Comparison of Controller Synthesis and Scheduling Techniques for Dynamically Reconfigurable Allocation of Tasks on Computing Resources

Jolahn Vaudey

Master of Science in Informatics at Grenoble
Master Informatique
Specialization SHCE

Comparison of Controller Synthesis and Scheduling Techniques for Dynamically Reconfigurable Allocation of Tasks on Computing Resources

Jolahn VAUDEY

29 Août 2022

Research project performed at LIG / Inria Grenoble

Under the supervision of:
Gwenaël Delaval
Raphaël Bleuse

Defended before a jury composed of:
Head of the jury
Jury member 1
Jury member 2

Septembre 2022

Remerciements

Je tiens tout d'abord à remercier M. Rutten, responsable de l'équipe CTRL-A, de m'avoir accueilli au sein de sa structure.

Merci également à mes deux maîtres de stage, M. Delaval et M. Bleuse, pour les conseils prodigués et le suivi

Table des matières

— 1 —
Introduction

1.1 Contexte académique du sujet

Pour commencer, penchons-nous sur la synthèse de contrôleurs. L'idée derrière cette technique est, à partir d'une description formelle d'un système, de générer un contrôleur capable de maintenir ce système dans un état désirable, en observant les entrées/sorties et en contrôlant les commandes émises. Récemment, la synthèse de contrôleurs a été utilisée dans des recherches relevant de la cybersécurité [7], où le but était, lorsqu'un composant du système devenait compromis, de réattribuer les tâches qui lui étaient allouées aux machines restantes afin de maximiser le nombre de tâches maintenues dans un état nominal. Le langage de programmation utilisé était Heptagon, couplé à ReaX, un outil permettant de réaliser la synthèse de contrôleurs discrets. Des retours ont montré que cela ressemblait fortement à un problème d'ordonnancement, et qu'il serait donc intéressant de comparer la méthode utilisée aux techniques issues de la recherche en théorie de l'ordonnancement. Ce sujet de stage a ainsi été créé pour réaliser la comparaison de ces approches, sous la supervision de deux chercheurs chacun experts d'un des deux domaines abordés.

1.2 Description informelle des problèmes traités

Par rapport au sujet traité dans l'article traitant de cybersécurité, nous nous restreindrons ici à des problèmes plus simples et génériques, afin de se rapprocher de problèmes d'ordonnancement typiques. Nous nous plaçons dans une situation où nous devons répartir un certain nombre n de tâches, sur m machines, de manière à ce que toutes les tâches aient eu le temps de s'exécuter avant qu'un cycle ne soit écoulé. Les tâches sont en effet supposées se répéter tant qu'elles ne sont pas désactivées, et l'ordonnancement obtenu est donc une configuration cyclique qui reste en place jusqu'à ce qu'un changement advienne. Chacune de ces tâches nécessite une proportion de cycle différente selon la machine utilisée pour l'exécuter, et les tâches doivent s'effectuer en une seule fois, sans possibilité de changer de machine. Enfin, il est envisageable que certaines tâches, ou certaines machines, ne soient pas respectivement actives ou disponibles pendant certains cycles. On souhaitera ici traiter de problèmes tels que, si toutes les ressources sauf deux quelconques sont désactivées, toutes les tâches peuvent encore être ordonnancées.

1.3 Approches employées

Le stage a été divisé en trois parties de manière intuitive. Premièrement, il a fallu modéliser les problèmes à l'aide du langage Heptagon. Ensuite, nous avons dû trouver les algorithmes d'ordonnancement pertinent et les implémenter. Dans ces deux cas, un vecteur est renvoyé en sortie, indiquant pour chaque tâche la machine correspondante. Une fois que les deux méthodes précédentes ont été mises au point, il a fallu comparer les résultats obtenus.

1.3.1 Synthèse de contrôleurs

Pour commencer, nous avons donc abordé la synthèse de contrôleurs. En s'inspirant des travaux précédemment effectués [7] [11], l'idée était de créer un modèle Heptagon plus générique que ceux utilisés dans lesdites recherches. Cette méthode de génération de contrôleurs a pour avantage de permettre une réaction en temps supposé linéaire aux perturbations en ligne, mais la durée de la synthèse hors-ligne est exponentielle en temps. De plus, la mémoire utilisée croît également très rapidement, au point de limiter fortement les cas traitables. Avec cette méthode, toutes les caractéristiques du problème doivent être connues à l'avance, car nécessaires à la synthèse du contrôleur. Ce dernier, utilisé en ligne, ne prend en entrée que la configuration actuelle du système, c'est-à-dire les tâches et les ressources désactivées. Une fois la configuration actuelle connue, le contrôleur peut donc restituer la solution correspondante, calculée en amont pendant la synthèse. Heptagon est de plus un langage de quatrième génération, générant du code C ou Java à partir d'une spécification, ce qui permet au code Heptagon d'être concis, lisible et facile à écrire à partir d'une description du problème.

1.3.2 Techniques d'ordonnancement

L'autre partie du stage consistait à étudier les solutions existantes dans le domaine de l'ordonnancement permettant de remplir la même fonction. Deux principales difficultés se sont présentées pendant la recherche de tels algorithmes. Premièrement, le problème qui nous intéresse, la faisabilité d'un ordonnancement, n'est pas un problème d'optimisation, forme classique des problèmes d'ordonnancement traités dans l'état de l'art, mais plutôt de décision. Quel que soit l'algorithme retenu, il fallait donc l'adapter. De plus, ce problème de décision est fortement NP difficile [16]. Comme les techniques d'approximation disponibles ne permettent pas une comparaison correcte avec l'exécutable produit par Heptagon/BZR, nous avons donc utilisé en priorité des algorithmes exacts, donc exponentiels en pire cas.

Les méthodes retenues sont l'optimisation linéaire en nombres entiers et la programmation par contraintes. Ces techniques ont pour avantage de posséder une approche descriptive à la résolution de problèmes, de manière similaire à Heptagon. De plus, la programmation par contraintes est une approche résolvant des problèmes de décision, ce qui permet de résoudre notre problème de manière plus naturelle. Des heuristiques utilisant la programmation linéaire ou des techniques d'ordonnancement de listes ont également été étudiées. Ces techniques sont entièrement appliquées en ligne, et nécessitent en entrée la description complète du problème, soit les machines activées, les tâches activées et les temps d'exécution correspondants.

1.3.3 Méthodes de comparaison

Afin de comparer les deux approches, et de déterminer leurs inconvénients et/ou avantages respectifs, les métriques retenues sont la durée de synthèse/compilation et le temps de réponse du programme créé pour Heptagon/BZR, et juste le temps de réponse pour l'algorithme d'ordonnancement. Comme nous nous sommes doutés que l'algorithme issu de la synthèse de contrôleur serait plus rapide que celui utilisant des techniques d'ordonnancement, nous avons souhaité à la fois comparer ces temps de réponse, mais également calculer combien de cycles sont nécessaires pour "rentabiliser" le temps de synthèse. Une autre métrique, moins bien définie, correspond aux limites de l'approche par synthèse de contrôleurs. Nous pouvons en effet nous attendre à ce qu'elle arrête de fonctionner à partir d'un nombre de tâches ou de ressources pourtant encore faible, tandis que les techniques d'ordonnancement devraient pouvoir traiter des problèmes de tailles beaucoup plus importantes.

1.4 Aperçu des résultats

Les résultats sont en majeure partie similaires à nos attentes : le programme généré par Heptagon/BZR est plus rapide que les techniques d'ordonnancement concurrentes exactes à base de programmation linéaire, mais la synthèse elle-même est extrêmement longue, et ne termine assez rapidement plus du tout. De plus, il s'avère en fait que cet écart de performance sur des cas très simples n'a presque aucune valeur, puisqu'un algorithme naïf énumérant les différentes configurations jusqu'à trouver les bonnes est bien plus rapide que le temps de compilation d'Heptagon, tout en permettant un temps de réponse en ligne similaire. Du côté de l'ordonnancement, il semble intéressant d'utiliser les techniques d'ordonnancement de liste en priorité, et de n'utiliser les algorithmes exacts qu'en absence de résultat positif. Des pistes d'amélioration des performances du compilateur Heptagon ont été trouvées, permettant de générer du code plus efficace pour les contrôleurs.

— 2 —
État de l'art

2.1 Synthèse de contrôleur

2.1.1 Le langage Heptagon

Heptagon [17] est un langage synchrone utilisant des flux de données. Ce langage est un dérivé de Lustre [2], et partage avec ce dernier la majeure partie de sa grammaire. Son compilateur, également nommé Heptagon, permet de générer du code C ou Java.

Pour donner un exemple introductif recouvrant un maximum de concepts, voici un nœud recevant en entrée un booléen entree1 et un entier entree2 , et retournant un entier en sortie. Si entree1 est vrai, la valeur de entree2 est retournée. Sinon, la précédente valeur de sortie (ou l'actuelle pour le premier pas) est retournée.

```
1 node noeud(entree1:bool; entree2:int) returns (sortie:int)
2 var local_var : int;
3 let
4 sortie = if (entree1) then entree2 else local_var;
5 local_var = entree -> pre sortie;
6 tel
```

"if" est ici un opérateur ternaire strict, évaluant de toutes façons ses trois opérandes.

"pre" est un opérateur permettant d'accéder à la valeur que son opérande avait au pas précédent.

Le mot-clef var permet de déclarer des variables locales. Ici, une variable entière local_var est déclarée (même si elle est superflue ici, nous pourrions la remplacer lorsqu'elle est utilisée par sa définition 0 **->pre** sortie).

La principale fonctionnalité spécifique à Heptagon qui a été utilisée pendant ce stage est la possibilité de définir des contrats. Par exemple, si l'utilisateur du programme précédent ne souhaite pas définir lui-même la valeur de la consigne booléenne entree1, mais préférerait plutôt que la valeur maximale rencontrée depuis le début de l'exécution soit gardée automatiquement en sachant que la valeur de l'entrée est toujours positive, le programme peut être réécrit de la façon suivante :

```
1 node noeud(entree:int) returns (sortie:int)
2 contract
3     assume entree >=0
4     enforce sortie >= (0 -> pre sortie)
5     with controllable: bool
```

```
6  var local_var : int;
7  let
8  sortie = if (controllable) then entree else local_var;
9  local_var = entree -> pre sortie;
10 tel
```

La partie "assume" permet de définir les hypothèses de fonctionnement du programme. Ici, l'hypothèse retenue est que l'entrée est toujours positive.

Le champ "enforce" est suivi d'une expression booléenne, qui doit impérativement être vraie lorsque les hypothèses évoquées précédemment sont vérifiées. Ici, la valeur en sortie se doit d'être à chaque pas supérieure ou égale à la valeur précédente et la première valeur prise est nécessairement supérieure à 0.

Le champ "with" déclare de nouvelles variables, dites "contrôlables", qui seront fixées par le contrôleur à l'exécution pour faire en sorte que le contrat soit respecté. Elles ne peuvent être que de type booléen ou énuméré. Ici, une variable contrôlable booléenne est utilisée.

Lors de la compilation, si des contrats sont présents, Heptagon génère du code à destination de l'outil ReaX, qui permet la synthèse de contrôleurs discrets. Le code ainsi généré est alors transformé en programme Heptagon, compilé à son tour, et assemblé avec le code du nœud.

En ajoutant que le langage Heptagon permet la définition et l'utilisation de variables de type tableaux, nous aurons fait le tour des spécificités utilisées durant ce stage, et pouvons donc désormais décrire l'implémentation d'un ordonnanceur en Heptagon. Intéressons-nous à ce qui a déjà été fait dans ce sens lors de recherches précédentes.

2.1.2 Recherches précédentes

Ce sujet de stage se place dans la continuité des travaux réalisés notamment par Gwenaël Delaval, l'un des chercheurs encadrant ce stage, sur la reconfiguration d'un système industriel subissant une attaque [7], ainsi que sur la création d'un DSL permettant la reconfiguration automatique d'un FPGA [11].

Dans l'article traitant de cybersécurité [7], le but est, lorsqu'une des machines est attaquée, de transférer ses tâches en cours d'exécution à d'autres machines non compromises de manière à maximiser le nombre de programmes restant dans un état nominal. Dans ce contexte, d'autres éléments de fonctionnement de ce type de système sont émulés, tels que la possibilité de passer un programme en mode dégradé, ce qui réduit le temps nécessaire pour l'exécuter, ou bien encore la notion de section critique empêchant un programme de changer de machine. Ces notions ne se retrouvent pas dans la formalisation actuelle, pour gagner en généralité. L'article concluait en tout cas qu'il était possible de créer un tel contrôleur à l'aide d'Heptagon, mais uniquement pour de petits systèmes, soit 7 machines ou moins. La durée exponentielle de la synthèse empêche d'appliquer cette méthode à de plus gros problèmes.

Dans le second article, un DSL (Domain Specific Language) basé sur Heptagon est proposé [11]. Il permet, à l'aide d'une description des différentes ressources en présence, des différentes tâches à exécuter et d'invariants de décrire le fonctionnement attendu d'un système exécuté sur une FPGA (Field Programmable Gate Array). Cette description, écrite donc à l'aide du DSL, permet la génération automatique d'un contrôleur à l'aide d'Heptagon et de ReaX. Au-delà de se faire une idée du type de problèmes "d'ordonnancement" traitables à l'aide d'Heptagon, les cas exhibés dans cet article sont assez éloignés de ce qui a été étudié pendant ce stage.

2.2 Ordonnancement et optimisation sous contraintes

En premier lieu, pour pouvoir déterminer où notre problème se situe parmi les problèmes d'ordonnancement, il a été nécessaire de le placer dans leur classification, qui est notamment décrite dans cet article [10]. L'objectif était de trouver une notation de Graham $\alpha|\beta|\gamma$ se rapprochant le plus possible des situations qui nous intéressent. Le sens de ces notations est rappelé en annexe 1. Comme nous avons souhaité que la durée d'exécution des tâches varie en fonction des machines, nous nous sommes placés dans le cadre des machines parallèles sans lien entre elles, soit $\alpha = R$. Ensuite, nous désirions poser une deadline globale, qui correspond à la fin d'un cycle du système. Cet élément n'a finalement pas été intégré à la description du problème d'optimisation. En effet, ce qui nous intéresse n'est pas d'optimiser une valeur en lien avec l'ordonnancement obtenu, mais bien uniquement de s'assurer de la faisabilité de celui-ci. Notre cas peut donc être décrit comme un problème de décision associé à $R||C_{max}$, le problème d'optimisation dont l'objectif est de minimiser la date de fin d'exécution. Nous souhaitons ici savoir s'il est possible ou non de trouver un ordonnancement ayant une durée inférieure à 1, soit un cycle, plutôt que de minimiser cette valeur.

Pour étudier les résultats de complexité et les algorithmes d'approximation disponibles, un site nommé "The scheduling Zoo" [5] a été utilisé. La mauvaise nouvelle est que $P||C_{max}$ est fortement NP difficile [9], donc notre cas l'est également : il est impossible de trouver un algorithme polynomial pour répondre à notre problème (sauf si $P = NP$). Étant donné que le problème de décision étudié s'est révélé être fortement NP difficile également, ce qui sera abordé lors de la description formelle du problème traité, ce résultat n'est pas surprenant. Il existe cependant un algorithme polynomial qui est une 2-approximation duale [12], c'est-à-dire qu'en recevant en consigne un certain temps d'exécution, il renverra un ordonnancement dont la durée est au pire deux fois plus longue que la consigne, ou bien signalera qu'il est impossible de trouver un ordonnancement respectant la consigne. À noter que cet algorithme n'est pas capable de répondre au problème de décision "y a-t-il un ordonnancement de longueur totale en dessous d'un seuil donné" qui nous intéresse. Malheureusement, ce type d'approximation n'est pas vraiment utilisable seul dans le cadre d'une comparaison avec les performances du contrôleur généré par Heptagon, qui est supposé donner toujours un résultat exact si cela est possible. Nous avons ainsi dû utiliser des méthodes exactes, et donc exponentielles en pire cas.

Les approches utilisant des métaheuristiques telles que les algorithmes génétiques ont été étudiées, mais pas retenues. En effet, ces heuristiques ne possèdent aucunes garanties de temps d'exécution, et les premiers résultats montraient une grande variabilité de celui-ci. Les méthodes exactes finalement utilisées sont les programmes linéaires en nombres entiers et la programmation par contraintes, puisque le problème se décrit de manière intuitive dans ces formalisations.

2.2.1 Programmation linéaire

La programmation linéaire, également appelée optimisation linéaire est une des techniques fréquemment utilisées pour réaliser des ordonnancements. Le but est d'assigner des valeurs à un certain nombre n de variables définies dans $\mathbb{R}$, de manière à respecter des contraintes linéaires et de minimiser ou maximiser une fonction linéaire. Des algorithmes de complexité polynomiales existent pour résoudre ce type de problème lorsque les variables sont des réels,

par exemple la méthode des points intérieurs [14]. Cependant, ces variables représenteront dans notre cas des attributions tâche-machine, il s'agit donc plutôt de valeurs booléennes. Dans les cas où les variables sont booléennes (ou entières), le problème est NP difficile, et les solveurs utilisent donc des heuristiques pour rester suffisamment rapide.

2.2.2 Programmation par contraintes

Les problèmes traités par la programmation par contraintes sont similaires à ceux que la programmation linéaire peut résoudre. La principale différence est l'absence de fonction objectif à minimiser ou maximiser, ce qui signifie que cette méthode traite de problème de décision, et pas d'optimisation. Comme ce qui nous intéresse est l'existence d'un ordonnancement, cela nous évite une transformation du problème à résoudre. De plus, les variables peuvent appartenir à des domaines ad hoc potentiellement différents pour chacune d'entre elles, et les conditions ne sont plus nécessairement linéaires. Résoudre ce type de problème est une fois de plus NP difficile, mais les heuristiques utilisées sont différentes.

Maintenant que les techniques qui vont être utilisées sont introduites, nous pouvons désormais décrire de manière formelle les problèmes qui nous intéressent, et les implémentations associées en utilisant d'une part la suite Heptagon, et d'autre parts des outils issus des sciences de l'ordonnancement.

— 3 —

Description du problème et implémentations dans les différents formalismes

3.1 Description formelle de la classe de problèmes traités

La seconde tâche à réaliser en débutant le projet, après avoir réalisé l'état de l'art, était logiquement de définir précisément la classe de problèmes qui nous intéresse. De manière informelle, nous rappelons qu'il nous faut décrire des problèmes comportant un certain nombre de machines et de tâches, s'exécutant par cycles. À chaque tâche est associé un temps d'exécution potentiellement différent pour chaque machine, exprimé en proportion de la durée d'un cycle. Il est supposé que les tâches peuvent ou non être actives pendant un cycle donné, et de façon analogue les ressources peuvent également devenir indisponibles. Ces perturbations sont gérées directement par l'environnement, et donc pas par le contrôleur. Nous posons également comme hypothèse qu'au moins deux ressources sont toujours disponibles à tout moment.

3.1.1 Modèle proposé

Paramètres en entrée du problème :

- Un ensemble de n (entier strictement positif) tâches T_i, $1 \leq i \leq n$.
- Un ensemble de m (entier supérieur ou égale à 2) ressources R_j, $1 \leq j \leq m$.
- Pour $1 \leq i \leq n$ et $1 \leq j \leq m$, des réels positifs $p_{i,j}$ correspondant à la proportion de la durée d'un cycle nécessaire pour que T_i s'exécute sur R_j. Usuellement, $0 < p_{i,j} \leq 1$.

  Si $p_{i,j} > 1$, cela signifie que la tâche T_i n'est pas exécutable sur la ressource R_j.

Paramètres dynamiques du problème :

- Active, un tableau de booléens de taille n, tel que $Active[i] = true$ indique que la tâche T_i est active, et $Active[i] = false$ indique que la tâche T_i est inactive, pour $1 \leq i \leq n$.
- Available, un tableau de booléens de taille p, tel que $Available[j] = true$ indique que la ressource R_j est disponible, et $Available[j] = false$ indique que la ressource R_j n'est pas disponible, pour $1 \leq j \leq m$. À tout moment, $\exists i, i'$ tel que $1 \leq i < i' \leq m$ et $Available[i] = Available[i'] = true$.

Le format utilisé pour décrire les problèmes au format textuel est décrit en annexe 2.

3.1.2 Forme d'une solution

À chaque changement dans les variables dynamiques du problème, la sortie attendue est un vecteur "Attribution" de taille n, tel que pour tout $1 \leq i \leq m$, l'élément à l'emplacement i dans le vecteur est contenu entre 1 et m. $Attribution[i] = j$ représente l'attribution de la tâche i à la ressource j.

3.1.3 Appartenance à la classe NP

Avant de rechercher des algorithmes spécifiques et de mesurer les performances effectives des différentes approches, il est intéressant de connaître la classe de complexité à laquelle appartiennent les problèmes qui nous intéressent. C'est dans notre situation assez simple :

Nous pouvons reformuler notre problème de la façon suivante : nous possédons m processeurs de vitesses variables, pour lesquels nous souhaitons trouver un ordonnancement non préemptif de n tâches, permettant de ne pas dépasser une deadline globale (de 1 ici). Il s'agit donc d'une version analogue au problème nommé "MULTIPROCESSOR SCHEDULING" dans le livre "Computers and intractability. A guide to the theory of NP-completeness" [16]. La seule différence réside dans la possibilité de donner différents temps d'exécution aux tâches en fonction des ressources qui les exécutent. Notre cas étant plus général, le problème décrit dans le livre s'y réduit. Comme il est fortement NP difficile lorsque le nombre de machines disponibles est arbitraire, les problèmes que nous souhaitons résoudre appartiennent également à cette classe. Il est de plus linéaire en le nombre de tâches de vérifier si une solution est bonne, nous pouvons donc affirmer que notre problème se situe bien dans NP.

Nous ne pourrons donc pas trouver d'algorithmes polynomiaux, ni même de schéma d'approximation entièrement polynomial, et nous pouvons donc nous attendre à ce que les méthodes utilisées explosent en temps demandé lorsque le nombre de tâches ou de machines croît.

3.2 Traduction de la classe de problèmes en Heptagon

3.2.1 Définition du programme principal

Nous nous inspirons fortement des travaux effectués dans la publication portant sur la cybersécurité [7], puisque notre modèle est une version simplifiée des problèmes qui y sont traités.

Avant toute chose, un type correspondant à l'énumération des ressources est défini.

```
type resource = R_1 | R_2 | ... | R_m | None
```

La signature du nœud principal du système est déclarée de la façon suivante :

```
node main(active_1, active_2, ..., active_n,
available_1, available_2, ..., available_m:bool)
returns (loc_1,loc_2, ..., loc_n : resource;
         objective:bool; nbRes: int)
```

Où active_i est vrai ssi la tâche correspondante est active, et available_j est vrai ssi la ressource correspondante est disponible. Les loc_i utilisent le type ressource défini plus haut, et loc_i = R_j ssi le contrôleur exécute la tâche T_i sur la ressource R_j. Si loc_i = None, alors la tâche n'est exécutée sur aucune ressource. Objective est un booléen qui sera toujours à vrai (utilisé par la synthèse de contrôleur), et nbRes est une variable comptant le nombre de ressources disponibles, et est également présente pour aider à la synthèse de contrôleurs.

Ensuite, les nœuds correspondant aux tâches sont déclarés comme suit :

```
node task<<c_1, c_2, ..., c_m : float>>
(consigne : resource)
returns (loc:resource; occ_1, occ_2, ..., occ_m: float;)
```

Où les c_j sont des paramètres statiques, correspondant au coût en proportion d'utilisation de la ressource par la tâche si elle est allouée à la ressource R_j et la valeur c_j de la tâche i est donc identique au temps d'exécution p_i,j, consigne est une variable manipulée par le contrôleur pour indiquer la ressource à laquelle la tâche est associée, loc est une Ressource valant R_j si la tâche est allouée à la ressource R_j, et 0 si elle n'est exécutée sur aucune ressource.

Les occ_j indiquent la quantité de capacité utilisée par la tâche sur la ressource R_j pendant le pas actuel. Typiquement, cela vaudra 0 sauf si loc = R_j, auquel cas cela vaudra c_j.

Le corps des nœuds représentant les tâches, et implémentant le fonctionnement décrit ci-dessus est :

```
let
    loc = consigne;
    occ_i = if loc = R_i then c_i else 0.0;
tel
```

Nous pouvons désormais définir le corps du nœud principal : il consiste principalement en une instanciation du nœud "task" pour chacune des tâches du problème, avec les paramètres qui lui correspondent :

```
var occ_1_1, occ_2_1, ..., occ_n_1,
    occ_1_2, occ_2_2, ..., occ_n_2,
    ...
    occ_1_m, occ2_m, ... occ_n_m : float;
    occ_tot_1, occ_tot_2, ..., occ_tot_m: float;
let
```

```
    (loc_1, occ_1_1, occ_1_2, ..., occ_1_m)
    = inlined task<<c_1_1, c_1_2, ..., c_1_m>>(consigne_1);
    (loc_2, occ_2_1, occ_2_2, ..., occ_2_m)
    = inlined task<<c_2_1, c_2_2, ..., c_2_m>>(consigne_2);
    ...
    (loc_n, occ_n_1, occ_n_2, ..., occ_n_m)
    = inlined task<<c_n_1, c_n_2, ..., c_1_m>>(consigne_n);
    occ_tot_1 = occ_1_1 +. occ_2_1 +. ... +. occ_n_1;
    occ_tot_2 = occ_1_2 +. occ_2_2 +. ... +. occ_n_2;
    ...
    occ_tot_m = occ_1_m +. occ_2_m +. ... +. occ_n_m;
    objective =
        (active_1 or loc_1 = none) and (not active_1 or loc_1 <> none )
        and
        (active_2 or loc_2 = none) and (not active_2 or loc_2 <> none )
        and
        ...
        (active_n or loc_n = none) and (not active_n or loc_n <> none )
        and
        (available_1 or occ_tot_1 = 0) and (not available_1 or occ_tot_1 <= 1.0)
        and
        (available_2 or occ_tot_2 = 0) and (not available_2 or occ_tot_2 <= 1.0)
        and
        ...
        (available_m or occ_tot_m = 0) and (not available_m or occ_tot_m <= 1.0);
    nbRes =
        (if available_1 then 1 else 0) +
        (if available_2 then 1 else 0) +
        ... +
        (if available_m then 1 else 0);


tel
```

Où les c_i_j correspondent aux $p_{i,j}$ de la formalisation du problème et les occ_i_j correspondent à la proportion du cycle de la machine R_j actuellement utilisée par la tâche T_i.

Les équations correspondant aux variables locales "occ_tot_j", à "objective" et à nbRes sont liées au contrat : il faut faire en sorte que la variable "objective" soit toujours vraie, et il est supposé que nbRes sera toujours supérieure ou égale à 2. La première partie de "objective" oblige les tâches actives à être affectées à une ressource, et les tâches inactives à n'être associées à aucune ressource.

La seconde partie empêche les ressources inactives d'exécuter des tâches (leurs capacités ne peuvent pas être utilisées), et empêche les ressources actives de dépasser leurs capacités (en proportion d'utilisation).

Les tâches sont "inlinées" pour que la synthèse de contrôleur ait accès à leurs corps, puisque le nœud task ne possède pas de contrat.

Afin de s'assurer que les propriétés sont bien respectées, et pour définir les variables consignes_i, nous allons maintenant décrire le contrat associé au nœud principal :

```
contract
    assume nbRes >=2
    enforce objective

    with    (consigne_1, consigne_2, ..., consigne_n:ressource)
```

Ici, le but du contrat est uniquement de forcer la variable "objective" à true. L'ordre des variables contrôlées a de l'importance : la synthèse tente de donner une valeur "true" aux variables booléennes, ou la première valeur de l'énumération pour les types énumérés (dans notre cas, R_1). Si elle n'y arrive pas, elle commence par modifier les valeurs des dernières variables

déclarées. Dans notre cas, cela signifie que le contrôleur aura tendance à exécuter les tâches sur les premières ressources disponibles (dans l'ordre numérique, R_i avant R_{i+1}).

3.2.2 Une exécution trop rapide qui nécessite une répétition des pas

La durée d'un pas du programme ci-dessus est très faible pour les tailles de problèmes que nous pouvons traiter, si faible qu'il n'est pas vraiment possible de la mesurer seule. Nous devons donc exécuter un grand nombre de pas identiques pour mesurer le temps d'exécution effectif. En pratique, des tableaux sont utilisés ici pour exécuter en "parallèle" (avec les mêmes entrées, mais exécuté de façon séquentielle) nbsim (constante entière supérieure ou égale à 1) versions identiques du nœud principal. Cette constante nbsim vaut dans la version actuelle du programme de test 50000. Lors de la traduction du problème en Heptagon, nous produisons donc un second fichier .ept, permettant cette exécution multiple. La création de ce second fichier est décrite en annexe 4.

3.2.3 Problèmes rencontrés avec l'outil ReaX

Dans son état actuel, l'outil utilisé pour réaliser la synthèse de contrôleurs, ReaX, à posé quelques problèmes lors de l'implémentation. En effet, celui-ci ne se comporte pas comme il le devrait dans certains cas.

Premièrement, des essais ont été réalisés en début de stage, utilisant une fonctionnalité de ReaX permettant de définir des variables à maximiser/minimiser. Cet ajout d'un objectif d'optimisation aurait permis un rapprochement plus poussé avec les problèmes traités par les techniques d'ordonnancement. Cependant, cette optimisation ne semblait au final pas fonctionner, et l'objectif était complètement ignoré.

Ensuite, dans la version actuelle des programmes Heptagon générés, ReaX renvoie des faux positifs. En effet, il semble que la synthèse n'échoue pas même lorsqu'une configuration avec au moins deux machines disponibles ne possède aucun ordonnancement satisfaisant. Pour remédier à ce problème, la faisabilité des problèmes est calculée en amont par des algorithmes naïfs énumérant les possibilités, ce qui n'est possible que car les problèmes traités sont de petite taille.

3.3 Utilisation de techniques d'ordonnancement pour résoudre les problèmes

3.3.1 Traduction en programme linéaire

Le programme linéaire peut s'écrire de la manière suivante :

$$
\begin{aligned}
\text{minimize} \quad & C_{max} \\
\text{subject to} \quad & \sum_{j \in [[1,m]]} x_{ij} = 1, \quad i = 1, \ldots, n, \\
& \sum_{i \in [[1,n]]} x_{ij} p_{ij} \leq 1, \quad j = 1, \ldots, m, \\
& \sum_{i \in [[1,n]]} x_{ij} p_{ij} \leq C_{max}, \quad j = 1, \ldots, m, \\
& x_{ij} \quad \in \{0,1\}, \quad i = 1, \ldots, n \quad j = 1, \ldots, m, \\
& C_{max} \geq 0
\end{aligned}
$$

Les deux dernières contraintes concernent les valeurs que les variables peuvent prendre. Les variables d'attributions x_{ij} sont des booléens, et l'objectif C_{max} est un réel supérieur à 0. Il serait possible de restreindre davantage C_{max}, et le définir comme supérieur ou égal à 1. Cette contrainte permettrait de répondre directement à notre problème de décision plutôt qu'au problème d'optimisation. Une variable x_{ij} est à vrai si et seulement si la tâche i est associée à la ressource j dans l'ordonnancement proposé. La première contrainte nous assure que pour une tâche i donnée, une seule ressource j est associée.

La seconde contrainte garantit que pour chaque ressource, la somme des durées d'exécution des tâches qui lui sont associées ne dépasse pas 1, soit la durée du cycle. Enfin, la troisième contrainte est en fait la définition de C_{max} : ce nombre doit être plus grand que toutes les durées d'exécution des différentes machines.

Ce programme est implémenté en Python à l'aide du module PULP [18]. Ce dernier permet de décrire ce type de problème de manière intuitive, et possède des interfaces avec un grand nombre de solveurs de programmes linéaires.

3.3.2 Traduction en programmation par contraintes

La représentation du programme par contrainte est identique à celle du programme linéaire, mais sans objectif de minimisation. La suite utilisée pour l'implémentation est OR-Tools [19], et plus précisément le module python associé. Cette implémentation diffère légèrement du problème décrit cependant, puisque OR-Tools n'est pas compatible avec le calcul flottant. Toutes les durées d'exécution sont multipliées par un grand entier K et tronquées, et l'objectif est changé en conséquence.

3.3.3 Heuristiques et approximations utilisées

2-Approximation utilisant un programme linéaire

Nous utilisons l'algorithme notamment décrit dans cette publication [15], qui consiste à relaxer le problème linéaire initial. Les variables attribuant des tâches à une machine, plutôt que de

rester dans le domaine booléen, peuvent alors prendre des valeurs situées entre 0 et 1. Une fois une solution à ce problème relaxé obtenue, celle-ci est transformée pour devenir une solution réalisable, sans attribution fractionnaire. La méthode de transformation utilisée ici garantit, pour une consigne donnée, d'obtenir une solution dont le temps d'exécution est au pire deux fois cette consigne. Si nous ne trouvons pas de solution au problème relaxé ayant un temps d'exécution en dessous de cette consigne, nous pouvons être certains qu'il n'en existe pas pour le problème original non plus.

Heuristiques à base de liste

Nous employons également des heuristiques d'ordonnancement de liste décrites dans cet article [6], plus précisément les techniques MET, MCT et HLPT. Tous ces algorithmes fonctionnent de manière analogue, recevant une liste de tâches et leur attribuant des machines de manière gloutonne. La différence se situe dans le critère glouton retenu.

La première d'entre elles, MET, signifie Minimum Execution Time. Elle attribue simplement les tâches à la machine qui les exécutera le plus rapidement.

La seconde, MCT qui signifie Minimum Completion Time, est déjà un peu plus évoluée. Les tâches sont prises dans un ordre quelconque, et placées sur les machines qui termineront leurs exécutions le plus tôt possible, en prenant en compte les temps d'exécution des autres tâches déjà attribuées.

Enfin, HLPT qui signifie Heterogeneous Largest Processing Time, se comporte exactement comme MCT, mais débute par un tri préalable des tâches. Le tri ici est réalisé par ordre décroissant de temps d'exécution moyen. Ces heuristiques ne présentent aucune garantie sur la qualité des solutions obtenues lorsque les machines sont indépendantes, ce qui est notre cas.

3.4 Comparaison des différentes techniques d'ordonnancement

3.4.1 Choix des solveurs de problèmes linéaires

Dans un souci de reproductibilité et d'accessibilité des résultats, nous nous sommes restreint à l'utilisation de solveurs gratuits et open source. Nous n'utiliserons donc pas ici de logiciels tels que Cplex ou Gurobi, qui pourraient potentiellement s'avérer plus rapide. Sur les exemples qui nous intéressent, c'est-à-dire $2 \leq n \leq 10$ et $2 \leq m \leq 5$, nous avons comparé les performances moyennes de trois solveurs de problèmes linéaires, qui semblaient avoir les meilleures performances parmi les logiciels open source, et d'un solveur de programmation par contraintes. Les trois solveurs de programme linéaires sont Solving Constraint Integer Programs (scip) [3] [4], COIN-OR Branch and cut (cbc) [8] et high performance software for linear optimization (HiGHS) [13]. Le solveur utilisé pour la programmation par contraintes est cp-sat, qui est intégré à la suite OR-Tools.

Nous avons réalisé un premier graphique comparant les performances de ces 4 méthodes, illustrant à nombre de machines fixé l'évolution du temps moyen nécessaire à chacun de ces algorithmes pour réaliser l'ordonnancement avec des barres d'erreur montrant l'intervalle de confiance à 95% (figure 3.1).

Les exemples générés pour cette comparaison suivent la loi suivante : Pour un problème donné avec n tâches et m machines, tous les temps d'exécution sont des variables aléatoires sui-

vant une même loi normale. L'espérance μ de cette loi normale est la fraction pouvant s'écrire $\frac{1}{i}$ avec $i \in \mathbb{N}^*$ la plus grande qui est inférieure ou égale à m/n. La variance σ^2 vaut un dixième de l'espérance. Pour chaque combinaison unique (n,m), les algorithmes ont été testés sur 150 exemples aléatoires (si un exemple n'avait pas de solution, un autre était généré à sa place).

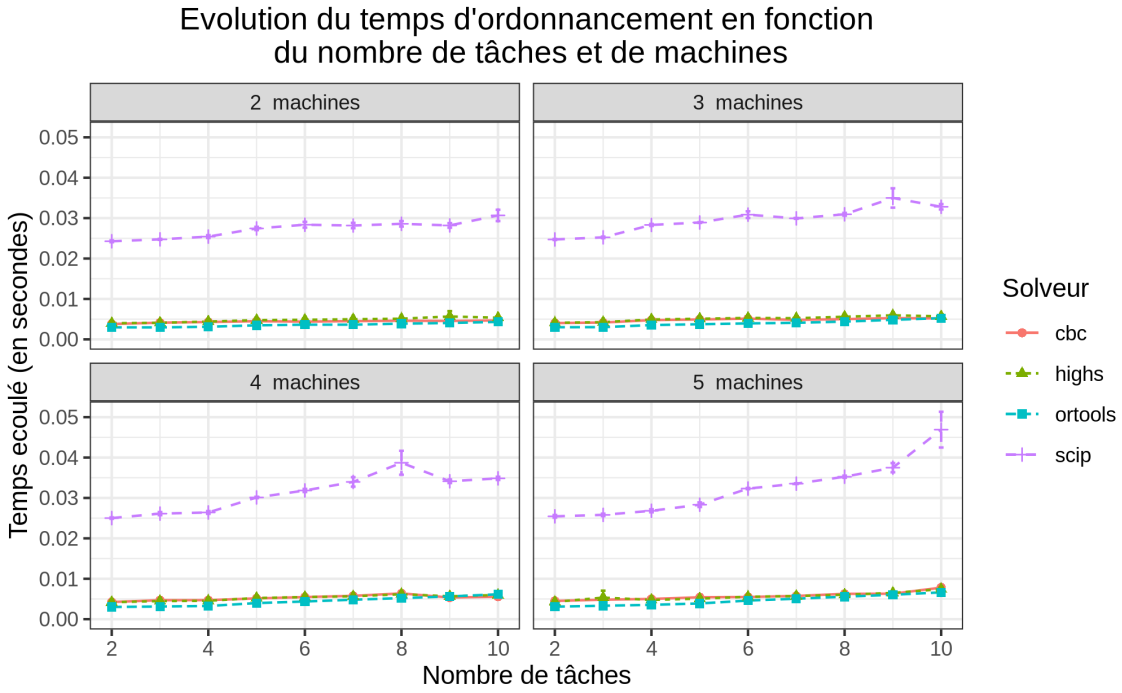

FIGURE 3.1 : Comparaison de quatre algorithmes d'ordonnancement

Ce graphique n'apporte en réalité qu'une information : sur les cas qui nous intéressent, le solveur scip présente de très mauvaises performances par rapport à sa concurrence. Nous avons donc refait la même illustration, en retirant ce logiciel des candidats (figure 3.2).

Cette fois-ci, nous pouvons observer que, pour les problèmes qui nous intéressent, l'approche à base de programmation par contraintes semble être en moyenne plus rapide. Pour ce qui est de la programmation linéaire, cbc semble présenter des performances moyennes légèrement meilleures et moins instables que celles de highs. Pour garder de la diversité dans les approches, nous avons donc gardé donc l'algorithme se basant sur cbc et celui se basant sur OR-Tools pour les comparer aux programmes générés par Heptagon/BZR. Ces algorithmes sont retenus notamment car, en se basant sur des essais, nous avons remarqué que l'approche de la programmation par contraintes semble plus efficace lorsque le nombre de machines et de tâches reste bas, mais devient très vite moins efficace par la suite. Il s'agit seulement du meilleur concurrent à Heptagon sur les cas que ce langage peut traiter parmi les algorithmes exacts évoqués ici.

3.4.2 Comparaison des performances sur des cas plus difficiles

Observons désormais les performances de ces trois algorithmes sur des cas plus difficiles (plus de tâches, de machines et en moyenne le temps d'exécution moyen des tâches est proche de m/n et pas de $2/n$). Une échelle logarithmique est utilisée pour l'abscisse et l'ordonnée, le

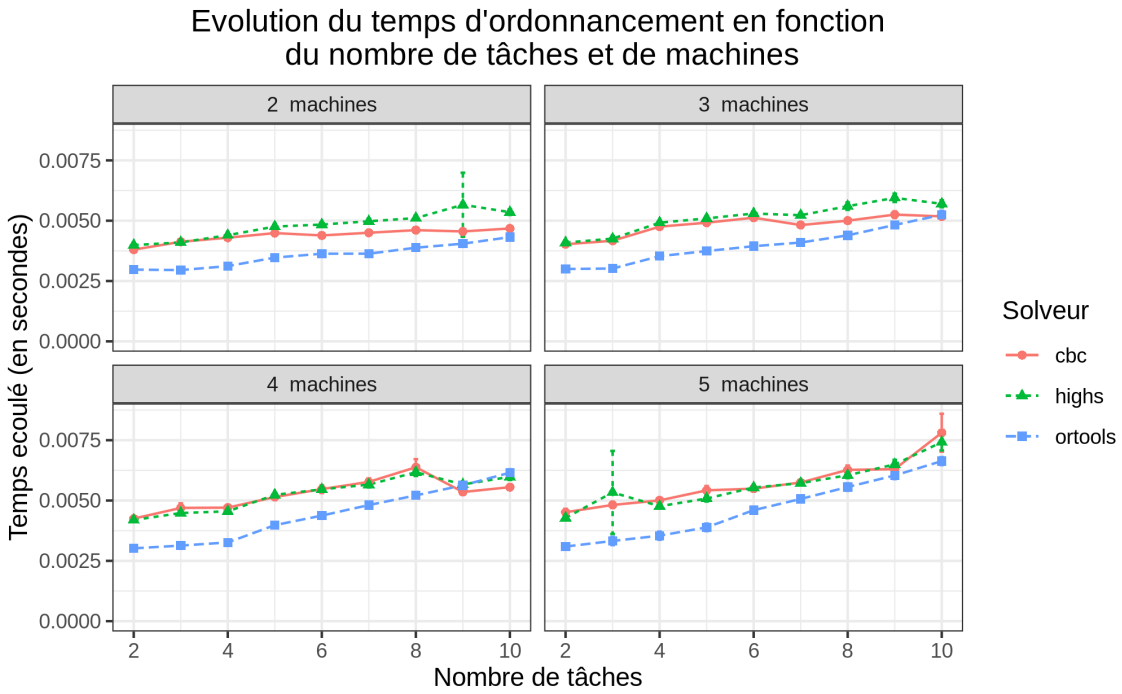

FIGURE 3.2 : Comparaison des trois algorithmes d'ordonnancement restant

nombre de tâches va de 2 à 2048 et le nombre de machines de 2 à 64, le tout par paliers de puissance de 2 (figure 3.3).

Il est remarquable que, pour ce type de problème, il n'est pas facile d'affirmer la supériorité générale d'une technique par rapport à une autre. Les deux solveurs de programmes linéaires ne semblent pas avoir un avantage clair l'un sur l'autre, même si highs donne des résultats un peu plus erratiques sur les cas les plus difficiles. La programmation par contraintes semble avoir l'avantage lorsque le nombre de machines est élevé ou que le nombre de tâches est bas.

Nous pouvons également observer que, de façon générale, les courbes des algorithmes ne suivent pas une trajectoire régulière. La raison est assez simple : ces algorithmes sont exponentiels en pire cas, et n'obtiennent d'aussi bons résultats que grâce à des heuristiques et autres optimisations, dont les résultats ne sont pas garantis. Le principal but de ce graphique est cependant de montrer que ces divers algorithmes d'ordonnancement continuent de fonctionner bien après que les quelques cas traitables par Heptagon. De plus, l'allure de ces courbes n'étant pas exponentielle, nous pourrions sans doute espérer encore obtenir des ordonnancements en augmentant encore davantage le nombre de tâches et de machines.

3.4.3 Pertinence des algorithmes d'approximation

Cas de la programmation linéaire avec arrondi

Cette fois-ci, nous observons l'évolution des performances comparées de la 2-approximation et de l'algorithme exacts, tous les deux utilisant le solveur Highs. Les problèmes ici sont plus simples et générés de manière à ce que l'approximation donne toujours un résultat correct, c'est à dire qu'il existe une configuration avec un temps total < 0.5. Une fois de plus, des échelles logarithmiques sont utilisées aux abscisses et aux ordonnées (figure 3.4).

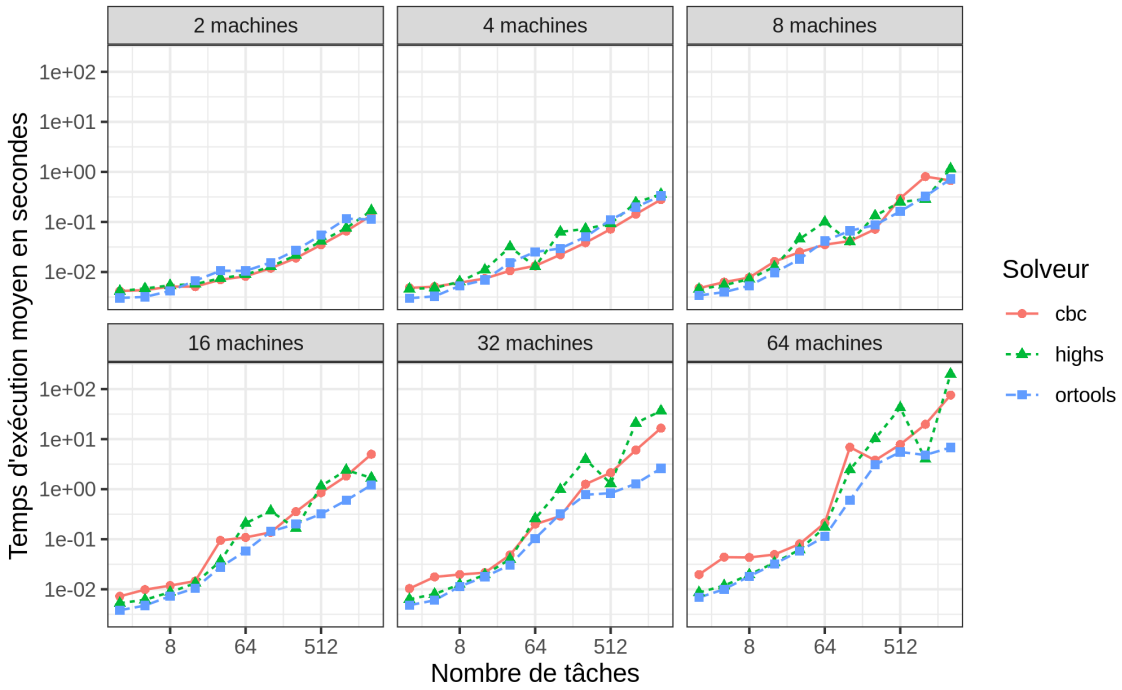

FIGURE 3.3 : Comparaison des trois algorithmes d'ordonnancement pour $n > 8$ sur un repère log/log

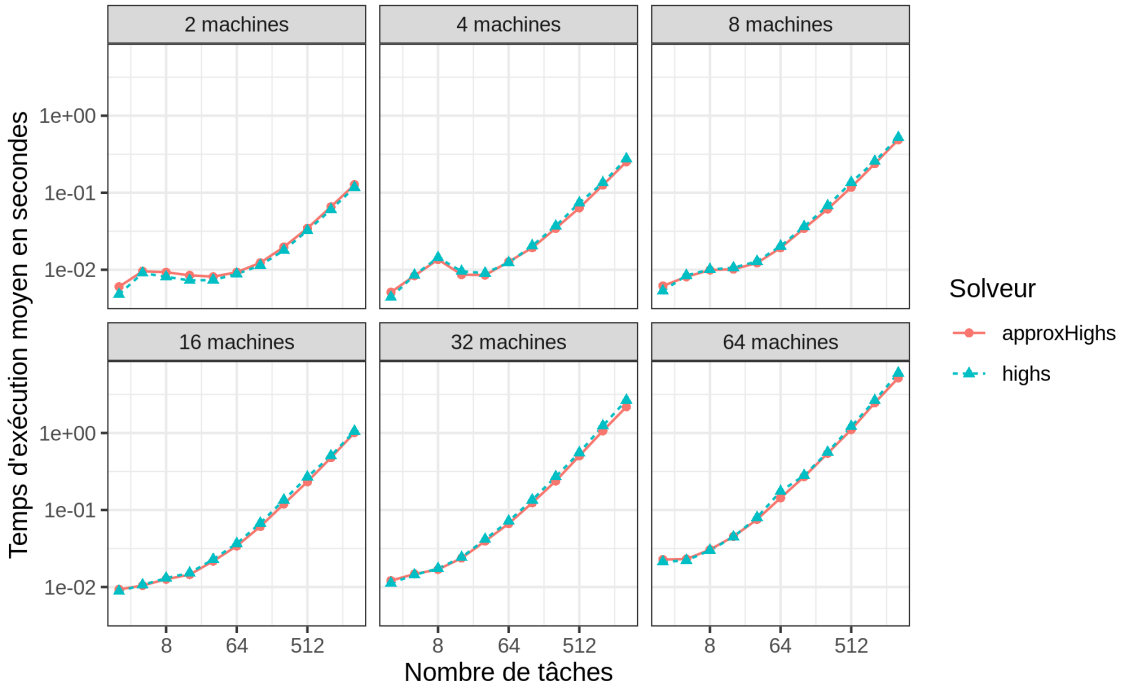

FIGURE 3.4 : Comparaison des algorithmes utilisant Highs (approximation et exact) sur un repère log/log

Le gain de performance ne semble pas évident, puisque les courbes de la méthode exacte et de la 2-approximation se chevauchent. Il ne semble donc pas intéressant de retenir cette méthode. Au vu de la proximité des résultats, il est probable que les solveurs de programmes linéaires entiers utilisent déjà des relaxations du problème original prenant la forme de programmes linéaires réels.

Cas des heuristiques utilisant l'ordonnancement de liste

Nous nous intéressons désormais à l'efficacité de l'ordonnancement de liste. Regardons en premier lieu les performances en temps des trois algorithmes sur des cas faciles (traitables par la 2-approximation, figure 3.5) et sur des cas difficiles (figure 3.6) :

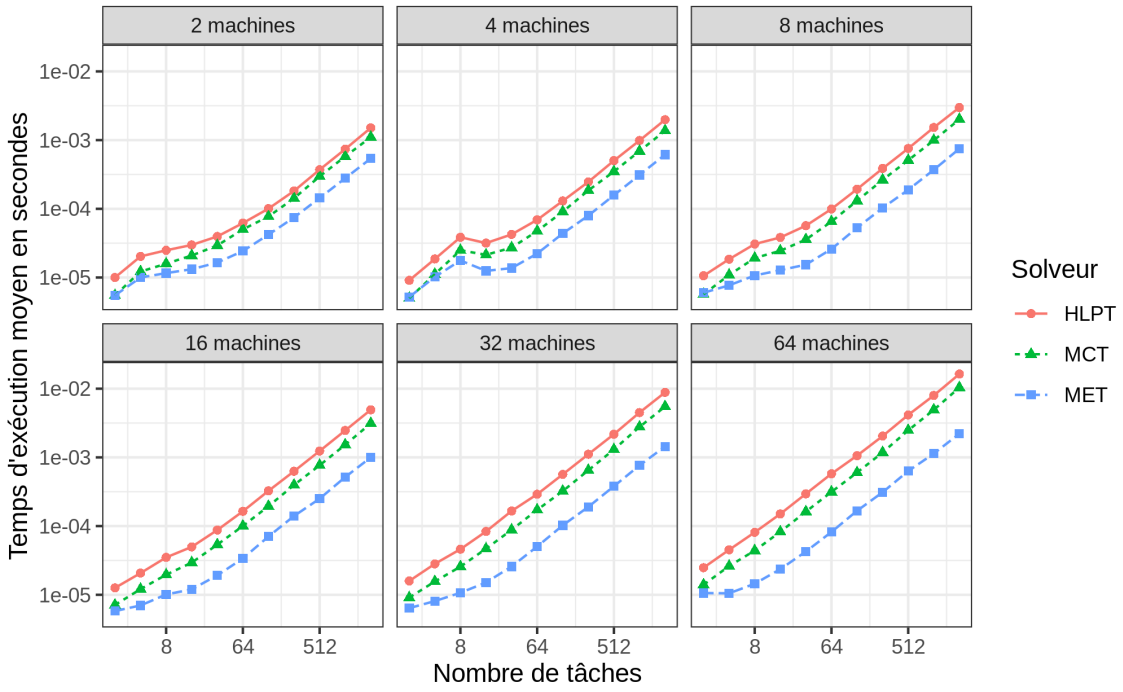

FIGURE 3.5 : Performances des heuristiques d'ordonnancement de liste sur des cas faciles représenté dans un repère log/log

Les résultats sont très similaires, puisque les caractéristiques précises du problème ne changent pas la complexité des algorithmes utilisés. Une fois de plus, une échelle logarithmique est utilisée sur les 2 axes, nous pouvons donc observer une évolution relativement linéaire de la complexité en fonction du nombre de tâches. De façon générale, les performances de ces heuristiques sont extrêmement bonnes, en moyenne mille fois plus rapide pour la plus lente d'entre elles par rapport à Highs.

Il faut toutefois désormais nous intéresser au taux d'échec de ces heuristiques. Ce taux d'échec est donc défini comme la proportion des cas où les heuristiques ne trouvent pas de solutions alors qu'il en existe une. Comme tous les problèmes traités dans les tests de ces heuristiques ont une solution, il s'agit simplement de la proportion de réponse négative. Dans les cas de problèmes simples, seul MET échoue, et ce 10% du temps. Dans le cas de problèmes plus difficiles, MET échoue 65% du temps, MCT 19% du temps et HLPT 13% du temps. Au

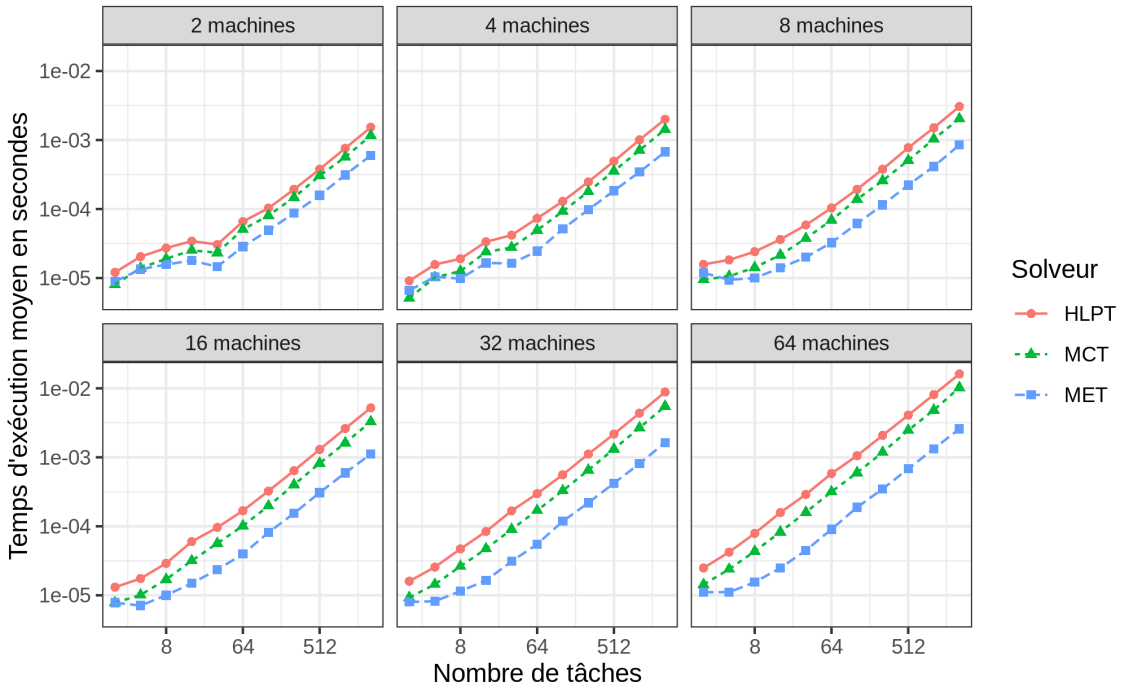

FIGURE 3.6 : Performances des heuristiques d'ordonnancement de liste sur des cas difficiles représenté dans un repère log/log

vu du gain de performance lorsque l'algorithme donne un résultat positif, la meilleure solution semble être d'utiliser MCT ou HLPT conjointement à un algorithme exact, ne lançant ce dernier qu'en cas de réponse négative de l'heuristique utilisée. Nous comparons cette méthode à l'utilisation de l'algorithme exact seul dans la figure 3.7

Nous noterons qu'en moyenne les performances de la méthode hybride sont ici systématiquement meilleures (un échec ne faisant perdre que peu de temps). Il est également remarquable que la répartition des cas où l'approximation échoue ne semble pas être due au hasard, et est assez hétérogène. Pour la comparaison avec Heptagon, nous préférerons utiliser les algorithmes d'ordonnancement exacts, dont les performances sont moins instables. Passons désormais à l'évaluation de ces différentes implémentations.

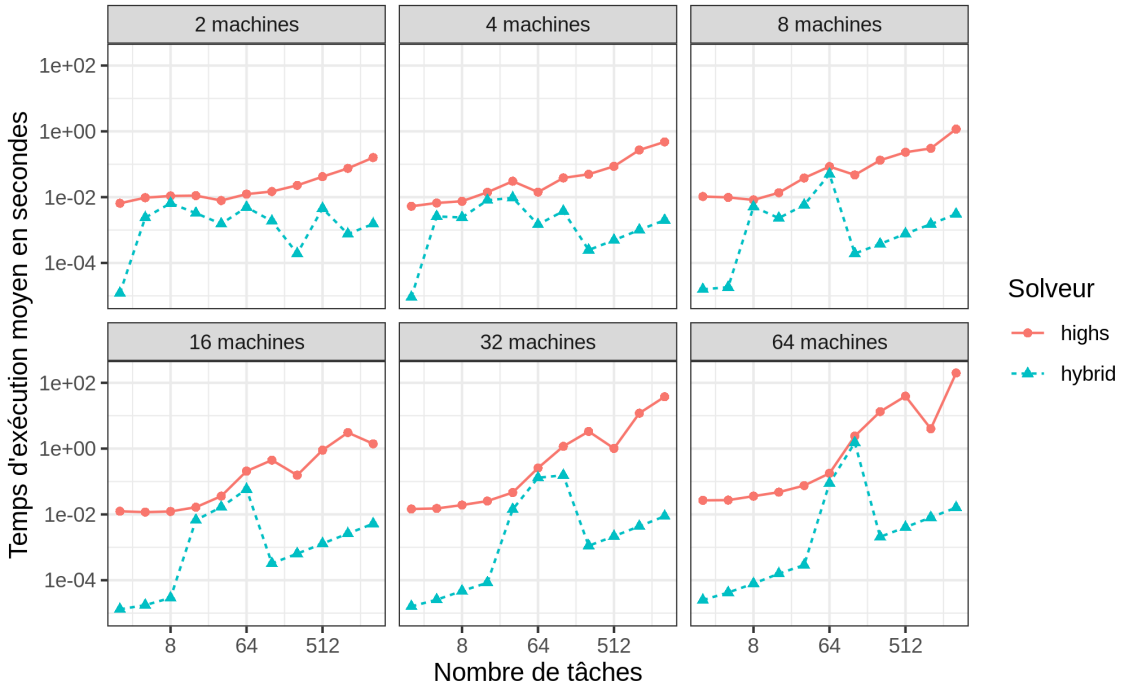

FIGURE 3.7 : Performances des heuristiques d'ordonnancement de liste sur des cas difficiles sur un repère log/log

— 4 —
Évaluations et comparaisons

4.1 Type de problèmes générés

Pour évaluer nos différents outils, et pour éviter de traiter des cas trop triviaux, les instances sont générées aléatoirement de la manière suivante : pour un problème donné avec n tâches et m machines, les temps d'exécution $p_{i,j}$ suivent une loi normale. L'espérance μ de cette loi normale est la fraction pouvant s'écrire $\frac{1}{i}$ avec $i \in \mathbb{N}^*$ la plus grande qui est inférieure ou égale à $\frac{2}{n}$. De manière plus simple, μ vaut $\frac{2}{n}$ si n est pair, et $\frac{2}{n+1}$ sinon. La variance σ^2 vaut un dixième de l'espérance. Ces paramètres permettent souvent de trouver une solution lorsque nous nous restreignons à n'importe laquelle combinaison de deux machines. Nous ne pouvons pas beaucoup mieux justifier ce choix : les problèmes pouvant être décrits avec la formalisation sont trop nombreux, nous devons choisir arbitrairement un sous-ensemble. Normalement, la famille choisie ne devrait pas être remplie de cas triviaux.

Pour s'assurer que les problèmes utilisés pour comparer les méthodes ont bien une solution pour toutes les configurations autorisées, nous ne lançons les algorithmes d'ordonnancement que sur des problèmes qui arrivent à passer la synthèse Heptagon. Le processus entier de synthèse est ainsi séparé de l'évaluation comparée des performances.

4.2 Évaluation hors ligne d'Heptagon

4.2.1 Métriques retenues

La particularité de l'approche utilisant Heptagon/BZR est le coût important de la synthèse, qui contrairement au reste des programmes décrits dans ce stage doit s'effectuer hors-ligne. Les programmes utilisés pour réaliser des ordonnancements n'ont pas besoin d'être synthétisés. Nous nous intéresserons donc ici au temps total de compilation et de synthèse. Nous observerons également l'évolution de la taille des exécutables produits.

4.2.2 Compilation et obtention des données

La processus de compilation Heptagon est comme suit :

- Compilation du programme .ept initial via heptc, production du code lié aux différents contrats

- Synthèse des contrôleurs à l'aide de ReaX
- Traduction du code généré par ReaX en langage Heptagon
- Compilation du contrôleur au format .ept généré par l'étape précédente.
- Compilation du programme complet à l'aide de gcc

Parmi toutes ces étapes, la seconde et la quatrième représentent l'essentiel du temps de calcul, la troisième et la cinquième sont assez courtes et la première est totalement négligeable. Le script réalisant ces étapes stocke les différentes durées d'exécution dans des fichiers au format TSV (annexe 5.1), afin de permettre l'analyse ultérieure de ces données. Les exécutables générés sont stockés dans le répertoire contenant les scripts en rapport avec l'évaluation en ligne, aux côtés des problèmes associés stockés sous la forme précédemment évoquée (annexe 2). Actuellement, nous compilons des cas avec n allant de 1 à 8, m allant de 2 à 5. Nous ne tentons pas de cas avec des n ou m plus élevés, car à partir d'un certain nombre de machines et / ou de tâches, la demande en mémoire de l'outil de synthèse explose au point que la synthèse n'aboutit pas. Nous pouvons citer comme limites $(n = 5, m = 10)$, $(n = 6, m = 7)$ et $(n = 7, m = 6)$, qui ne compilent pas pour cette raison, en tout cas sur l'ordinateur utilisé qui possède 16 Go de RAM.

À noter que dans les cas limites, le processus de compilation prend une bonne dizaine de minutes à s'effectuer. Cela nous rend la tâche de créer une base de données suffisamment large pour obtenir des résultats significatifs difficile, y compris pour la comparaison des différentes approches.

La compilation de chacun des problèmes est cependant indépendante, et le procédé n'occupe systématiquement qu'un seul cœur, il a donc été possible de modifier le script de synthèse pour qu'il compile plusieurs programmes de mêmes dimensions en parallèle. L'outil utilisé pour cela est "GNU Parallel" [20], qui permet notamment d'éviter automatiquement de lancer plus de processus que de cœurs disponibles. Si cette méthode permet déjà d'obtenir une accélération considérable sur notre machine possédant huit cœurs, elle est d'autant plus nécessaire qu'il a été envisagé de passer à l'échelle à l'aide de plateformes dédiées au calcul parallèle telles que Grid5000 [1], ce qui n'a finalement pas été nécessaire.

4.2.3 Evolution du temps de synthèse et de la taille de l'exécutable

L'évolution à nombre de machines fixé du temps de compilation en fonction du nombre de tâches est représentée sur quatre graphiques (figure 4.1).

Une échelle logarithmique est utilisée pour l'axe des ordonnées. Le côté exponentiel du temps de compilation apparaît donc clairement ici. Nous pouvons cependant remarquer que la courbe ne ressemble pas à une droite, et semble plutôt suivre une trajectoire en dents de scie. Ceci s'explique par l'arrondi à la fraction de numérateur 1 inférieure la plus proche utilisé lorsque n est impair. Nous pouvons voir que l'effet de cet arrondi sur la croissance du temps de synthèse augmente avec le nombre de tâches, seulement pour régler ce problème il faudrait se restreindre aux nombres pairs ou impairs de tâches, mais comme nous ne dépassons pas $n = 8$, nous n'aurions donc que quatre points à mesurer.

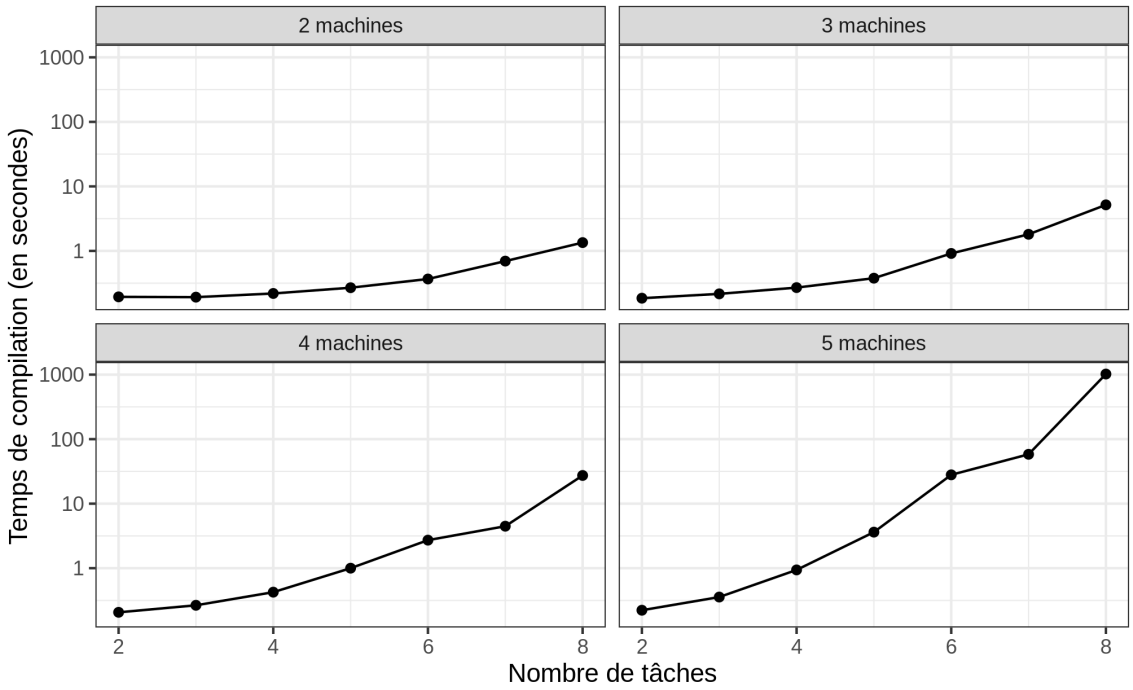

FIGURE 4.1 : Évolution du temps de synthèse, l'axe des ordonnées suit une échelle logarithmique

À présent, nous allons observer l'évolution de la taille du fichier exécutable produit en fonction du nombre de tâches et de ressources. Le format du graphe est identique au précédent (figure 4.2).

Deux points sont apparents sur ce graphique. Premièrement, la taille de l'exécutable est exponentielle en le nombre de tâches. Secondairement, l'évolution de cette taille possède une ressemblance frappante avec celle du temps de synthèse (voir la régression linéaire correspondante en annexe 8).

4.3 Comparaison en ligne

4.3.1 Métriques retenues

Dans le cadre des performances en ligne, la comparaison entre les deux approches devient plus naturelle. Nous avons notamment retenu le temps de réponse en ligne des différentes méthodes comme critère principal. Comme nous nous sommes attendu à ce que le contrôleur généré par Heptagon soit plus rapide, une autre métrique dérivée a été retenue : le nombre d'itérations avant que l'utilisation d'Heptagon ne devienne rentable, en prenant en compte le temps de compilation.

4.3.2 Évaluation et obtention des données

Le script d'évaluation, pour chacun des problèmes ayant passé le processus de compilation pendant l'étape précédente et pour chaque nombre de machines désactivées *nb* possible (0 à

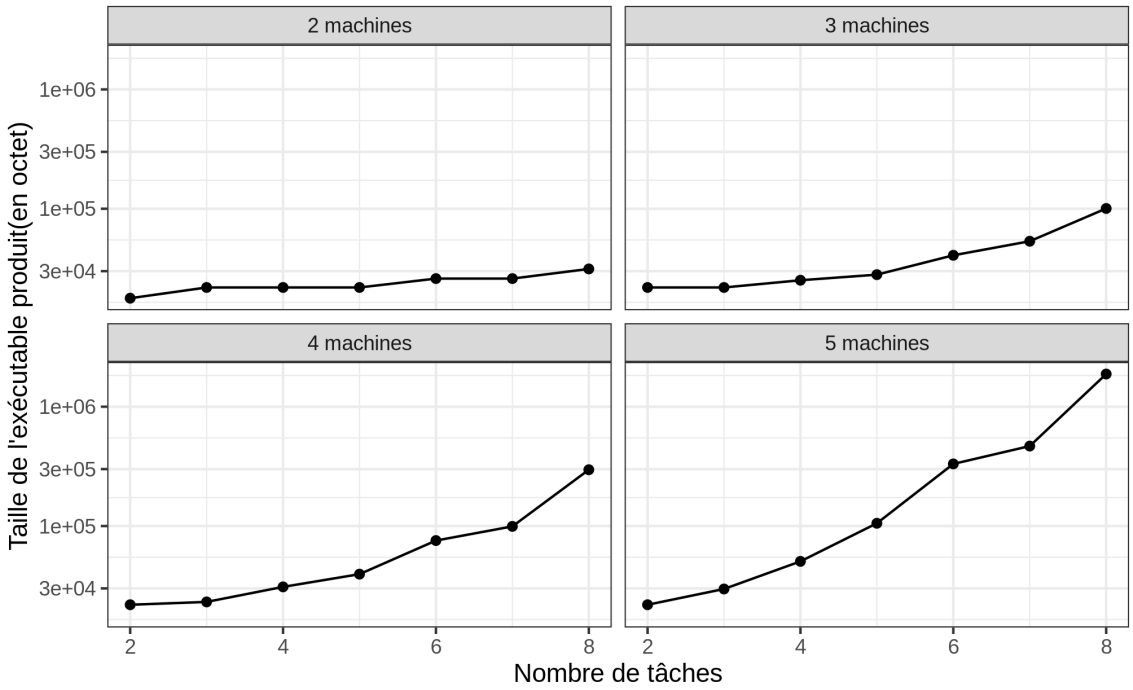

FIGURE 4.2 : Évolution de la taille de l'exécutable, l'axe des ordonnées suit une échelle logarithmique

$n-2$), génère des entrées pour le programme Heptagon. Ces entrées sont envoyées dans des fichiers temporaires, et prennent la forme de la liste de booléens suivante :

```
1 tache_1_active tache_2_active ... tache_n_active
2 resource_1_available resource_2_available ... resource_m_available
```

Dans les cas que nous générons, toutes les tâches restent actives, et nous désactivons aléatoirement *nb* ressources différentes. Ces fichiers d'entrées permettent immédiatement après d'extraire à partir des problèmes associés aux exécutables Heptagon les sous problèmes qui seront traités par les méthodes d'ordonnancement. Le procédé est très simple : un nouveau fichier est généré dans le format décrit en annexe 2, avec n prenant comme valeur le nombre de tâches actives, m le nombre de ressources disponibles, et en gardant uniquement les temps d'exécution encore pertinents. Un exemple de cette extraction est présent en annexe 3.

Une fois ces fichiers obtenus, nous les exploitons afin d'évaluer les différentes approches. Nous mesurons le temps d'exécution d'un pas des exécutables générés par Heptagon, c'est-à-dire de 50 000 pas du contrôleur évoqués en partie 3.2.2. Ensuite, nous mesurons le temps mis par les deux algorithmes d'ordonnancement pour résoudre le sous-problème correspondant 10 fois. Tous ces temps d'exécution sont stockés dans un même dossier au format TSV décrit en annexe 5.2. Un autre script permet de récupérer, toujours au format TSV décrit en annexe 5.3, la taille des fichiers décrivant les problèmes et des exécutables, associés aux dimensions du problème.

4.3.3 Observation des résultats

Au vu des temps de compilation des programmes Heptagon, il est difficile d'en obtenir en nombre. Les résultats décrits ci-après ne se baseront donc que sur dix problèmes pour chaque (n,m) fixé, pour le moment. Ce nombre pourrait augmenter en cas d'usage de Grid5000 pour générer un nombre plus conséquent d'exemples.

Pour obtenir un temps moyen pour chaque méthode et chaque couple (n,m), nous avons dû répéter un certain nombre de fois les différentes étapes. Pour la partie Heptagon, la méthode utilisée a déjà été décrite : des tableaux sont employés pour que chaque pas apparent soit l'équivalent de cinquante mille pas réels. Pour les techniques d'ordonnancement, nous réalisons simplement la moyenne du temps d'exécution de cinquante résolutions séparées. Une fois ramené à un temps d'exécution moyen exprimé en millisecondes, voici le graphe des temps d'exécution obtenu : (figure 4.3)

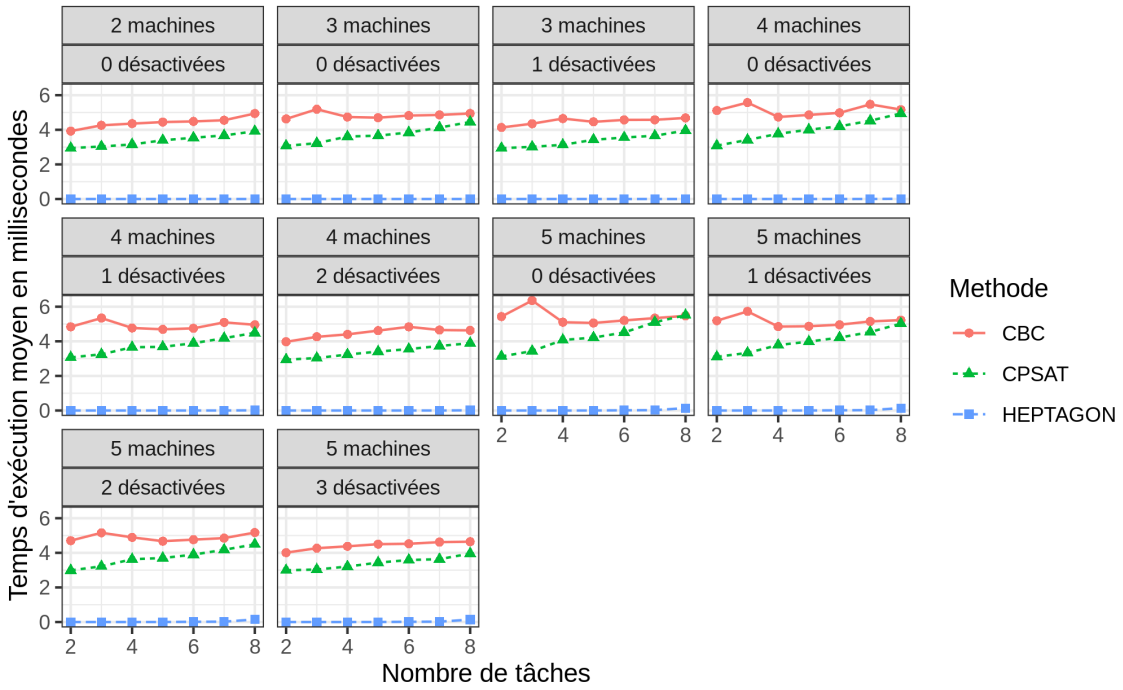

FIGURE 4.3 : Évolution du temps d'ordonnancement en ligne

Il est observable que l'algorithme d'ordonnancement utilisant CBC est nettement plus lent que celui utilisant cp-sat, particulièrement pour les cas les plus simples. Manifestement, le temps d'exécution d'un pas du programme généré par synthèse de contrôleur n'est pas dans le même ordre de grandeur, et est donc trop faible pour être comparé à l'œil nu sur ce graphe. Nous avons donc décidé de produire un graphique montrant plutôt l'évolution du rapport entre les temps d'exécution des algorithmes d'ordonnancement et la durée d'un pas d'Heptagon (figure 4.4).

Sur ce graphe, ayant une fois de plus une échelle logarithmique sur les ordonnées, nous pouvons observer que l'avantage de la synthèse de contrôleur s'estompe à vitesse exponentielle, particulièrement lorsque le nombre de machines augmente. Par exemple, dans le cas avec 5 machines, dont trois désactivées, on passe d'un rapport de l'ordre de 10^4 à 10 lorsque n augmente. Nous avons pu voir sur le graphe précédent que le temps d'exécution des algorithmes

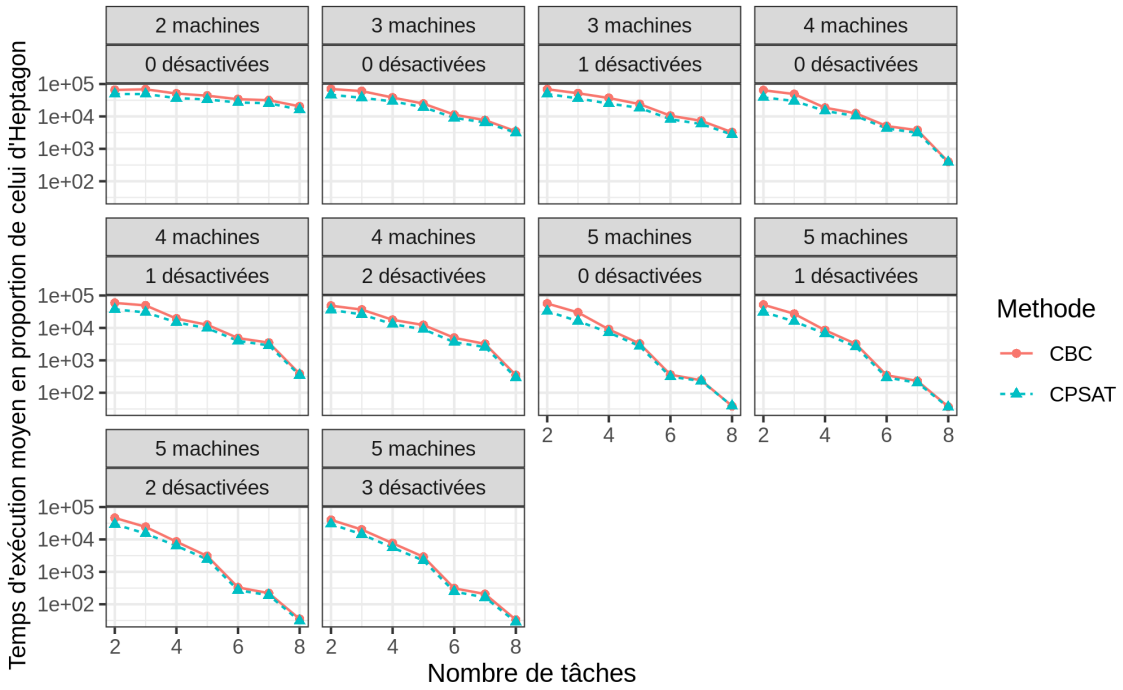

FIGURE 4.4 : Évolution du temps d'ordonnancement en ligne en proportion, l'axe des ordonnées suit une échelle logarithmique

d'ordonnancement, s'il augmente légèrement avec n et m, reste pourtant dans le même ordre de grandeur. Nous pouvons donc supposer que le temps d'exécution d'un pas du contrôleur est en fait exponentiel, ce qui va à l'encontre de nos attentes. Nous allons observer l'évolution du temps d'exécution d'un pas pour confirmer ce doute (figure 4.5).

Ce graphe corrobore bien l'observation précédente : au-delà des perturbations causées par les changements de difficultés lorsque le nombre de tâches oscille entre pair et impair, le temps d'exécution d'un pas du programme généré par Heptagon semble croître de manière exponentielle. Il est à noter que le nombre de machines désactivées ne modifie pas significativement le temps d'exécution pour Heptagon, d'où l'absence de prise en compte de ce critère ici (Graphe le prenant en compte en annexe 6).

Nous pouvons remarquer, à l'allure du graphe, qu'il est une fois de plus très similaire à celui du temps de compilation et de la taille du programme généré. Il semble de fait y avoir une forte corrélation, puisqu'en effectuant une régression linéaire en tentant d'expliquer le temps d'exécution par la taille du programme généré, nous obtenons une p-value minimale et un R^2 proche de 0.98. Plus de détails sont disponibles dans cette annexe 7. Le temps d'exécution est manifestement linéaire en la taille du fichier exécutable, malgré nos attentes initiales puisque nous nous attendions à une complexité logarithmique en fonction de la taille du fichier.

Le fait que les temps de compilation et d'exécution soient tout autant exponentiels va forcément affecter fortement le seuil de rentabilité d'Heptagon par rapport à l'ordonnancement, c'est à dire le nombre de pas nécessaires pour qu'Heptagon devienne plus rapide que les méthodes utilisant des solveurs, en prenant en compte l'importante durée de synthèse. Nous allons observer l'évolution de cette métrique (figure 4.6).

Sans surprise au vu des résultats précédents, le seuil de rentabilité évolue également de

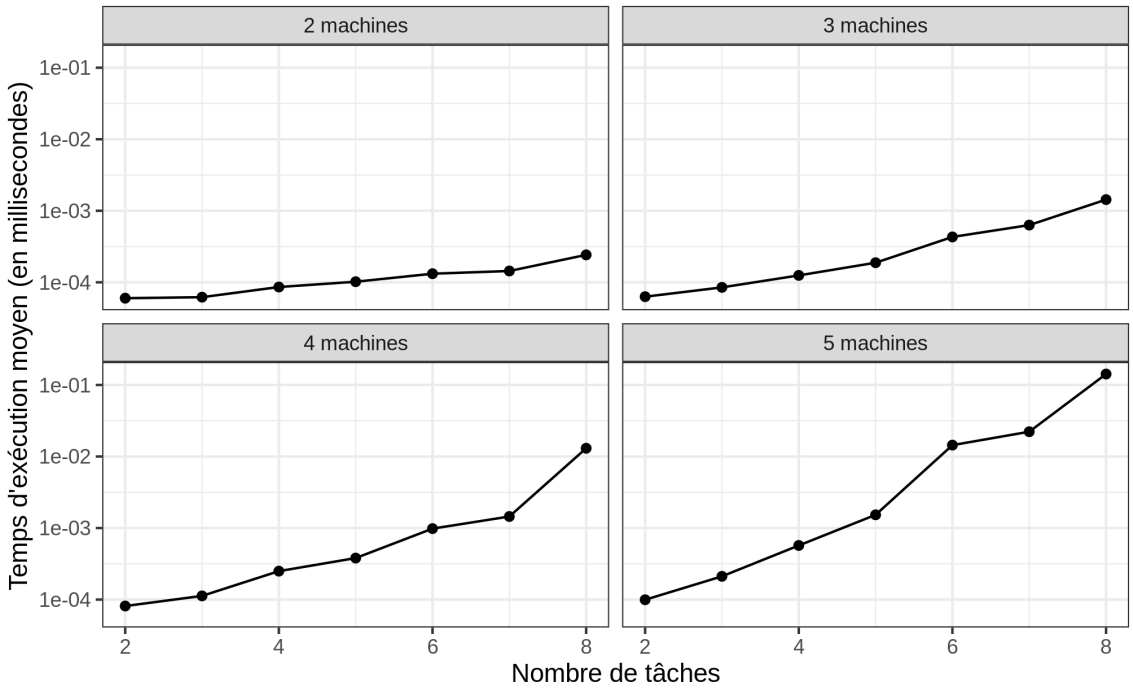

FIGURE 4.5 : Évolution du temps d'ordonnancement en ligne, l'axe des ordonnées suit une échelle logarithmique

manière exponentielle. S'il ne faut qu'une centaine d'itérations pour l'atteindre dans les cas les plus simples, dans le cas testé le plus difficile, il est nécessaire d'attendre une centaine de milliers d'itérations pour que le temps de synthèse soit rentabilisé.

Tous ces graphiques mettent en évidence que, même si la synthèse de contrôleurs possède certains avantages sur les techniques d'ordonnancement utilisées ici, principalement un temps de réponse plus faible en ligne, ses performances sont pires sur tous les autres critères. De plus, ces qualités restantes s'estompent très rapidement, et nous pouvons supposer que, si la synthèse permettait la compilation de problèmes un peu plus difficiles, elle deviendrait rapidement moins performante que les techniques concurrentes y compris en ligne.

Il existe cependant un problème bien plus majeur : les performances relatives à des algorithmes naïfs.

4.4 Utilisation d'algorithmes naïfs

4.4.1 Motivations menant à utiliser ces algorithmes

Nous avons remarqué pendant ce stage que les problèmes traités pour réaliser de l'ordonnancement en Heptagon ne profitent que très peu des particularités et des avantages du langage. En particulier, il n'y a ni état ni mémoire dans le programme exécutable, ce qui signifie que pour un jeu d'entrées données à un instant t, le programme se comportera de l'exacte même manière, peu importe les entrées reçues aux pas antérieurs. En pratique, cela signifie que la méthode utilisant Heptagon consiste à trouver un ordonnancement pour chaque possibilité de machines/tâches actives, et les restituer lorsque l'entrée correspondante est reçue.

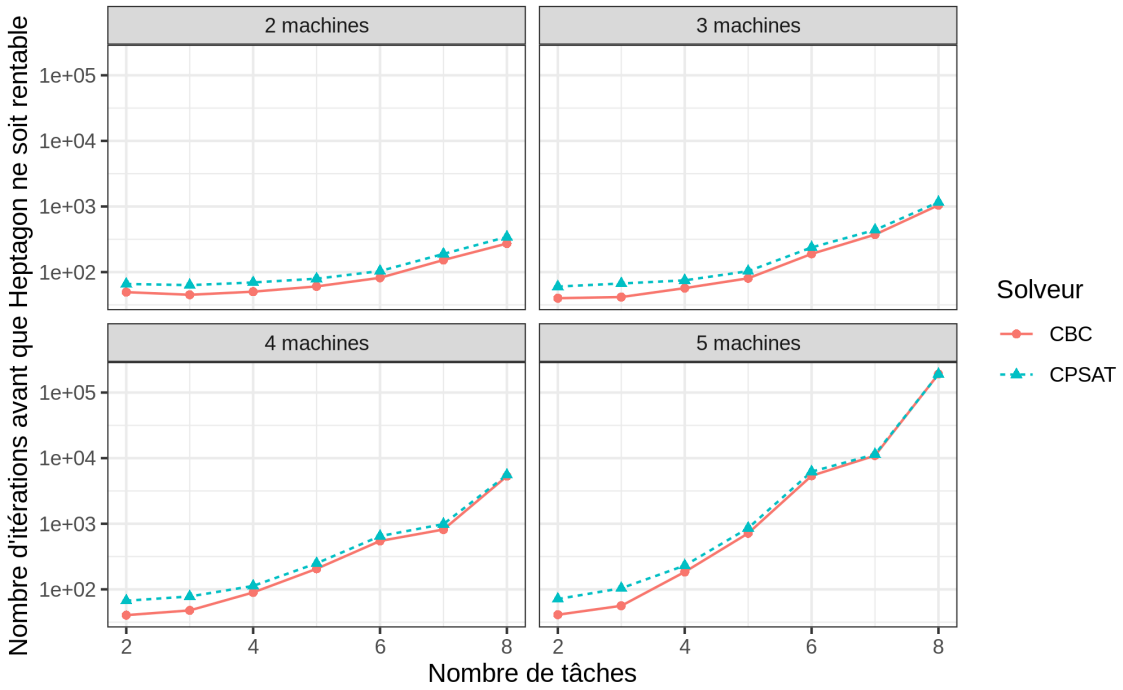

FIGURE 4.6 : Évolution du seuil de rentabilité des programmes Heptagon, l'axe des ordonnées suit une échelle logarithmique

De manière entièrement indépendante, l'outil de synthèse ReaX ne fonctionne pas actuellement entièrement comme il le faudrait, et réalise parfois des faux positifs lorsque les problèmes n'ont pas de solution, indiquant que la synthèse à réussi alors que c'est impossible. Les exécutables ainsi générés ne respectent bien entendu pas le contrat (puisque c'est impossible). Cependant, lorsque nous sommes certains que le problème à bien une solution, nous n'avons jamais mis en défaut le contrôleur généré. Nous avions donc décidé, au vu de la taille des problèmes traités, de vérifier en amont de la compilation que les problèmes possèdent bien un ordonnancement pour chaque couple de ressources. Cette vérification est réalisée à l'aide d'un programme énumérant naïvement les différentes possibilités pour chaque couple jusqu'à trouver une solution. S'il n'en trouve pas pour un couple, il s'arrête immédiatement et signale que l'ordonnancement est impossible, nous permettant de relancer la génération de problèmes.

Une fois que cet algorithme a été écrit et utilisé dans le processus de génération des jeux de données, et au vu de la rapidité du processus, l'idée est venue de comparer nos résultats à une méthode naïve similaire. Comme vu dans le premier paragraphe, nous avons donc réalisé un algorithme naïf en deux étapes, l'une réalisant un travail analogue à la synthèse en trouvant par énumération une solution (si elle existe) pour chaque combinaison de machines disponibles, et l'autre fonctionnant de manière similaire à l'ordonnancement en ligne par Heptagon, chargeant initialement la liste des solutions dans un dictionnaire ayant pour clefs les configurations, puis lisant simplement ce dictionnaire pour trouver la réponse adaptée aux entrées actuelles. En exploitant cette fois-ci l'hypothèse qui veut que l'on puisse trouver un ordonnancement pour n'importe quelle paire de deux machines restantes, nous avons écrit un second algorithme, cette fois-ci en ligne, qui tente de trouver un ordonnancement par énumération des configurations, mais en se limitant à deux machines parmi celles qui sont disponibles. Nous nommerons cet

algorithme "basic", et l'autre "naive" dans les graphes qui suivent.

4.4.2 Comparaisons avec les techniques précédentes

Performances comparatives en ligne

Nous comparons les performances de ces deux algorithmes à celles de l'exécutable généré par Heptagon (figure 4.7) :

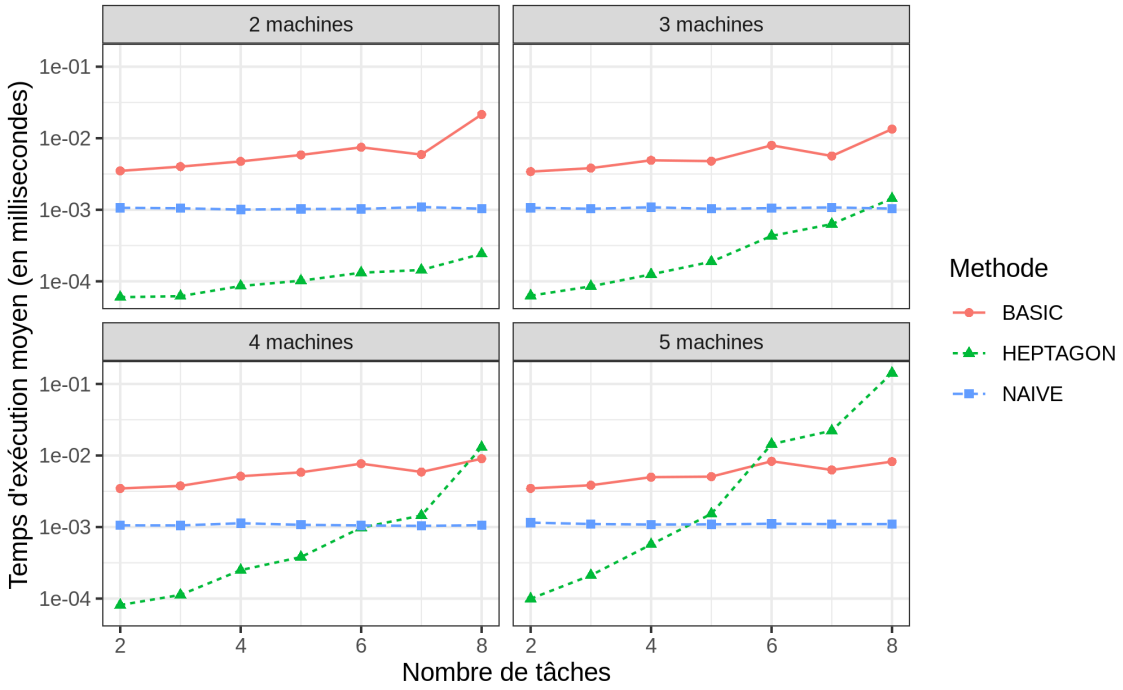

FIGURE 4.7 : Comparaison du temps d'ordonnancement en ligne d'Heptagon et des méthodes naïves, l'axe des ordonnées suit une échelle logarithmique

Avant toute autre remarque, notons que le code créé par Heptagon est en langage C, tandis que les algorithmes naïfs sont écrits et exécutés en Python. Il est notoire que le temps d'exécution d'une instruction en python est bien plus lent, nous comparerons donc ici surtout l'évolution des courbes.

En tablant sur une exécution 10 fois plus rapide sur le langage compilé, ce qui est largement en deçà de la vérité, nous noterons donc que l'algorithme "naive" est constamment meilleur que Heptagon. Après tout, il ne fait que lire le résultat plutôt que de le calculer. Nous comparerons plus tard les performances lors de la synthèse. Pour ce qui est de l'algorithme "basic", il est initialement probablement un peu plus lent, mais sa complexité semble croître beaucoup plus lentement que celle de l'exécutable Heptagon, ce qui est logique notamment puisqu'elle ne dépend pas du nombre de machines.

Comparaison des temps de "synthèse"

Nous comparons ici le temps de synthèse d'Heptagon au temps mis par l'algorithme naïf hors ligne pour calculer une solution pour chaque configuration possible du système étudié (figure

4.8) :

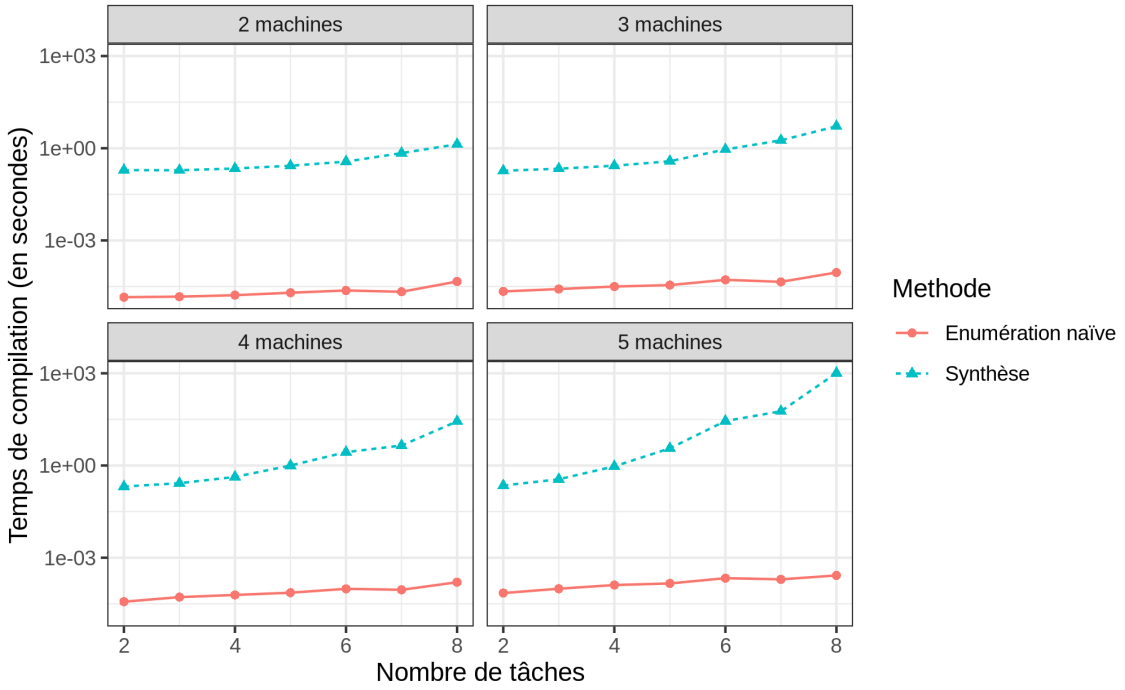

FIGURE 4.8 : Comparaison du temps de "synthèse" d'Heptagon et des algorithmes naïfs, l'axe des ordonnées suit une échelle logarithmique

Ici, la courbe des ordonnées utilise une fois de plus une échelle logarithmique, et nous pouvons donc observer l'avantage considérable de l'algorithme naïf par rapport à Heptagon. Il est systématiquement bien plus rapide, et sa croissance est bien moins exponentielle (notamment car les cas traités sont simples). Il ne semble donc bel et bien pas y avoir d'intérêt à utiliser Heptagon actuellement pour traiter ce genre de problème.

Comparaison avec les ordonnanceurs

Pour vérifier que ces algorithmes naïfs ne sont pas intéressants lorsque les problèmes sont plus difficiles, ils ont été comparés aux performances des ordonnanceurs sur des cas aux dimensions plus élevées que celles que peut traiter Heptagon. Nous commençons par observer les performances de l'algorithme "basic" en pire cas (figure 4.10). Sur ces exemples, qui vont de 2 à 24 tâches et de 2 à 8 machines, nous pouvons noter que, une fois les 15 tâches dépassées, l'algorithme "basic" devient beaucoup moins rentable en pire cas que tous les algorithmes d'ordonnancement. Sachant que nous pouvions facilement traiter des cas bien plus difficiles avec la programmation linéaire et la programmation par contraintes, ce résultat est bien celui qui est attendu.

D'un autre côté, comparons le temps de calcul hors ligne de l'algorithme "naive" dans le pire cas à ces algorithmes d'ordonnancement(figure 4.9). Cette fois-ci, les cas traités vont de 2 à 10 tâches, et de 2 à 6 machines, et le temps de pseudo synthèse explose très rapidement, mais toujours moins qu'Heptagon (100 secondes dans le pire cas ici, qu'Heptagon ne pourrait pas traiter actuellement).

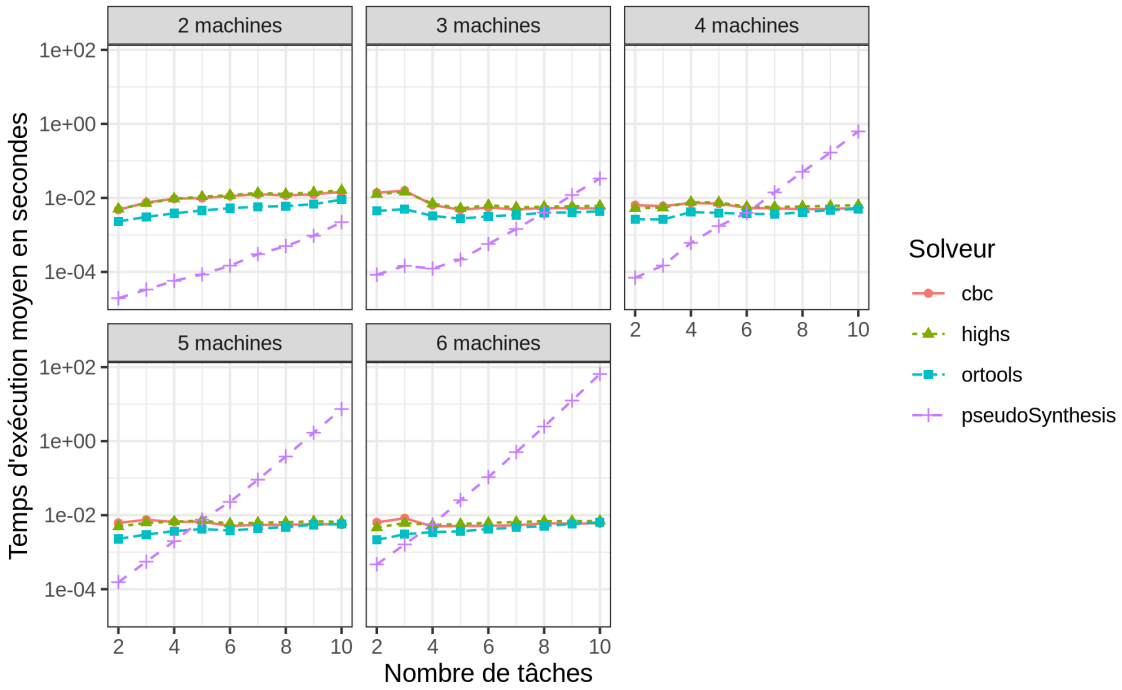

FIGURE 4.9 : Comparaison des techniques d'ordonnancement à la partie hors-ligne de "naive", l'axe des ordonnées suit une échelle logarithmique

Pour tenter de remédier à cette situation, il a été nécessaire de modifier le code exécutable généré par le compilateur Heptagon. Les modifications apportées sont décrites dans le chapitre suivant.

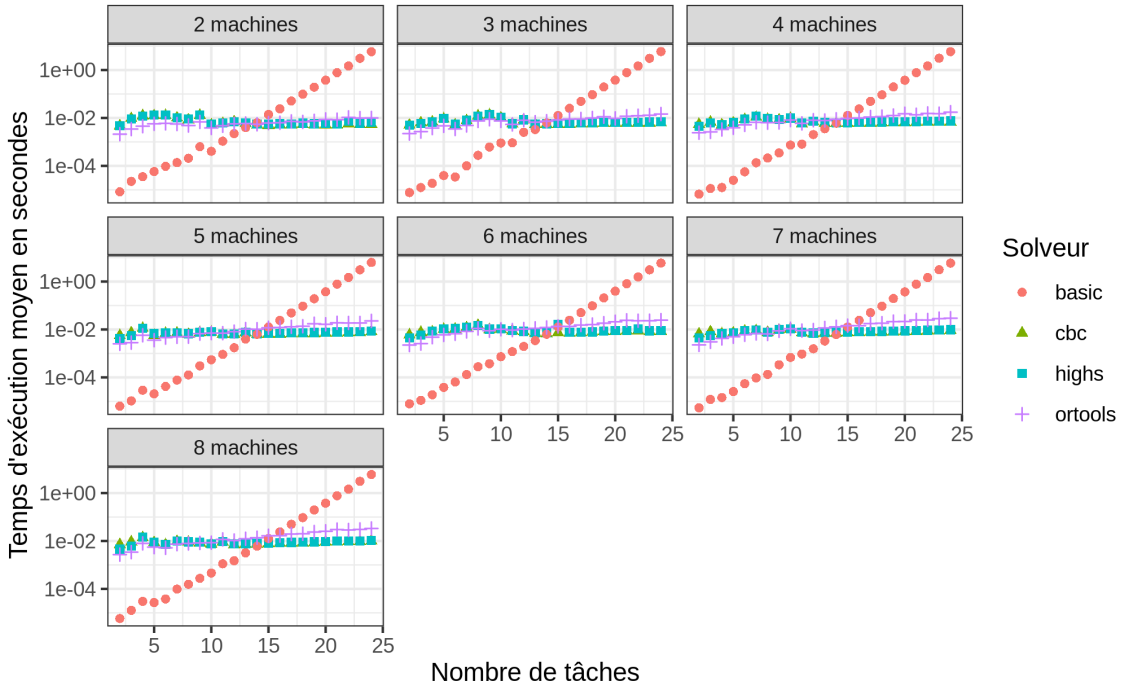

FIGURE 4.10 : Comparaison des techniques d'ordonnancement et de l'algorithme "basic", l'axe des ordonnées suit une échelle logarithmique

— 5 —
Modifications du compilateur Heptagon

5.1 Motivations

Comme évoqué précédemment, nous supposions en début de stage que la durée d'exécution du programme généré par Heptagon serait linéaire en le nombre de variables contenu dans le programme initial. En effet, la partie la plus coûteuse est l'exécution du contrôleur, que nous croyions être écrit sous la forme de if imbriqués, de telle sorte que sa complexité soit logarithmique en sa taille. L'outil ReaX manipule en effet des BDD, et il était supposé que le fichier Nbac en sortie gardait cette structure.

Il s'est cependant avéré que le code produit ne possédait au contraire aucun if imbriqué, et son temps d'exécution ne dépendait pas des valeurs reçues en entrée tout en étant linéaire en la taille du programme. Le résultat observé était un temps d'exécution qui, s'il reste faible dans les cas qui compilent, croît de manière exponentielle, contrairement à nos attentes.

Pour régler ce problème, l'idée a donc été à partir du code renvoyé par ReaX de retrouver la structure de BDD originale pour pouvoir produire du code plus rapide.

5.2 Emplacement des modifications

Au sein du processus de compilation, nous nous intéressons ici à ctrl2ept, l'utilitaire dédié à la traduction des fichier Nbac générés par ReaX en fichier source Heptagon. Une fois que cette traduction est effectuée, plutôt que de passer par le processus de compilation classique des fichiers .ept, nous exploitons l'arbre abstrait Heptagon obtenu pour générer directement du code C efficace.

5.3 Algorithmes employés

5.3.1 Restauration de la structure de BDD

Dans l'arbre abstrait initial, le corps est composé d'une équation par variable locale, et d'une équation par variable de sortie. Ces équations correspondent soit à une assignation simple, soit à une assignation conditionnée. Une assignation simple ressemble à cela :

```
1  variable = val;
```

Une assignation conditionnée ressemble, elle, à cela :

```
variable = if cond then val1 else val2;
```

La structure nécessaire est donc presque déjà présente : nous pouvons retrouver les variables "filles" d'une certaine variable en parcourant l'équation correspondant à la variable parente. Cependant, nous n'aurions alors pas accès à l'expression liée à ces variables filles. La solution proposée est simplement d'utiliser une table de hachage, liant chaque variable à l'expression qui lui correspond.

5.3.2 Code C correspondant à l'arbre binaire

Pour générer la fonction C correspondant au contrôleur, nous parcourons l'arbre abstrait Heptagon. La liste des variables d'entrée et de sortie, présente dans la déclaration du nœud, permet d'écrire la déclaration de la fonction correspondante. De même, la liste des variables locales présente dans la déclaration du corps du nœud permet une déclaration analogue dans la fonction C.

Ensuite, nous pouvons commencer la génération du corps de la fonction, en démarrant par les variables de sorties. L'idée est, avant de définir une variable, de calculer la valeur de toutes les valeurs nécessaires à cette définition.

Par exemple, une assignation simple sera traduite tout d'abord en générant le code correspondant à son membre de droite, puis par une assignation en C. Par exemple, le code Heptagon suivant :

```
var1 = var2;
var2 = var3;
var3 = cst;
```

est traduit par

```
var3 = cst;
var2 = var3;
var1 = var2;
```

De manière analogue, les assignations conditionnées seront traduites suivant le processus suivant.

1. Définir les variables nécessaires au calcul de la condition, tout en récupérant la représentation textuelle de celle-ci.
2. Écrire le début de la condition, et entrer dans la première branche
3. Définir les variables nécessaires au calcul de la première branche, puis réaliser l'assignation
4. Écrire la sortie de la première branche, et l'entrée dans la seconde
5. Définir les variables nécessaires au calcul de la seconde branche, puis réaliser l'assignation
6. Terminer la définition de la branche conditionnelle.

Par exemple, avec le code Heptagon suivant :

```
1  var1 = cst1;
2  var2 = cst2;
3  var3 = (input1 = cst3);
4  var4 = if (var3) then var1 else var2;
```

nous obtenons, après traduction :

```
1  var3 = (input1 == cst3);
2  if (var3) {
3   var1 = cst1;
4   var4 = var1;
5  } else {
6   var2 = cst2;
7   var4 = var2;
8  }
```

Les variables de sorties ne sont jamais redéfinies, si leurs valeurs sont utilisées dans une expression, le nom de la variable est utilisé directement. De plus, la manière de renvoyer plusieurs valeurs en sortie est la modification d'une structure dont le pointeur est passé en entrée. La variable contenant ce pointeur est toujours nommée *_out*, ce qui fait qu'une variable nommée *output* sera traduite par $\_out->ouput$. Les variables d'entrées sont traitées comme des constantes dans le corps de la fonction.

Les constantes des types énumérés ne semblent pas attachées à leur module originel, mais au module actuel. C'est un problème, car lors de la compilation en C, ces types sont déclarées avec leurs noms complets (*NomDuModule__nomDuType*). La parade trouvée consiste à garder en mémoire, pendant le parcours de l'arbre, le type de la variable en cours d'assignation ou de test d'égalité. En effet, ce type est bien associé au module qui nous intéresse, et permet donc de retrouver le nom complet.

5.3.3 Élimination des redondances

Le problème de la technique naïve employée à l'instant est la possibilité de fortes redondances dans le code. En effet, si une même variable locale est utilisée dans plusieurs calculs indépendants, elle est intégralement recalculée, causant une perte de performances, et augmentant drastiquement la taille de l'exécutable.

L'idée ici est donc de générer du code sans ces redondances, en commençant pour le moment par exécuter de manière inconditionnelle tout code redondant une seule fois.

Détection des redondances

Pour détecter le code redondant, nous associons à chaque variable locale un nombre *nbPath*, initialement non défini. De la même manière que pour générer du code C, nous parcourons les BDD correspondant aux variables de sorties. Chaque fois qu'une variable locale est rencontrée, sa valeur *nbPath* est incrémentée si elle est définie, et mise à 1 sinon. Le parcours continue si cette valeur est désormais 1, et s'arrête sinon. Une fois ces calculs réalisés, les variables ayant un *nbPath* supérieur ou égale à 2 sont celles définies de manière redondante.

Par exemple, dans le code Heptagon :

```
1  var1 = var2;
2  var2 = cst;
```

```
3   output1 = var1;
4   output2 = var1;
```

La variable *var*1 est redondante, mais pas *var*2, car un seul chemin mène à cette dernière depuis var1.

Tri topologique pour l'ordonnancement des équations

Pour obtenir l'ordre dans lequel le code doit désormais être généré, il est nécessaire de réaliser un tri topologique sur les variables. À cette fin, nous parcourons une fois de plus les différents BDD des variables de sorties, en associant cette fois-ci une valeur $depth$ aux différentes variables. Cette valeur est initialisée à 0 pour les variables en sorties. De plus, la valeur actuelle de $depth$ est également passée dans les paramètres de la fonction récursive, démarrant à 1 en début de parcours. Lorsqu'une variable est rencontrée, si sa profondeur n'est pas définie, ou est inférieure à la valeur actuelle, elle devient identique à cette dernière. Dans ce cas, le parcours continue, avec le $depth$ actuel incrémenté de 1. Sinon, le parcours s'arrête. Une fois tous les calculs réalisés, un ordre total est défini sur les variables, celles ayant la plus haute valeur de $depth$ devant être définies en premier.

Modifications de la génération du code C

La génération de code C change très peu, les variables redondantes sont désormais ajoutées aux variables de sorties pour définir les points de départ de l'algorithme. De plus, la génération de code se fait désormais dans l'ordre défini dans la section précédente. Enfin, ces variables redondantes ne sont évidemment plus redéfinies si elles sont rencontrées dans la définition d'une autre variable, seul leur nom est alors utilisé.

5.4 Effets sur les performances

Deux critères principaux seront utilisés ici pour comparer ces nouvelles méthodes au code anciennement généré : la rapidité d'exécution, et la taille du fichier produit.

5.4.1 Traduction initiale

Commençons par observer l'évolution du temps de réponse (figure 5.1) : sur les cas les plus difficiles traitables par Heptagon jusqu'ici, le nouvel exécutable est plus rapide d'un facteur 10^3, le gain de performances est donc considérable. À noter cependant qu'il reste probablement de complexité exponentielle, puisque l'échelle des ordonnées est ici logarithmique et que la courbe à l'allure d'une droite.

Pour ce qui est de l'évolution de la taille des fichiers exécutables produits (figure 5.2) : sans surprise, le nouveau code généré est bien plus volumineux, environ dix fois plus long en moyenne dans le pire des cas ici. Cela est certainement causé par la forte présence de code dédoublé.

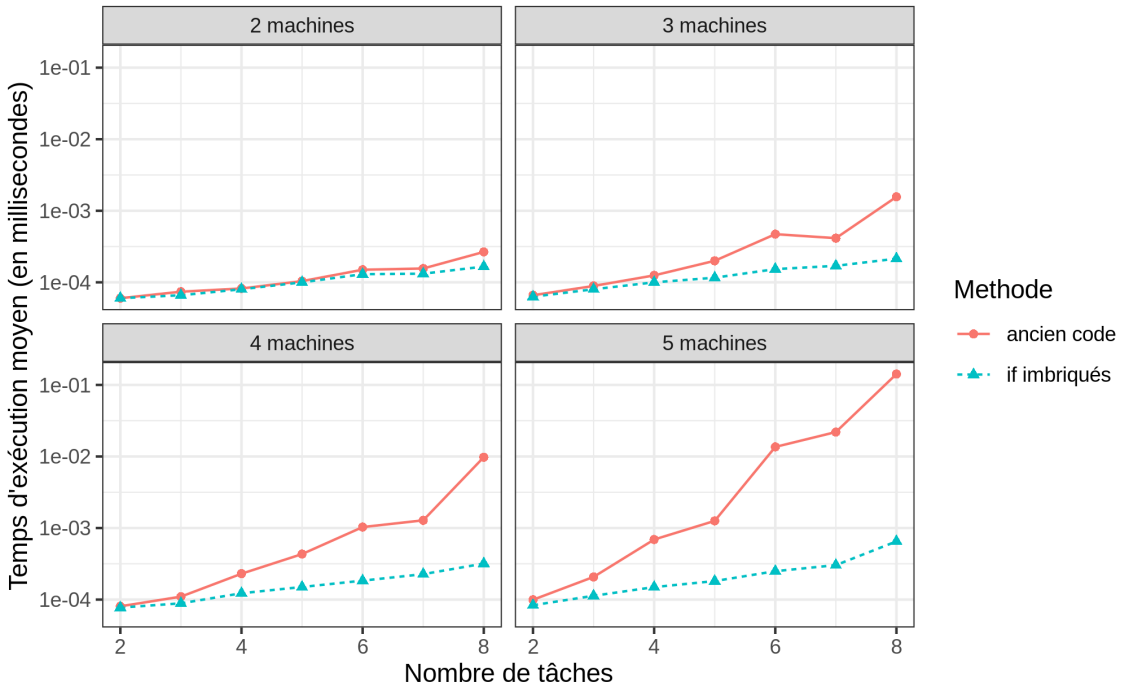

FIGURE 5.1 : Comparaison des temps de réponse en ligne de l'ancien code et de la nouvelle traduction, l'axe des ordonnées suit une échelle logarithmique

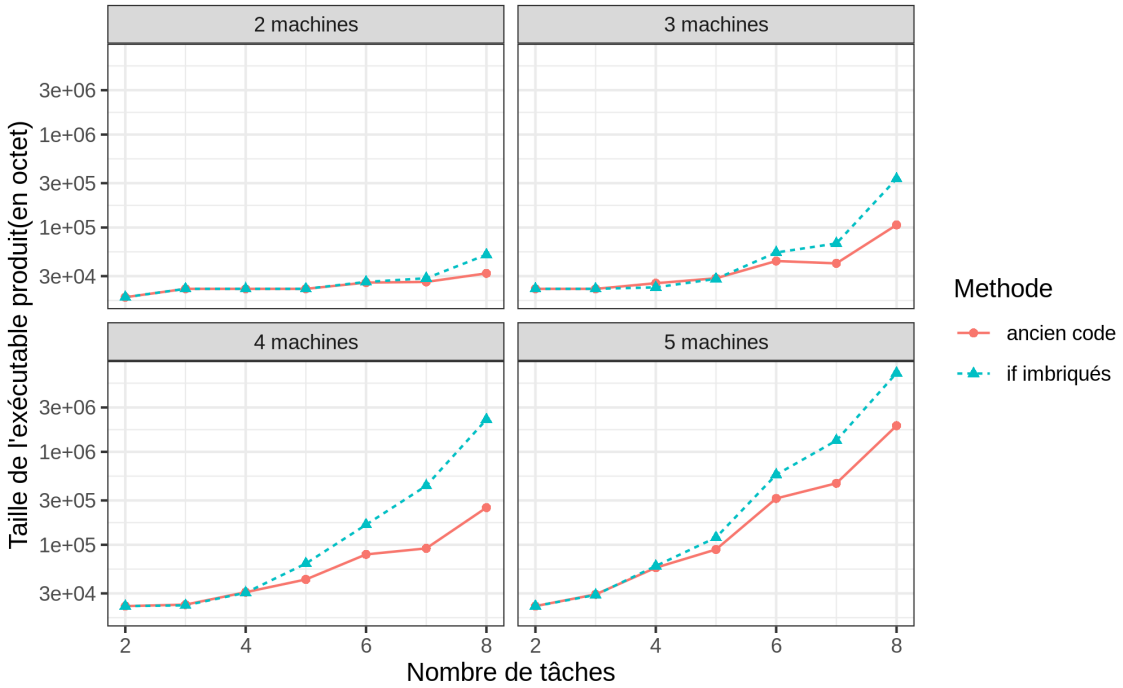

FIGURE 5.2 : Comparaison de la taille des exécutables produits par l'ancien code et la nouvelle traduction, l'axe des ordonnées suit une échelle logarithmique

5.4.2 Traduction sans redondances

Nous pouvons désormais observer les effets de l'élimination inconditionnelle des redondances. Tout d'abord, regardons les changements provoqués sur la durée d'exécution (figure 5.3) : les gains de performances par rapport à l'ancien code sont toujours présents, mais fortement atténués. Cela est dû, cette fois-ci, à l'exécution inconditionnelle de fragments de codes parfois inutiles.

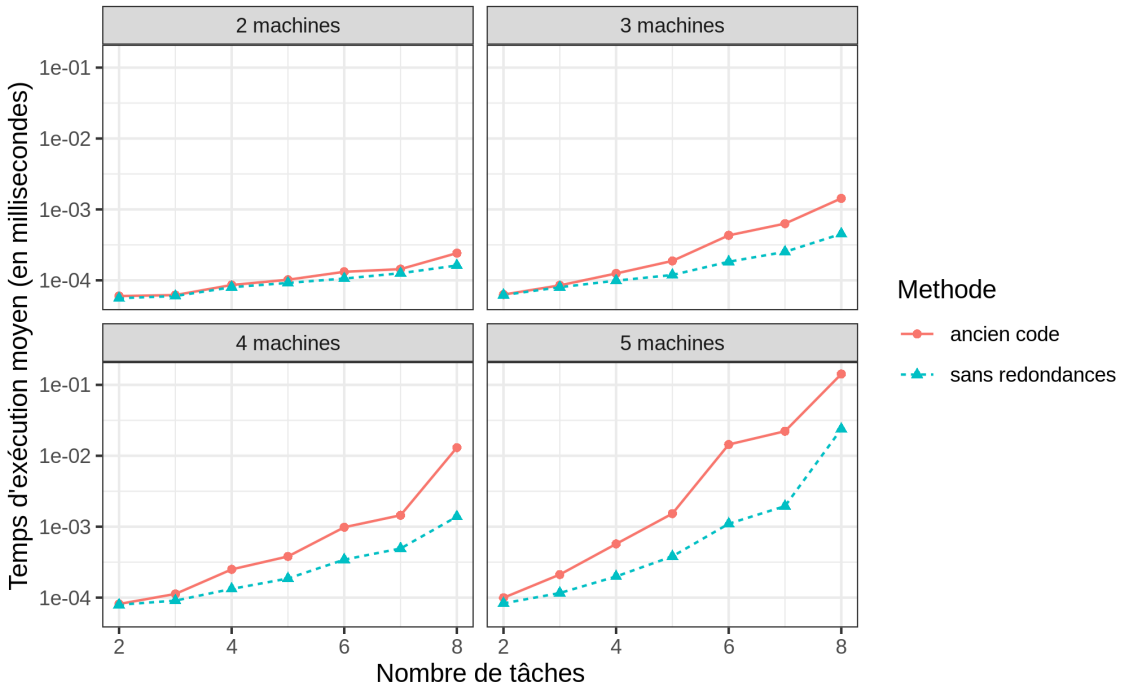

FIGURE 5.3 : Comparaison des temps de réponse en ligne de l'ancien code et de la nouvelle traduction sans redondances, l'axe des ordonnées suit une échelle logarithmique

Cependant, nous devrions observer une forte amélioration en ce qui concerne la taille des exécutables (figure 5.4) : en effet, l'élimination de redondances permet bien de réduire légèrement la taille du code par rapport aux anciennes techniques. Comparé aux ifs imbriqués naïvement, le gain est encore plus notable.

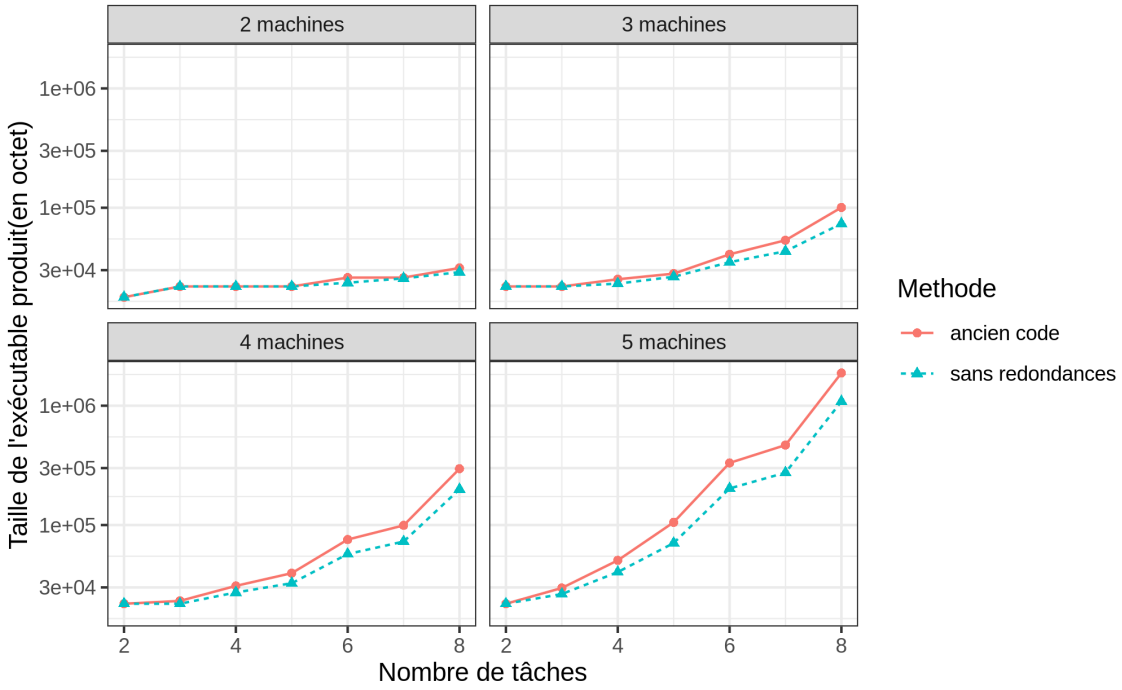

FIGURE 5.4 : Comparaison de la taille des exécutables produits par l'ancien code et la nouvelle traduction sans redondances, l'axe des ordonnées suit une échelle logarithmique

— 6 —
Conclusion

Pendant ce stage, un formalisme décrivant des problèmes d'attribution de tâches à des machines a été créé, puis traduit à la fois au format Heptagon et sous forme de programme linéaire et de programme par contraintes. Cela a permis de comparer ces différentes techniques sur un problème commun, intéressons-nous désormais aux résultats obtenus.

6.1 Comparaison des résultats avec les attentes

En premier lieu et conformément à ce qui était prévu, la synthèse de contrôleurs possède un temps de réponse en ligne plus faible que les méthodes d'ordonnancement utilisées sur les cas où elle est applicable. Dans le cadre d'un petit système ayant des contraintes de temps très fortes, la création d'un contrôleur via Heptagon/BZR reste donc bien une méthode avantageuse.

Toujours conformément aux attentes, les méthodes d'ordonnancement sont supérieures à Heptagon sur tous les autres points lorsqu'on les applique à ce type de problème générique. La taille du programme est stable plutôt qu'exponentielle, et nous pouvons appliquer les mêmes algorithmes à des cas beaucoup plus complexes plutôt que d'être restreint à des valeurs de n et m très faible.

En revanche, nous supposions que le temps d'exécution des programmes générés par Heptagon évoluerait de manière linéaire en fonction des dimensions de notre problème, et ce n'est finalement pas du tout le cas. En observant le code C produit, c'est finalement assez logique : dans le code des nœuds et du contrôleur, aucun "if" n'est imbriqué, et les branches "if" et "else" consistent toujours d'une seule instruction. La durée d'exécution est donc plus ou moins constante, quelles que soient les valeurs en entrée, et exponentielle en la taille du problème traité.

Cette croissance exponentielle indique que, même si nous arrivions à réaliser une synthèse de contrôleurs pour des cas de dimensions supérieures, il serait très probable que l'exécutable produit soit en pratique plus lent que les méthodes d'ordonnancement, perdant ainsi le seul avantage de la méthode.

Pour ce qui relève spécifiquement de l'ordonnancement, les techniques d'approximations d'ordonnancement de liste, qui avaient été initialement écartées, car elles ne possédaient pas de garanties pour les cas qui nous intéressent, se sont finalement révélées être très efficaces, répondant à la fois très rapidement par rapport aux algorithmes exacts et très fréquemment de manière positive lorsqu'il y a une solution. Enfin, le sous-ensemble de cas traitables par Heptagon/BZR est en fait tellement petit que de simples algorithmes naïfs parcourant l'ensemble

des configurations semblent être d'une rapidité comparable en ligne, sans le surcoût hors ligne monstrueux de la synthèse de contrôleurs.

6.2 Perspectives

La principale piste étudiée actuellement est l'amélioration des exécutables produits par Heptagon. Le problème semble provenir de la sortie de l'outil ReaX. Nous pensions en début de projet que le code généré à cette étape serait sous forme de cascade de condition, qui est la manière la plus intuitive de traduire un BDD (Binary decision diagram ou diagramme de décision binaire). Le code ne contient en fait aucun if imbriqué, et son temps d'exécution est, comme nous l'avons vu, linéairement dépendant de la taille du fichier, alors qu'il pourrait ne l'être que de manière logarithmique. L'objectif a donc été de "replier" ce code, afin de retrouver la structure de BDD originale et ainsi devenir capable de générer du code plus efficace. Une méthode s'en approchant est implémentée dans le chapitre 5, mais le code généré est redondant, répétant plusieurs fois certains calculs. Une piste d'amélioration serait alors de trouver un moyen efficace de calculer une seule fois la valeur de chaque variable.

De manière plus générale, il semble nécessaire d'optimiser davantage tous les processus de compilation d'Heptagon si l'on souhaite traiter des cas de dimensions plus grandes.

— A —
Annexe

A.1 Notation $\alpha|\beta|\gamma$

Cette notation, provenant d'une publication datant de 1979 [10], permet de classifier les problèmes d'optimisations consistant à attribuer des tâches à des machines. Elle se décompose en trois parties.

A.1.1 α : Caractéristiques des machines

Le premier champ, α, décrit le type de machine traité dans le problème. En se restreignant aux cas où les tâches ne sont pas découpées en sous-tâches. Dans ce cas, quatre types de problèmes peuvent être décrits, représentés par les symboles suivants :

- 1 signifie qu'une seule machine est présente
- P indique que toutes les machines sont identiques, ce qui signifie qu'une tâche i donnée ne possède qu'un seul temps d'exécution p_i, le même sur chaque machine.
- Q signifie que les machines sont dîtes "uniformes", soit que chaque machine i possède un coefficient de vitesse s_j. La durée d'exécution d'une tâche j sur la machine i est alors $\frac{p_i}{s_j}$
- R signifie que les machines sont indépendantes les unes des autres. Dans ce cas de figure, une tâche i donnée possède un temps d'exécution p_{ij} potentiellement différent pour chaque machine j, voir ne peut pas s'exécuter sur certaines machines.

À cela s'ajoute la possibilité de se limiter à un nombre de machines fixé. Ce nombre est alors indiqué après la lettre correspondant au type de machine. Par exemple, $Q3$ signifie que les machines étudiées sont uniformes, et au nombre de trois.

A.1.2 β : Caractéristiques des tâches

Le second champ, β, décrit les caractéristiques optionnelles des tâches que l'on souhaite ordonnancer. Nous en décrirons quelques-unes ici.

- *pmtn* qui signifie "preemption", indique que les tâches peuvent être mises en pause et redémarrées, potentiellement plus tard, sur une autre machine. Ce changement de contexte n'est pas supposé avoir de coût ici.
- *prec* qui signifie "precedence", donne une relation d'ordre sur les tâches, permettant de modéliser la nécessité d'exécuter certaines tâches avant de pouvoir en commencer une autre. Cette caractéristique peut être précisée par la forme de la relation d'ordre (par exemple "tree" pour un arbre).
- Des restrictions sur les valeurs des p_i (ou p_{ij} dans le cas où les machines seraient indépendantes). Par exemple, $p_i = 1$, qui signifie que tous les temps d'exécution sont unitaires, ou $p_i = p$ qui signifie que toutes les tâches possèdent le même temps d'exécution.
- r_i qui signifie que chaque tâche possède une "release date", une date avant laquelle la tâche ne peut pas être exécutée.
- d_i qui signifie que chaque tâche possède une "due date", une date avant laquelle la tâche devrait idéalement être totalement exécutée. Il est également possible d'indiquer que cette date est commune à toutes les tâches, avec $d_i = d$

Ces différentes caractéristiques peuvent être cumulées.

A.1.3 γ : Fonction Objectif

Le troisième champ, γ, permet de définir la fonction objectif que l'on cherche à minimiser. À noter que, pour un ordonnancement donné, C_i est la date de fin d'exécution pour la tâche i, et T_i son retard par rapport à une deadline d_i ($max(0, C_i - d_i)$). Voici quelques exemples de fonctions objectifs :

- C_{max}, qui consiste à minimiser le temps d'exécution global.
- $\sum C_j$, qui consiste à minimiser le temps de calcul utilisé.
- T_{max}, qui consiste à minimiser le retard par rapport aux deadlines des différentes tâches.

A.2 Représentation textuelle des problèmes

Il n'est pas nécessaire de représenter les tableaux Available et Active, puisqu'il s'agit de paramètres pris en compte en ligne. Pour modéliser un problème, il suffit alors d'un fichier texte, contenant dans l'ordre :

- Le nombre n de tâches sur une première ligne
- Le nombre m de ressources sur une seconde ligne
- $p_{1,1}$, $p_{1,2}$, ..., $p_{1,m}$
  ...
  $p_{n,1}$, $p_{n,2}$, ..., $p_{n,m}$ dans cet ordre sur n lignes.

Par exemple, pour un problème à trois tâches et deux ressources, on pourrait écrire :

```
3          # Nombre de taches
2          # Nombre de ressources
0.5   0.5  # Temps d'execution de la tache 1
0.25  0.5  # Temps d'execution de la tache 2
1     0.75 # Temps d'execution de la tache 3
```

A.3 Exemple d'extraction de sous-problème

Nous partons d'un problème avec 4 tâches et 3 ressources, décrit de la façon suivante :

```
# Nombre de taches
4
# Nombre de ressources
3
# Temps d'executions
0.5  0.4  0.3
0.5  0.7  0.6
0.1  0.3  0.2
0.6  0.6  0.5
```

Un fichier d'entrée (pour le programme Heptagon correspondant) désactivant la deuxième tâche et la troisième ressource ressemblerait à

```
1 0 1 1
1 1 0
```

Le sous-problème extrait serait :

```
# Nombre de taches
3
# Nombre de ressources
2
# Temps d'executions
0.5  0.4
0.1  0.3
0.6  0.6
```

A.4 Programme permettant la répétition des pas de l'exécutable Heptagon

Ce fichier est décomposé en trois parties. Premièrement, le module principal est importé, et les constantes sont déclarées.

```
open ModuleName
const nbsim: int = 50000
const nbtableau: int = nbsim-1
```

Comme évoqué plus haut, nbsim régente le nombre d'exécutions du nœud principal à effectuer à la suite. La constante réellement utilisée dans le reste du programme est nbtableau,

inférieure de 1 à nbsim car l'une des exécutions est traitée à part. Comme le programme est entièrement déterministe, nous ne souhaitons pas stocker plus d'une fois ses résultats. Nous avons donc créé une version silencieuse du nœud principal, nommée en conséquence mainSilent.

```
node mainSilent(active_1, active_2, ..., active_n,
available_1, available_2, ..., available_m :bool)
returns ()
var
   loc_1, loc_2, ..., loc_n : resource;
   objective: bool; nbRes: int;
let
   (loc_1, loc_2, ..., loc_n,
   objective, nbRes) =
   main(active_1, ..., active_n, available_1, ..., available_m);
tel
```

Tous les flux de sortie sont redirigés sur des variables déclarées localement, et rien n'est renvoyé par ce nœud. Il permet donc bien d'exécuter de manière silencieuse le nœud main. Finalement, le nœud principal "sim" de ce second module est implémenté. Celui-ci se comporte en apparence comme le nœud main (il possède les mêmes entrées et sorties), mais exécute nbsim fois chaque pas de manière séquentielle plutôt que d'effectuer une seule exécution.

```
node sim(active_1, active_2, ..., active_n,
available_1, available_2, ..., available_m :bool)
returns
   (loc_1, loc_2, ..., loc_n : resource;
   objective: bool; nbRes: int)
var
   t1, t2, ..., t(n+m): bool^nbtableau;
let
   (loc_1, loc_2, loc_3,
   objective, nbRes) =
   main(active_1, ..., active_n, available_1, ..., available_m);
   ti = active_i^nbtableau;
   t(n+j) = available_j^nbtableau;
() = map<<nbtableau>> mainSilent (
   t1, t2, ..., t(n+m));
tel
```

Les flux en entrée sont copiés nbtableau fois pour obtenir les entrées des différentes instances du nœud principal. Nous ne récupérons les sorties que d'une seule instance, et les autres nœuds sont exécutés de manière silencieuse. Le coût de création de ces tableaux devrait être négligeable par rapport au temps d'exécution du programme total. Il est à noter que la déclaration de tableaux se réalise à l'aide de l'opérande ^, le type *int*^10 correspondant par exemple aux tableaux d'entier de taille 10.

A.5 Formats des fichiers TSV

A.5.1 Format des mesures hors ligne

```
n  m  num   CompTime  SynthTime   HeptGenTime  HeptCompTime
CCompTime   CGenTime  CCompTimev2  NaiveTime
2  2  1  0.02  0.02  0.00  0.00  0.15  0.00  0.14  0.0000131130
```

```
4 2 2 2 0.02 0.02 0.00 0.00 0.15 0.00 0.14 0.0000140667
5 2 2 3 0.02 0.03 0.00 0.01 0.15 0.00 0.14 0.0000143051
```

Avec :

- *n* le nombre de tâches
- *m* le nombre de ressources
- *num* permet de différencier les différents problèmes de même dimensions
- *CompTime* la durée de compilation du code Heptagon initial
- *SynthTime* la durée de la synthèse de contrôleurs par ReaX
- *HeptGenTime* la durée de transformation du code du contrôleur écrit par ReaX en programme Heptagon
- *HeptCompTime* la durée de compilation du programme Heptagon correspondant au contrôleur
- *CCompTime* la durée de compilation du code final à l'aide de gcc
- *CGenTime* la durée de transformation du contrôleur en code C, pour la seconde version de l'exécutable Heptagon
- *CCompTimev*2 la durée de compilation via gcc du code amélioré
- *NaiveTime* la durée de l'énumération naïve des différentes configurations possibles.

A.5.2 Format des mesures de performances

```
1 n m num nbUnavailable HeptagonTimeFor50000 HeptagonV2TimeFor50000
2 SchedulingTimeMilpFor10 SchedulingTimeConsFor10 basicTimeFor10 naiveTimeFor10
3 2 2 1 0 0.003 0.002 .0383937358856201172 .0294110774993896483 .0000319480 .0000100138
4 2 2 2 0 0.003 0.003 .0408782958984375012 .0321424007415771487 .0000307557 .0000092986
5 2 2 3 0 0.003 0.002 .0389099121093749997 .0298140048980712892 .0000340938 .0000104906
```

Avec :

- *n* le nombre de tâches
- *m* le nombre de ressources
- *num* permet de différencier les différents problèmes de même dimensions
- *nbUnavailable* correspond au nombre de ressources désactivées dans le sous-problème
- *HeptagonTimeFor*50000 correspond au temps d'exécution en secondes de 50000 pas du programme généré par Heptagon
- *HeptagonV2TimeFor*50000 correspond au temps d'exécution en secondes de 50000 pas du programme Heptagon utilisant le code C généré directement à partir du contrôleur.

- *SchedulingTimeMilpFor*10 correspond au temps nécessaire en secondes pour résoudre 10 fois le sous-problème à l'aide de CBC

- *SchedulingTimeConsFor*10 correspond au temps nécessaire en secondes pour résoudre 10 fois le sous-problème à l'aide de CP-SAT

- *basicTimeFor*10 correspond au temps nécessaire pour trouver un ordonnancement par énumération, en tentant de placer toutes les tâches sur deux machines quelconques.

- *naiveTimeFor*10 correspond au temps nécessaire pour retrouver l'ordonnancement calculé de manière naïve précédemment.

A.5.3 Format des mesures de tailles de fichiers

```
1 n  m  num   problem_file_size  program_file_size  program_file_size_v2
2 2  2  1  145    17800
3 2  2  2  145    17800
4 2  2  3  145    17800
```

Avec :

- *n* le nombre de tâches

- *m* le nombre de ressources

- *num* permet de différencier les différents problèmes de même dimensions

- *problem_file_size* correspond à la taille en octet du fichier décrivant le problème

- *program_file_size* correspond à la taille en octet de l'exécutable produit par Heptagon

A.6 Temps d'exécution en ligne du programme Heptagon, en prenant en compte les machines désactivées

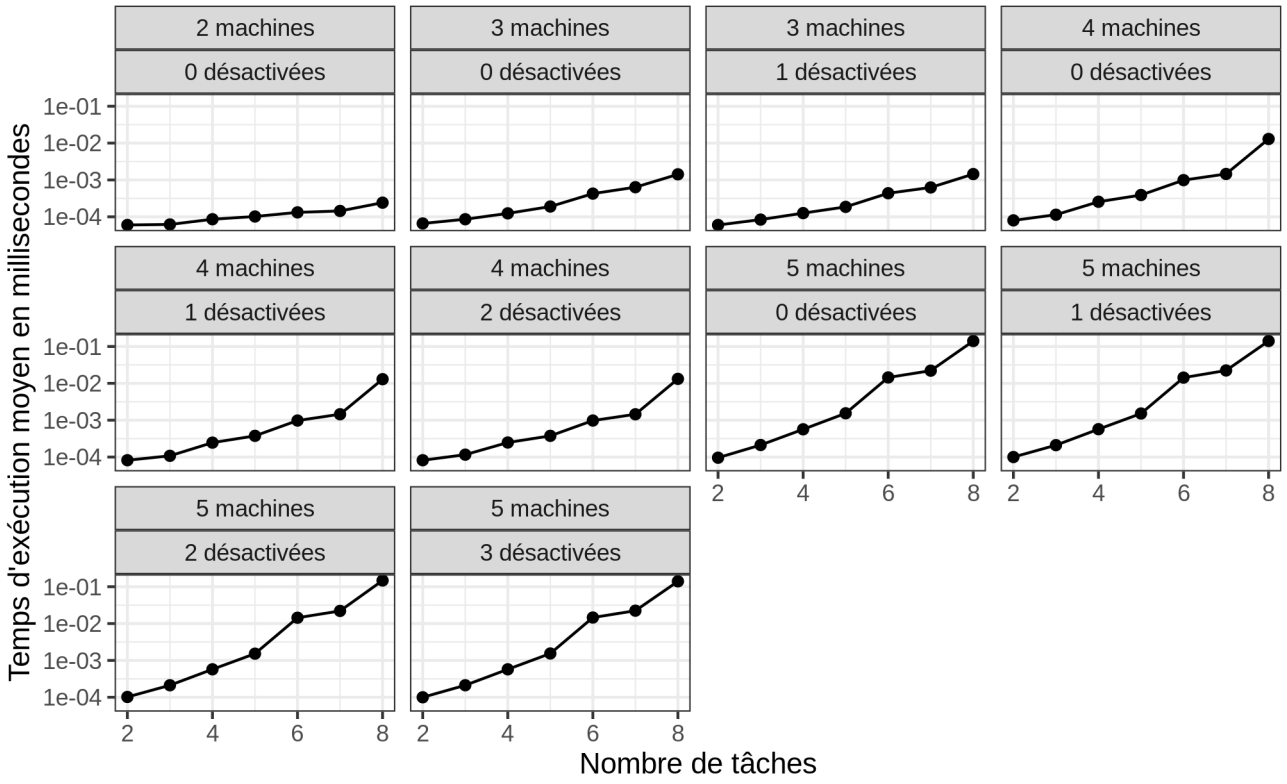

FIGURE A.1 : Évolution du temps d'ordonnancement en ligne en proportion

A.7 Régression : Durée d'exécution en fonction de la taille de l'exécutable

Voici les résultats de la régression :

```
Call:
lm(formula = RuntimeTime ~ size, data = comparison_data_2)

Residuals:
      Min        1Q    Median        3Q       Max
-0.016361 -0.001575  0.001579  0.002475  0.035186

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -4.065e-03  1.030e-04  -39.47   <2e-16 ***
size         7.539e-08  2.218e-10  339.94   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.004306 on 2098 degrees of freedom
Multiple R-squared:  0.9822,    Adjusted R-squared:  0.9822
F-statistic: 1.156e+05 on 1 and 2098 DF,  p-value: < 2.2e-16
```

La proportion de variance expliquée par la régression est extrêmement élevée (R^2 à 0.98), et la p-value est extrêmement faible. Il semble certain que le temps d'exécution des programmes générés par Heptagon dépend linéairement de la taille de leur code.

A.8 Régression : Taille de l'exécutable en fonction du temps de compilation

Voici les résultats de la régression :

```
Call:
lm(formula = RuntimeTime ~ CompilTime, data = comparison_data)

Residuals:
      Min         1Q     Median         3Q        Max
-0.116931 -0.003298 -0.003162 -0.002473  0.122637

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 3.371e-03  3.670e-04   9.185   <2e-16 ***
CompilTime  1.252e-04  1.604e-06  78.050   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.01632 on 2098 degrees of freedom
Multiple R-squared:  0.7438,    Adjusted R-squared:  0.7437
F-statistic:  6092 on 1 and 2098 DF,  p-value: < 2.2e-16
```

On peut noter que la proportion de variance expliquée est assez élevée (R^2 à 0.75), et que la p-value est extrêmement faible. Il semble bien y avoir une corrélation entre ces deux mesures, même si elle n'est pas nécessairement linéaire.

Bibliographie

[1] Daniel BALOUEK, Alexandra CARPEN AMARIE, Ghislain CHARRIER, Frédéric DESPREZ, Emmanuel JEANNOT, Emmanuel JEANVOINE, Adrien LÈBRE, David MARGERY, Nicolas NICLAUSSE, Lucas NUSSBAUM, Olivier RICHARD, Christian PÉREZ, Flavien QUESNEL, Cyril ROHR et Luc SARZYNIEC. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In : *Cloud Computing and Services Science*. Sous la dir. d'Ivan I. IVANOV, Marten van SINDEREN, Frank LEYMANN et Tony SHAN. T. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, p. 3-20. ISBN : 978-3-319-04518-4. DOI : 10.1007/978-3-319-04519-1_1.

[2] Jean-Louis BERGERAND. "LUSTRE : un langage déclaratif pour le temps réel". Theses. Institut National Polytechnique de Grenoble - INPG, 1986-01. URL : https://tel.archives-ouvertes.fr/tel-00320006.

[3] Ksenia BESTUZHEVA, Mathieu BESANÇON, Wei-Kun CHEN, Antonia CHMIELA, Tim DONKIEWICZ, Jasper van DOORNMALEN, Leon EIFLER, Oliver GAUL, Gerald GAMRATH, Ambros GLEIXNER, Leona GOTTWALD, Christoph GRACZYK, Katrin HALBIG, Alexander HOEN, Christopher HOJNY, Rolf van der HULST, Thorsten KOCH, Marco LÜBBECKE, Stephen J. MAHER, Frederic MATTER, Erik MÜHMER, Benjamin MÜLLER, Marc E. PFETSCH, Daniel REHFELDT, Steffan SCHLEIN, Franziska SCHLÖSSER, Felipe SERRANO, Yuji SHINANO, Boro SOFRANAC, Mark TURNER, Stefan VIGERSKE, Fabian WEGSCHEIDER, Philipp WELLNER, Dieter WENINGER et Jakob WITZIG. *The SCIP Optimization Suite 8.0*. Technical Report. Optimization Online, 2021-12. URL : http://www.optimization-online.org/DB_HTML/2021/12/8728.html.

[4] Ksenia BESTUZHEVA, Mathieu BESANÇON, Wei-Kun CHEN, Antonia CHMIELA, Tim DONKIEWICZ, Jasper van DOORNMALEN, Leon EIFLER, Oliver GAUL, Gerald GAMRATH, Ambros GLEIXNER, Leona GOTTWALD, Christoph GRACZYK, Katrin HALBIG, Alexander HOEN, Christopher HOJNY, Rolf van der HULST, Thorsten KOCH, Marco LÜBBECKE, Stephen J. MAHER, Frederic MATTER, Erik MÜHMER, Benjamin MÜLLER, Marc E. PFETSCH, Daniel REHFELDT, Steffan SCHLEIN, Franziska SCHLÖSSER, Felipe SERRANO, Yuji SHINANO, Boro SOFRANAC, Mark TURNER, Stefan VIGERSKE, Fabian WEGSCHEIDER, Philipp WELLNER, Dieter WENINGER et Jakob WITZIG. *The SCIP Optimization Suite 8.0*. ZIB-Report 21-41. Zuse Institute Berlin, 2021-12. URL : http://nbn-resolving.de/urn:nbn:de:0297-zib-85309.

[5] Peter BRUCKER, Christoph DÜRR, Sven JÄGER, Sigrid KNUST, Damien PROT, Rob van STEE et Óscar C. VÁSQUEZ. *Scheduling Zoo*. [Online ; accessed 12-April-2022]. URL : http://schedulingzoo.lip6.fr/.

[6] Louis-Claude CANON et Laurent PHILIPPE. "On the Heterogeneity Bias of Cost Matrices for Assessing Scheduling Algorithms". In : *IEEE Transactions on Parallel and Distributed Systems* 28.6 (2017), p. 1675-1688. DOI : 10.1109/TPDS.2016.2629503. URL : http://lccanon.free.fr/publications/RR-FEMTO-ST-8663-2015.pdf.

[7] Gwenaël DELAVAL, Ayan HORE, Stéphane MOCANU, Lucie MULLER et Eric RUTTEN. "Discrete Control of Response for Cybersecurity in Industrial Control". In : *IFAC 2020 - IFAC World Congress 2020*. Proc. of the 21st IFAC World Congress. Berlin, Germany, 2020-07, p. 1-8. URL : https://hal.archives-ouvertes.fr/hal-02569406.

[8] John FORREST, Ted RALPHS, Haroldo Gambini SANTOS, Stefan VIGERSKE, Lou HAFER, John FORREST, Bjarni KRISTJANSSON, JPFASANO, EDWINSTRAVER, Miles LUBIN, RLOUGEE, JPGONCAL1, JAN-WILLEM, H-I-GASSMANN, Samuel BRITO, CRISTINA, Matthew SALTZMAN, TOSTTOST, Fumiaki MATSUSHIMA et TO-ST. *coin-or/Cbc : Release releases/2.10.7*. Version releases/2.10.7. 2022-01. DOI : 10.5281/zenodo.5904374. URL : https://doi.org/10.5281/zenodo.5904374.

[9] Michael R GAREY et David S JOHNSON. ""strong"np-completeness results : Motivation, examples, and implications". In : *Journal of the ACM (JACM)* 25.3 (1978), p. 499-508.

[10] R.L. GRAHAM, E.L. LAWLER, J.K. LENSTRA et A.H.G.Rinnooy KAN. "Optimization and Approximation in Deterministic Sequencing and Scheduling : a Survey". In : *Discrete Optimization II*. Sous la dir. de P.L. HAMMER, E.L. JOHNSON et B.H. KORTE. T. 5. Annals of Discrete Mathematics. Elsevier, 1979, p. 287-326. DOI : https://doi.org/10.1016/S0167-5060(08)70356-X. URL : https://www.sciencedirect.com/science/article/pii/S016750600870356X.

[11] Soguy Mak-Karé GUEYE, Gwenaël DELAVAL, Eric RUTTEN, Dominique HELLER et Jean-Philippe DIGUET. "A Domain-specific Language for Autonomic Managers in FPGA Reconfigurable Architectures". In : *ICAC 2018 - 15th IEEE International Conference on Autonomic Computing*. Trento, Italy : IEEE, 2018-09, p. 1-10. URL : https://hal.archives-ouvertes.fr/hal-01868675.

[12] Dorit S. HOCHBAUM et David B. SHMOYS. "Using Dual Approximation Algorithms for Scheduling Problems : Theoretical and Practical Results". In : *Journal of the ACM* 34.1 (1987-01), p. 144-162. ISSN : 0004-5411. DOI : 10.1145/7531.7535.

[13] Q. HUANGFU et J. A. J. HALL. *Parallelizing the dual revised simplex method*. 2015. DOI : 10.48550/ARXIV.1503.01889. URL : https://arxiv.org/abs/1503.01889.

[14] N. KARMARKAR. "A New Polynomial-Time Algorithm for Linear Programming". In : *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*. STOC '84. New York, NY, USA : Association for Computing Machinery, 1984, p. 302-311. ISBN : 0897911334. DOI : 10.1145/800057.808695. URL : https://doi.org/10.1145/800057.808695.

[15] Jan Karel LENSTRA, David B. SHMOYS et Eva TARDOS. "Approximation algorithms for scheduling unrelated parallel machines". In : *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*. 1987, p. 217-224. DOI : 10.1109/SFCS.1987.8.

[16] Harry R. LEWIS. “Michael R. Garey and David S. Johnson. Computers and intractability. A guide to the theory of NP-completeness. W. H. Freeman and Company, San Francisco1979, x 338 pp.” In : *Journal of Symbolic Logic* 48 (1983), p. 238.

[17] *Manuel Heptagon*. [Online ; accessed 11-April-2022]. URL : http://bzr.inria.fr/pub/heptagon-manual.pdf.

[18] Stuart MITCHELL, Michael J. O’SULLIVAN et Iain DUNNING. “PuLP : A Linear Programming Toolkit for Python”. In : 2011.

[19] Laurent PERRON et Vincent FURNON. *OR-Tools*. Version 7.2. Google, 2019-07-19. URL : https://developers.google.com/optimization/.

[20] O. TANGE. “GNU Parallel—The Command-Line Power Tool”. In : *;login : The USENIX Magazine* 36.1 (2011-02), p. 42-47. URL : https://www.usenix.org/publications/login/february-2011-volume-36-number-1/gnu-parallel-command-line-power-tool.