})

    add_library(
        tb2
        ${LIBTB2FILE}
    )
    target_link_libraries(tb2 gmp)

    INSTALL(TARGETS tb2
        RUNTIME DESTINATION bin
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib
    )
ENDIF(LIBTB2)

#####
# LIBTB2INTCOST GENERATION
#####
IF(LIBTB2INT)
    MESSAGE(STATUS "COMPILING LIBTB2 INT .....")
    # INCLUDE_DIRECTORIES ( ${CMAKE_CURRENT_SOURCE_DIR}/${My_Source} )
    LINK_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR})
    ADD_CUSTOM_TARGET(gen_version ALL /bin/sh ${CMAKE_CURRENT_SOURCE_DIR}/cmake-
script/genVersionFile.sh)
    SET_SOURCE_FILES_PROPERTIES(ToulbarVersion.hpp PROPERTIES GENERATED 1)
    add_library(
        tb2int
        ${LIBTB2FILE}
    )

    INSTALL(TARGETS tb2int
        RUNTIME DESTINATION bin
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib
    )
ENDIF(LIBTB2INT)
```

```
ENDIF(LIBTB2INT)

#####
# Ilog Solver 6.0
#####
include(${My_cmake_script}/iloglue.cmake)

#####
# define option
#####
include(${My_cmake_script}/define_option.cmake)
MESSAGE(STATUS "##### define_option passed#####")

#####
# toulbar2test
#####
include(${My_cmake_script}/toulbar2test.cmake)

#####
# unit tests
#####
IF(COVER_TEST)
include(${My_cmake_script}/test.cmake)

MESSAGE(STATUS "##### test.cmake ==> tests script generated")
MESSAGE(STATUS "##### CTEST : toulbar2 default option = ${TOPT} (-DTOPT= ...to
change it) #####")

#####
# Cover tests
#####
include(${My_cmake_script}/cover-test.cmake)
ENDIF(COVER_TEST)

IF(BENCH)
include(${My_cmake_script}/test_bench.cmake)
include(${My_cmake_script}/add_make_command.cmake)

configure_file(${CMAKE_CURRENT_SOURCE_DIR}/misc/script/MatchRegexp.txt
${CMAKE_CURRENT_BINARY_DIR}/MatchRegexp.txt COPYONLY)

configure_file(${CMAKE_CURRENT_SOURCE_DIR}/misc/script/run_test.pl
${EXECUTABLE_OUTPUT_PATH}/run_test.pl COPYONLY)

configure_file(${CMAKE_CURRENT_SOURCE_DIR}/misc/script/make_report.pl
${CMAKE_CURRENT_BINARY_DIR}/make_report.pl COPYONLY)

configure_file(${CMAKE_CURRENT_SOURCE_DIR}/misc/script/exp_opt.pl
${CMAKE_CURRENT_BINARY_DIR}/exp_opt.pl COPYONLY)

ENDIF (BENCH)

#####
# Doc generation
#####
IF (BUILD_API_DOC)
include(${My_cmake_script}/UseDoxygen.cmake)
ENDIF (BUILD_API_DOC)

ADD_CUSTOM_TARGET(man ALL)

ADD_CUSTOM_COMMAND(
TARGET man
DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/man/toulbar2.1
COMMAND cp ${CMAKE_CURRENT_SOURCE_DIR}/man/toulbar2.1 $
```

```
{CMAKE_CURRENT_BINARY_DIR}/toulbar2.1
    OUTPUTS ${CMAKE_CURRENT_BINARY_DIR}/toulbar2.1
)

#####
# doc install
#####
install(DIRECTORY ${My_doc} DESTINATION ${doc_destination}/${Toulbar_NAME} PATTERN
".svn" EXCLUDE)
install(FILES ${CMAKE_CURRENT_BINARY_DIR}/toulbar2.1 DESTINATION share/man/man1)

#####
# examples installation
#####
install(DIRECTORY ${Default_validation_dir} DESTINATION ${doc_destination}/${
{Toulbar_NAME}/examples/ PATTERN ".svn" EXCLUDE)
install(DIRECTORY ${Default_cover_dir} DESTINATION ${doc_destination}/${
{Toulbar_NAME}/examples/ PATTERN ".svn" EXCLUDE)

#####
# PACKAGING
#####
set(CPACK_PROJECT_CONFIG_FILE ${CMAKE_CURRENT_SOURCE_DIR}/src/MyCPackConf.cmake)
INCLUDE(InstallRequiredSystemLibraries)
include(${My_cmake_script}/package.cmake)

##### END
```

10.4 Annexe D : Utilisation de toulbar2 avec l'option eps

Génération des sous-problèmes avec HBFS

L'option `-eps` a été ajoutée à la longue liste d'options de toulbar2. On trouvera ci-dessous un exemple d'utilisation avec mesure du *wall time* en secondes.

```
/usr/bin/time -f "Generation Time: %es" ./toulbar2 404.wcsp -eps
```

Cette commande produit la sortie ci-dessous et 3 fichiers :

1. `eps.sh` : script d'exécution de toulbar2 en parallel à adapter si nécessaire,
2. `nbProcess.txt` : contient le nombre de sous-problèmes(SP) à générer. Ce n'est qu'une valeur cible approximative. Le nombre effectif de sous-problèmes générés sera en général différent. Si le fichier existe, le nombre-cible de SP sera lu dans ce fichier, sinon il est créé avec un nombre = $30k$ avec k nombre de processus. L'idée derrière ce fichier était de mettre en place un apprentissage pour déterminer automatiquement, en fonction de la plateforme, le nombre-cible optimal de SP à générer.
3. `subProblems.txt` : chaque ligne contient un sous-problème de la forme ci-dessous qui décrit une séquence de décisions $v.\delta$. Elle correspond à un nœud dans l'arbre de recherche.

```
-x=", 44#0, 46=1, 60=1, 45=1, 50=1, 71=0, 58=1, 88=1, 61=1, 89=1, 62=1, 11=1  
, 63=1, 86#1"
```

L'option `-x` de toulbar2 permet de résoudre ce SP spécifique. `44#0` signifie par exemple que la décision sur la variable de rang 44 est de retirer la valeur 0 de son domaine, etc.

Dans cet exemple, 120 sous-problèmes ont été demandés mais 141 sont produits par toulbar2 pour obtenir une *partition* du problème global; si ce n'est pas une *partition*, l'espace d'états ne sera pas exploré exhaustivement et donnera probablement une solution sous-optimale.

```
c ./toulbar2 version : 1.0.1-449-gb040efb-kad (1567241515),  
copyright (c) 2006-2019, toulbar2 team  
HBFS Embarrassingly Parallel Search activated.  
loading wcsp file: ./404.wcsp  
Read 100 variables, with 4 values at most, and 710 cost functions,  
with maximum arity 3.  
Cost function decomposition time : 4e-05 seconds.  
Reverse DAC dual bound: 65 (+13.846%)  
Reverse DAC dual bound: 66 (+1.515%)  
Preprocessing time: 0.004 seconds.  
88 unassigned variables, 226 values in all current domains  
(med. size:2, max size:4) and 594 non-unary cost functions (med. degree:1)  
Initial lower and upper bounds: [66, 158] 58.228%
```

```

SEQUENTIAL HBFS MODE!!! ADD -para OPTION FOR PARALLEL MODE
New solution: 122 (0 backtracks, 17 nodes, depth 18)
Optimality gap: [67, 122] 45.082 % (17 backtracks, 34 nodes)
New solution: 120 (17 backtracks, 77 nodes, depth 43)
Optimality gap: [73, 120] 39.167 % (59 backtracks, 119 nodes)
Optimality gap: [83, 120] 30.833 % (94 backtracks, 195 nodes)
Optimality gap: [84, 120] 30.000 % (123 backtracks, 261 nodes)
TOTAL NUMBER OF NODES ADDED IN OPEN LISTE :146
NUMBER OF NODES IN OPEN LISTE WHEN openNodeLimit is reached : 141
BEST CURRENT SOLUTION FOUND AT DUMP TIME : UB = 120
NUMBER OF SUBPROBLEMS REALLY GENERATED : 141
Text has been written in file subProblems.txt
Text has been written in file eps.sh
Temps d'execution : 0.01s

```

Remarque 1 : Outre la partition en sous-problèmes, hbfs fournit la meilleure solution au moment de la production des SP, c'est à dire au moment du *dump*. Cette dernière peut être utilisée par toulbar2 dans la phase parallèle.

Remarque 2 : Une instruction tel que "if(node.getCost() < wcsp->getUb())" élimine les nœuds qui ne peuvent de donner de solution. C'est la condition de non-élagage.

Remarque 3 : La production des sous-problèmes par toulbar2 bénéficie du pré-processing. Les 3 remarques précédentes montrent que les SP générés seront de bonne qualité.

Phase parallel

La commande ci-dessous donne un exemple d'utilisation de GNU parallel sur une machine à 4 coeurs. On utilise la solution $UB = 120$ trouvée au moment du dump. Chaque sous-problème du fichier *subProblems.txt* est transmis à GNU Parallel qui assure l'assignation des processus à 4 processeurs. Ainsi, dans cet exemple, on a demandé $30k = 30 * 4 = 120$ sous-problèmes, 146 nœuds sont passés par la file open, 141 ont effectivement été produits. La génération a pris 0.1s. Ce temps est négligeable mais ce n'est pas le cas pour d'autres problèmes.

```
cat subProblems.txt | time parallel -j4 --eta -k ./toulbar2 404.wcsp
-ub=120 {}
```

10.5 Annexe E : Résumé étendu pour la CP 2019

Parallel Hybrid Best-First Search for Cost Function Networks

Abdelkader Beldjilali, David Allouche, Simon de Givry

INRA, MIA Toulouse, UR-875
31320 Castanet-Tolosan, France

Introduction

Cost Function Networks (CFNs), also known as Weighted Constraint Satisfaction Problems (Meseguer, Rossi, and Schiex 2006) is a mathematical model which has been derived from Constraint Satisfaction Problems by replacing constraints with cost functions. In a CFN, we are given a set of variables with an associated finite domain and a set of local cost functions. Each cost function involves some variables and associates a non-negative integer cost to each of the possible combinations of values they may take. The usual problem considered is to assign all variables in a way that minimizes the sum of all costs. This problem is NP-hard, and exact methods usually rely on Branch and Bound (B&B) algorithms exploring a binary search tree with propagation at each node in order to improve the problem lower bound and prune domain values with a forbidden cost (Cooper et al. 2010). Several B&B search methods have been developed in the CFN solver `toulbar2`¹.

In this work we describe a first parallel version of Hybrid Best-First Search (HBFS) (Allouche et al. 2015) and give a preliminary empirical evaluation on combinatorial optimization problems from uncapacitated warehouse location, computational protein design, and genetics. This last section includes solving time and speed-up comparisons between our approach within `toulbar2` and other parallel approaches available in IBM Ilog `cplex` and `daoopt` (Otten and Dechter 2017) (i.e., respectively a Mixed Integer Programming and an AND/OR search Graphical Model solver).

Parallel HBFS

The sequential version of HBFS (Allouche et al. 2015) is a B&B method for CFNs that combines Best-First Search (BFS) and Depth-First Search (DFS). Like BFS, HBFS provides an anytime global lower bound on the optimum, while also providing anytime upper bounds, like DFS. Hence, it provides feedback on the progress of search and solution quality in the form of an optimality gap. Besides, it exhibits highly dynamic behavior that allows it to perform on par with methods like Limited Discrepancy Search (LDS) and frequent restarting in terms of quickly finding good solutions. As in BFS, HBFS maintains a frontier of open search nodes. It expands each open node using DFS with a limit on

its number of backtracks. Each bounded DFS returns a new list of open nodes to be inserted in the BFS frontier.

The parallel version of HBFS is based on the Master-Worker parallel paradigm (Ralphs et al. 2018) where the *Master* is in charge of the open node frontier and dispatches the current best (with minimum lower bound) open node plus the current best solution found so far to the next available *Worker*. The *Worker* performs a bounded DFS starting from the received node and returns to the *Master* the resulting list of open nodes (see Fig. 1, with a DFS limit of 3 backtracks). Each open node is associated to a corresponding lower bound and a vector of search decisions. The *Worker* also returns the best solution found during its limited search if any. Only the *Master* has a global view of the whole search and reports optimality gaps until the proof of optimality is reached (when the current best frontier lower bound, including active *Worker* starting nodes, is equal or greater than the cost of the best solution found so far).

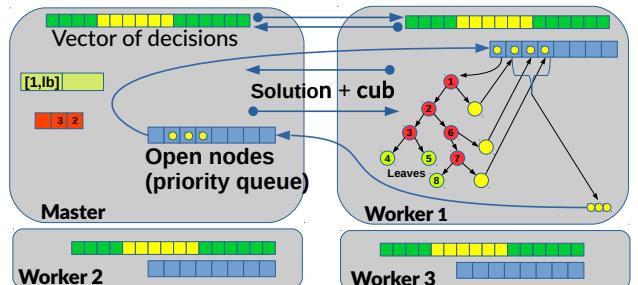


Figure 1: Parallel HBFS using the Master-Worker paradigm.

Instance	n	d	Time (sec.)		Speed-up
			HBFS	HBFS-24	
capmo1	200	100	10.92	5.14	2.12
capmo2	200	100	1.80	2.04	0.88
capmo3	200	100	6.09	3.73	1.63
capmo4	200	100	4.36	3.21	1.36
capmo5	200	100	2.69	2.58	1.04

Table 1: Warehouse benchmark (Larrosa et al. 2005) with *n*, number of variables, and *d*, maximum domain size.

Experimental Results

We implemented parallel HBFS inside `toulbar2` using the MPI library. Experiments were performed on 24-core

¹<https://github.com/toulbar2/toulbar2>

servers (Intel Xeon E5-2680/87 at 2.50/3GHz and 256 GB) and the GenoToul cluster (64-core nodes of Intel Xeon E5-2683 at 2.10GHz).

Instance	<i>n</i>	<i>d</i>	Time (sec.)		Speed-up
			HBFS	HBFS-24	
1xaw	107	412	721.43	568.50	1.27
3lf9	120	416	407.28	407.92	1
5dbl	130	384	122.84	171.82	0.71
5e10	133	400	147.73	198.23	0.75
5e0z	136	420	148.26	193.41	0.77
5eqz	138	434	3,366.11	1,049.11	3.21
1dvo	152	389	622.03	463.29	1.34
4bxp	170	439	327.46	395.16	0.83
1is1	185	431	-	2,545.82	-
2gee	188	397	797.22	863.64	0.92
5jdd	263	406	-	2,758.98	-
3r8q	271	418	1,605.30	1,294.97	1.24
1f00	282	430	-	2,140.40	-

Table 2: Computational Protein Design (CPD) benchmark (Ouali et al. 2017). A ‘-’ indicates that the corresponding method failed to prove optimality in less than 1 hour.

Instance	<i>n</i>	<i>d</i>	cplex	cplex-10	HBFS	HBFS-10	Speed-up
1UBI	13	198	-	-	1,023	214.02	4.78
2DHC	14	198	-	-	8.2	5.83	1.41
2DRI	37	186	-	-	135.5	30.00	4.52
1CDL	40	186	-	-	392.6	54.95	7.14
1CMI	42	186	-	6,177	6.6	6.11	1.08
1BRS	44	194	-	-	555.3	107.86	5.15
1GVP	52	182	-	-	596.1	185.75	3.21
1IRIS	56	182	-	-	129.7	36.23	3.58
3CHY	74	66	-	5,259	88.7	20.71	4.28

Table 3: Another CPD benchmark (Allouche et al. 2014). A ‘-’ indicates that the corresponding method failed to prove optimality in less than 9,000 seconds. Only instances solved in more than 5 sec. by any successful method are reported.

	pedigree19	pedigree31	pedigree44	pedigree51
<i>n</i>	793	1,183	811	1,152
<i>d</i>	5	5	4	5
cplex	790	59.30	6.35	36.23
//10	191 (4.14)	9.00 (6.59)	2.48 (2.56)	9.43 (3.84)
//30	75 (10.53)	7.17 (8.27)	2.69 (2.36)	5.34 (6.78)
daoopt	375,110	16,238	95,830	101,788
//20	27,281 (13.75)	1,055 (15.39)	6,739 (14.22)	6,406 (15.89)
//100	7,492 (50.07)	201 (80.79)	1,799 (53.27)	1,578 (64.50)
HBFS	3,126	4.34	39.72	1,608
//10	434.27 (7.20)	1.51 (2.87)	6.08 (6.53)	179.22 (8.97)
//20	227.02 (13.77)	1.39 (3.12)	3.18 (12.49)	72.30 (22.24)
//100	119.43 (26.17)	0.97 (4.47)	1.64 (24.22)	31.40 (51.21)

Table 4: Linkage benchmark (Favier et al. 2011) with different number of cores (speed-up in parentheses).

We report in Table 1 and Table 2 solving times to find and prove optimality on Warehouse and CPD benchmarks for the sequential and 24-core parallel versions of HBFS. Parallel HBFS solved three more CPD instances within the 1-hour time-out. The maximum speed-up was 2.12 (resp. 3.21) for Warehouse (resp. CPD), which is rather limited compared to the number of cores used. Experiments on difficult instances of another CPD benchmark using only 10 cores resulted in better speed-ups (up to 7.14 on 1CDL, see Table 3). Moreover our CFN approach was much faster than cplex. In Table 4, we compared cplex, daoopt, and HBFS on the

Linkage benchmark. We report daoopt time from (Otten and Dechter 2017), obtained on a cluster of dual 2.67 GHz Intel Xeon X5650 6-core CPUs and 24 GB of RAM. Here, the sequential version of HBFS is dominated by cplex on three instances among four. But, the parallel version of HBFS got better relative speed-ups than cplex when the number of cores increases. We found daoopt got very good speed-ups on these instances but still was far from cplex in terms of CPU times.

Conclusions

Parallel HBFS is a first parallel approach for HBFS. It provides interesting results on several instances, outperforming in some cases state of the art solvers like cplex and daoopt. Even if the scalability of our approach must be subject of deeper investigation, due to the minimal size of the information shared between the Master and the Workers, the approach is very likely compliant with a larger number of cores. We found that the speed-up was very instance dependent, and must be also investigated.

As future work, we will take into account the structure of CFNs by parallelizing Backtrack with Tree Decomposition (BTD-HBFS) (Allouche et al. 2015). The resulting parallel method could replace LDS inside a parallel large neighborhood search strategy (Ouali et al. 2017) offering better anytime lower and upper bounds.

Acknowledgments This work has been partially funded by the french Agence nationale de la Recherche, reference ANR-16-C40-0028. We are grateful to the genotoul bioinformatics platform Toulouse Midi-Pyrénées (Bioinfo Genotoul) for providing computing and storage resources.

References

- Allouche, D.; André, I.; Barbe, S.; Davies, J.; de Givry, S.; Katsirelos, G.; O’Sullivan, B.; Prestwich, S.; Schiex, T.; and Traoré, S. 2014. Computational protein design as an optimization problem. *Artificial Intelligence* 212:59–79.
- Allouche, D.; de Givry, S.; Katsirelos, G.; Schiex, T.; and Zytnicki, M. 2015. Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP. In *Proc. of CP*, 12–28.
- Cooper, M.; de Givry, S.; Sanchez, M.; Schiex, T.; Zytnicki, M.; and Werner, T. 2010. Soft arc consistency revisited. *AI* 174:449–478.
- Favier, A.; Givry, S.; Legarra, A.; and Schiex, T. 2011. Pairwise decomposition for combinatorial optim. in graphical models. In *Proc. of IJCAI*, 2126–2132.
- Larrosa, J.; de Givry, S.; Heras, F.; and Zytnicki, M. 2005. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *Proc. of IJCAI*, 84–89.
- Meseguer, P.; Rossi, F.; and Schiex, T. 2006. Soft constraints processing. In *Handbook of Constraint Programming*. Elsevier. chapter 9, 279–326.
- Otten, L., and Dechter, R. 2017. And/or branch-and-bound on a computational grid. *JAIR* 59:351–435.
- Ouali, A.; Allouche, D.; de Givry, S.; Loudni, S.; Lebbah, Y.; Eckhardt, F.; and Loukil, L. 2017. Iterative Decomposition Guided Variable Neighborhood Search for Graphical Model Energy Minimization. In *Proc. of UAI-17*, 550–559.
- Ralphs, T.; Shinano, Y.; Berthold, T.; and Koch, T. 2018. *Handbook of Parallel Constraint Reasoning*. Springer. chapter Parallel Solvers for Mixed Integer Linear Optimization, 283–336.



Acronymes

ABDFBB Adaptive Bounded Depth First Search B&B : Le B&B avec parcours DFS est borné par le nombre Z de retours arrière ou backtracks autorisés ; une fois atteinte cette limite Z, le DFS s'interrompt . Il est adaptatif car cette borne Z varie selon une certaine heuristique.

B&B Branch and Bound ou Evaluation-Séparation : méthode générique très utilisée en recherche opérationnelle. Cf. par exemple <https://www.techno-science.net/definition/6348.html> ou <https://www.youtube.com/watch?v=E7hJXsywOdA>.

BFS Breadth First Search.

BNFS Best Node First Search : L'acronyme BNFS est utilisé ici pour éviter l'ambiguïté avec BFS : <https://www.techiedelight.com>. *The defining characteristic of BNFS is that, unlike DFS or BFS (which blindly examines/expands a cell without knowing anything about it or its properties), best-first search uses an evaluation function (sometimes called a "heuristic") to determine which object is the most promising, and then examines this object. This "best first" behaviour is implemented with a Priority Queue.* Cf. <https://courses.cs.washington.edu/courses/cse326/03su/homework/hw3/bestfirstsearch.html>.

BTD-HBFS Backtracking with Tree Decomposition Hybrid Best-node First Search.

CFN Cost Function Network : Le problème est un hyper-graphe dont les nœuds sont les variables et les arêtes les contraintes.

DFBB Depth First Search with B&B.

DFS Depth First Search.

EDAC Existential Directional Arc Consistency.

EPA Etablissement Public à Caractère Administratif.

EPS Embarrassingly Parallel Search.

EPST Etablissement Public à Caractère Scientifique et Technologique.

EPT Equivalent Preserving Transformations.

FOSS Free Open Source Software.

HBFS Hybrid Best-First Search.

HPC High Performance Computing.

IDE Integrated Development Environment.

ILP Integer Linear Programming.

INRA Institut National de Recherche Agronomique.

INRAE Institut National de Recherche pour l’Agriculture, l’Alimentation et l’Environnement.

IRSTEA Institut National de Recherche en Sciences et Technologies pour l’Environnement et l’Agriculture.

JSON JavaScript Object Notation.

MIA département de Mathématiques et Informatique.

MIAT unité de Mathématiques et Informatique de Toulouse.

MILP Mixed-Integer Linear Programming.

MPI Message Passing Interface.

PFE Projet de Fin d’Etudes.

S&E Séparation et Evaluation.

SAB équipe Statistiques et Algorithmique pour la Biologie.

SLURM Simple Linux Utility for Resource Management.

SPMD Single Program Multiple Data.

VAC Virtual Arc Consistency.

VCSP Valued Constraint Satisfaction Problem.

WCSP Weighted Constraint Satisfaction Problems.



Glossaire

Arité Nombre de variables impliquées dans une fonction de coûts. L'ensemble de ces variables est appelé Support. L'arité est donc le cardinal du support de la contrainte.

Cluster Un cluster est une grappe d'ordinateurs, appelés nœuds en référence à la théorie des graphes. Relativement homogènes, ces derniers sont connectés par un réseau local, propriété d'une entité déterminée et unique qui en assure la gestion. Les clusters fournissent la partie calculatoire des services fournis par les data centers. https://fr.wikipedia.org/wiki/Grappe_de_serveurs. Chaque nœud se présente souvent sous la forme d'une carte mère montée dans des racks, eux-mêmes montés dans des baies, elles-mêmes installées dans un local climatisé et sécurisé. Mais on peut monter son propre cluster avec un ensemble d'ordinateurs, de consoles de jeux, etc. Il est aussi possible d'acheter des services de calcul dans les data centers des "clouds"^[4] qui désignent une infrastructure également centralisée qui peut faire usage de clusters, de superordinateur, pour fournir divers niveaux de prestations.

contrainte souple synonyme de contrainte valuée et de fonction de coûts.

contrainte souple d'arité nulle notée w_\emptyset , elle correspond à une fonction de coûts constante, de support vide donc indépendante de toute variable du problème.

cpu Central Processing Unit : terme qu'on utilisera comme synonyme de cœurs ou de processeur qui désignent le processeur physique qui exécute un processus.

Gap d'optimalité Intervalle qui encadre la valeur optimale, noté dans ce rapport $[clb, cub]$ ou $[LB, UB]$.

genologin Désigne la nouvelle tranche du cluster installée en 2017. C'est aussi le nom des frontaux genologin1 et genologin2. Il comprend 48 nœuds, du node101 au node148, 32 cœurs et 256 GB de RAM par nœud, interconnexion par InfiniBand 56GB/s. Workload manager SGE remplacé par SLURM. Total : 1584 cœur / 3168 threads / 51 TFlops.

GenoToul Plateforme bioinformatique faisant partie du réseau GenoToul qui met à disposition des ressources (calcul, stockage, banques de données, logiciels) et des compétences pour accompagner les programmes de biologie et de bioinformatique aux plans régional, national et international. La capacité de calcul, en août 2019, est assurée par un cluster comprenant environ 3000 coeurs réels installé dans le data center, ou arche de données, du centre de recherche de l'INRA Toulouse. <http://bioinfo.genotoul.fr/>.

grid computing Les grilles de calcul, à la différence des clusters, constituent un assemblage d'ordinateurs hétérogènes connectés par un réseau parfois de portée mondiale, e.g. l'Internet. Les propriétaires/administrateurs sont multiples, non connus en général, le matériel et les logiciels sont hétérogènes.https://fr.wikipedia.org/wiki/Grid_computing. Exemples de projets de calcul distribué par grille : détection d'intelligence extra-terrestre : <https://fr.wikipedia.org/wiki/SETI@home>.

hypergraphe graphe dont les nœuds peuvent être reliés par plus de 1 arrête, ou plus de 2 arcs s'il est orienté.

InfiniBand Standard de communication à haut débit de l'ordre d'une dizaine ou centaine de Gbit/s et de latence inférieure à la microseconde.

Latence Dans un contexte MPI, la latence représente le temps nécessaire pour qu'un émetteur et un récepteur s'échangent un message vide.

Master-Worker Paradigme de parallélisation qui utilise un processus master qui distribue le travail aux workers et centralise les résultats obtenus par ces derniers.

nœud Elément constitutif d'un système comprenant au moins un processeur muni de sa propre mémoire.

processus programme en cours d'exécution auquel le système d'exploitation octroie des ressources, en particulier un espace mémoire privé. Il possède un état.

Speedup Rapport entre le temps d'exécution d'un programme séquentiel et le temps du d'exécution du même programme parallélisé.

SPMD Single Program, Multiple Data : modèle d'exécution où chaque CPU exécute le même programme indépendamment sauf en ce qui concerne les échanges de messages comme avec le standard MPI. Les données manipulées ne sont pas forcément identiques.

supercomputer Bien qu'un cluster puisse être considéré comme un super ordinateur, la définition retenue ici concerne des systèmes qui sont davantage intégrés et non simplement constitués d'un ensemble de nœuds indépendants. Dans un super ordinateur, vu comme une unité d'exécution unique, l'intégration des processeurs permet des communications plus rapides entre processeurs et entre mémoire et processeurs. Probablement que les clusters sont plus adaptés à l'exécution de programmes se prêtant au paradigme *Embarrassingly Parallel* : bio-informatique, phy-

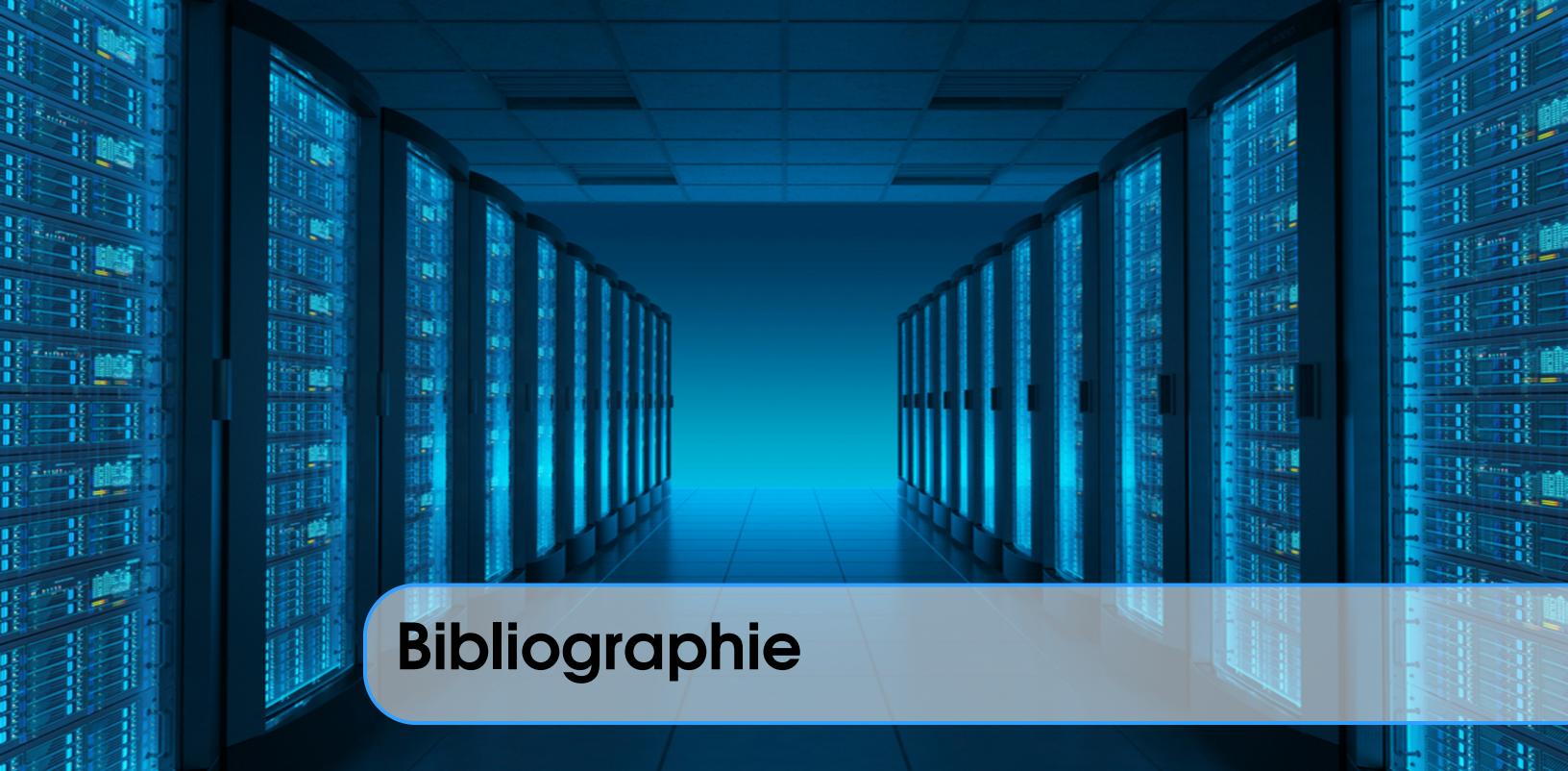
sique des particules et les super ordinateurs, aux traitements de tâches en parallèles dépendantes les unes des autres : mécanique des fluides, prévisions météorologiques. Les performances de HBFS seraient ainsi supérieures sur un super ordinateur. Exemple de supercomputer : IBM Blue Gene/Q baptisé MIRA dont une photo illustre l'en-tête de ce glossaire. By Courtesy Argonne National Laboratory, CC BY 2.0, [https://en.wikipedia.org/wiki/Mira_\(supercomputer\)](https://en.wikipedia.org/wiki/Mira_(supercomputer)).

thread Appelé aussi processus léger. S'exécute dans l'espace mémoire d'un processus "lourd". Il possède sa propre pile mais partage l'espace mémoire avec d'autres threads.

toulbar2 Contraction de Toulouse et Barcelone. Le 2 indique la version C++ de toulbar programmé en C. Solveur exact de divers problèmes d'optimisation combinatoire décrits par un réseau de fonctions de coûts : <http://www7.inra.fr/mia/T/toulbar2/>, <https://github.com/toulbar2/toulbar2>.

Tâche Programme constitué d'une séquences d'instructions qui décrivent les actions nécessaires à la réalisation de cette tâche. Une tâche décrit ce qu'il faut faire. Une tâche est effectuée par un processus qui utilise un processeur.

échelonnable Scalabilité ou Scalability : Capacité d'un algorithme à tirer partie d'une augmentation des ressources, principalement du nombre de cœurs. Nécessite d'éviter les goulots d'étranglement dans les communications.



Bibliographie

Articles

- [1] D. ALLOUCHE, S. De GIVRY et T. SCHIEX. “Towards Parallel Non Serial Dynamic Programming for Solving Hard Weighted CSP”. In : (2010) (cf. pages 25, 39).
- [2] D ALLOUCHE, J DAVIES, S de GIVRY, G KATSIRELOS, T SCHIEX, S TRAORÉ, I ANDRÉ, S BARBE, S PRESTWICH et B O’SULLIVAN. “Computational Protein Design as an Optimization Problem”. In : AI 212 (2014), pages 59-79 (cf. page 102).
- [3] David ALLOUCHE, Simon de GIVRY, George KATSIRELOS, Thomas SCHIEX et Matthias ZYTNICKI. “Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP”. In : (2015). Sous la direction de Gilles PESANT, pages 12-29 (cf. pages 25, 48, 63).
- [4] Michael ARMBRUST, Armando FOX, Rean GRIFFITH, Anthony D. JOSEPH, Randy H. KATZ, Andrew KONWINSKI, Gunho LEE, David A. PATTERSON, Ariel RABKIN, Ion STOICA et Matei ZAHARIA. “Above the Clouds: A Berkeley View of Cloud Computing”. In : UCB/EECS-2009-28 (2009). URL : <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html> (cf. page 133).
- [6] Martin COOPER, Simon DE GIVRY, Marti SANCHEZ, Thomas SCHIEX, Matthias ZYTNICKI et Tomas WERNER. “Soft arc consistency revisited”. anglais. In : *Artificial Intelligence* 174 (2010), pages 449-478. URL : http://www.irit.fr/publis/ADRIA/VAC_OSAC_final.pdf (cf. page 56).
- [7] A DJERRAH, Bertrand LECUN, Van-Dat CUNG et Catherine ROUCAIROL. “Bob++: Framework for Solving Optimization Problems with Branch-and-Bound methods”. In : (juin 2006), pages 369-370. DOI : [10.1109/HPDC.2006.1652188](https://doi.org/10.1109/HPDC.2006.1652188).

- [8] A FAVIER, S de GIVRY, A LEGARRA et T SCHIEX. “Pairwise decomposition for combinatorial optim. in graphical models”. In : (2011), pages 2126-2132 (cf. page 101).
- [9] Ian P. GENT, Chris JEFFERSON, Ian MIGUEL, Neil C. A. MOORE, Peter NIGHTINGALE, Patrick PROSSER et Chris UNSWORTH. “A Preliminary Review of Literature on Parallel Constraint Solving”. In : () .
- [10] Simon de GIVRY, Federico HERAS, Matthias ZYTNICKI et Javier LARROSA. “Existential arc consistency: Getting closer to full arc consistency in weighted CSPs”. In : (2005) (cf. pages 47, 56, 63).
- [11] Simon de GIVRY, Thomas SCHIEX et Gérard VERFAILLIE. “Exploiting Tree Decomposition and Soft Local Consistency In Weighted CSP”. In : (2006).
- [13] Philippe JÉGOU, Hélène KANSO et Cyril TERRIOUX. “Adaptive and Opportunistic Exploitation of Tree-Decompositions for Weighted CSPs”. In : (nov. 2017). DOI : [10.1109/ICTAI.2017.00064](https://doi.org/10.1109/ICTAI.2017.00064) (cf. page 25).
- [14] Akihiro KISHIMOTO, Radu MARINESCU et Adi BOTEA. “Parallel Recursive Best-first AND/OR Search for Exact MAP Inference in Graphical Models”. In : NIPS’15 (2015), pages 928-936. URL : <http://dl.acm.org/citation.cfm?id=2969239.2969343> (cf. pages 25, 39, 40).
- [15] Javier LARROSA, Emma ROLLON et Rina DECHTER. “Limited Discrepancy AND/OR Search and Its Application to Optimization Tasks in Graphical Models”. In : IJCAI’16 (2016), pages 617-623. URL : <http://dl.acm.org/citation.cfm?id=3060621.3060708> (cf. page 40).
- [16] Javier LARROSA, Simon S DE GIVRY, F HERAS et M ZYTNICKI. “Existential arc consistency: getting closer to full arc consistency in weighted CSPs”. In : (2005), pages 84-89 (cf. page 100).
- [17] Bernard MANS, Thierry MAUTOR et Catherine ROUCAIROL. “A parallel depth first search branch and bound algorithm for the quadratic assignment problem”. In : *European Journal of Operational Research* 81.3 (mar. 1995), pages 617-628. URL : <https://ideas.repec.org/a/eee/ejores/v81y1995i3p617-628.html> (cf. pages 25, 39).
- [19] Tarek MENOUER. “Solving combinatorial problems using a parallel framework”. In : *J. Parallel Distrib. Comput.* 112 (2018), pages 140-153. DOI : [10.1016/j.jpdc.2017.05.019](https://doi.org/10.1016/j.jpdc.2017.05.019). URL : <https://doi.org/10.1016/j.jpdc.2017.05.019> (cf. page 79).
- [20] Tarek MENOUER, Bertrand LECUN et P VANDER-SWALMEN. “Parallélisation d’un solveur de contraintes avec le framework parallèle BOBPP”. In : (jan. 2013) (cf. page 79).
- [21] L OTTEN et R DECHTER. “AND/OR Branch-and-Bound on a Computational Grid”. In : *JAIR* 59 (2017), pages 351-435.

- [22] Abdelkader OUALI, David ALLOUCHE, Simon DE GIVRY, Samir LOUDNI, Yahia LEBBAH, Francisco ECKHARDT et Lakhdar LOUKIL. “Iterative Decomposition Guided Variable Neighborhood Search for Graphical Model Energy Minimization”. In : (août 2017). URL : <https://hal.archives-ouvertes.fr/hal-01628162> (cf. pages 25, 39, 40, 101).
- [24] Ted RALPHS, Yuji SHINANO, Timo BERTHOLD et Thorsten KOCH. “Parallel Solvers for Mixed Integer Linear Optimization”. In : (2018), pages 283-336 (cf. pages 40, 67, 75, 76, 78-81, 94).
- [26] Jean-Charles RÉGIN, Mohamed REZGUI et Arnaud MALAPERT. “Embarrassingly Parallel Search”. In : (2013). Sous la direction de Christian SCHULTE, pages 596-610 (cf. pages 25, 39, 40, 77, 81-83).
- [27] Thomas SCHIEX, Helene FARGIER et Gerard VERFAILLIE. “Valued Constraint Satisfaction Problems: Hard and Easy Problems”. In : IJCAI’95 (1995), pages 631-637. URL : <http://dl.acm.org/citation.cfm?id=1625855.1625938> (cf. page 47).
- [28] “Weighted constraint satisfaction problem”. In : (2016). URL : https://en.wikipedia.org/wiki/Weighted_constraint_satisfaction_problem (cf. pages 47, 48).

Books

- [5] Martin C. COOPER, Simon GIVRY et Thomas SCHIEX. *Valued Constraint Satisfaction Problems*. Sous la direction de SPRINGER. Tome 2. Chapitre 7 (cf. pages 47, 49).
- [12] Ananth GRAMA, Anshul GUPTA, George KARYPIS et Vipin KUMAR. *Introduction to Parallel Computing*. Addison Wesley, 2003.
- [18] Timothy MATTSON, Beverly SANDERS et Berna MASSINGILL. *Patterns For Parallel Programming*. Addison-Wesley, 2004.
- [23] Peter PACHECO. *An Introduction to Parallel Programming*. Elsevier Science Technology, 2011 (cf. page 72).
- [25] Jean-Charles RÉGIN et Arnaud MALAPERT. *Parallel Constraint Programming*. Springer, 2018. Chapitre 9, pages 337-379 (cf. pages 25, 39, 77, 78, 84).