Prédiction et augmentation de ressources à la rescousse de l'algorithmique

Vincent Fagnon

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Mathématiques et Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Prédiction et augmentation de ressources à la rescousse de l'algorithmique

Prediction and resource augmentation come to the rescue of algorithmics.

Présentée par :

Vincent FAGNON

Direction de thèse :

Denis TRYSTRAM Professeur des Universités, Grenoble INP	Directeur de thèse
Giorgio LUCARELLI Enseignant-Chercheur, Université de Lorraine	Co-encadrant de thèse

Rapporteurs :

JOHANNE COHEN
Directeur de recherche, CNRS DELEGATION ILE-DE-FRANCE SUD
CHRISTOPH DURR
Directeur de recherche, CNRS DELEGATION PARIS CENTRE

Thèse soutenue publiquement le **6 juillet 2023**, devant le jury composé de :

JOHANNE COHEN Directeur de recherche, CNRS DELEGATION ILE-DE-FRANCE SUD	Rapporteure
MORITZ MUHLENTHALER Maître de conférences, GRENOBLE INP	Examinateur
CHRISTOPH DURR Directeur de recherche, CNRS DELEGATION PARIS CENTRE	Rapporteur
YVES ROBERT Professeur des Universités, ENS DE LYON	Président
JEAN-CHARLES BILLAUT Professeur des Universités, UNIVERSITE DE TOURS	Examinateur
KIM THANG NGUYEN Professeur des Universités, GRENOBLE INP	Examinateur

Invités :

GIORGIO LUCARELLI
Maître de conférences, UNIVERSITE DE LORRAINE
DENIS TRYSTRAM
Professeur des Universités, GRENOBLE INP

> *Mathematician is a machine which turns coffee into theorems.*

— **Alfréd Rényi**

Remerciements

D'abord et avant tout, un énorme merci à mon encadrant et mon directeur, qui ont, une fois de plus, accompagné un doctorant dans cette expérience humaine et holistique. Merci de m'avoir soutenu tout au long de cette aventure et de m'avoir supporté. Ces visioconférences matinales constructives, agrémentées du chant des poules en arrière-plan, ainsi que votre disponibilité exceptionnelle resteront en particulier gravées dans ma mémoire.

Merci de m'avoir lancé au cœur de groupes d'étudiants auxquels j'ai eu le plaisir d'enseigner tout au long de ces années. J'espère sincèrement qu'ils garderont un souvenir aussi mémorable que le mien de ces moments partagés, ou du moins qu'ils en garderont un souvenir tout court ! Merci de m'avoir poussé (parfois traîné, tiré,... avec ou sans mon consentement) hors de ma zone de confort pour enseigner dans des domaines qui m'étaient inconnus jusqu'alors ! Quelle idée, vous êtes fous. En parlant de folie, merci au conformisme du lundi, ou à la dissidence du vendredi ? On ne sait pas, on ne sait plus.

Merci à toute l'équipe DATA-POL, que ce soit pour les discussions autour d'un tableau blanc ou les moments détentes à la cafétéria ! Tout ça a grandement contribué à l'ambiance chaleureuse qui règne au labo. Un merci tout particulier pour la gestion impeccable de l'équipe, dans l'ombre, qui permet à la boutique de continuer à tourner.

Un merci spécial pour toutes les interventions salvatrices ; les sauvetages d'ordinateur, les installations en panique, les tutoriels expliqués pour les débutants. Finalement, c'est Debian que j'aurais su supporter le plus longtemps et j'ai même appris à apprécier Git (et il y a, bien sûr, une nuance fondamentale entre apprécier et comprendre) !

Merci à la coinche du midi et à son écosystème de diablotins, qui, sous couvert d'étudier le coût de l'information en théorie des jeux, permettent à tous les curieux de s'intégrer à l'équipe ! Et comment ne pas mentionner la coinche du soir !

Merci aussi aux les collègues des autres laboratoires pour leur accueil chaleureux, les échanges fructueux et les discussions passionnantes sur des problèmes théoriques qui m'étaient jusqu'alors inconnus. Je me réjouis à l'idée de vous rendre visite à l'occasion, et de marier, peut-être un jour, pralines, cannelés, noix et mirabelles.

Enfin, je souhaite adresser une mention spéciale de remerciements aux personnes qui se reconnaîtront dans ces propos : des enfants qui gribouillent toujours sur les murs, les meilleurs plans houmous de la région, le mouton sans lumière, la découverte du café, l'informagie, le Timanoix, le 61978, les trajets japonais, le rattrapage identique à l'exam, les mozza sticks, le sauvetage avec draw.io, le paprika, la tasse moche, le Luxembourg, les slides des skis, les IPA au pays du caramel, les deux p'tits monstres et toute la famille, mon (seul et unique) merge request, et enfin, à Bernoull*i* qui devrait faire des maths encore plus discrètes.

Ce travail est soutenu par le projet Energumen (ANR-18-CE25-0008).

Abstract

Currently, the trend is towards the pooling of IT resources. This centralization offers several advantages, including better management of workload variations, avoidance of extreme usage peaks, and easier optimization of calculations since data is on-site. However, the advent of the Internet of Things (IoT) has created an increase in the need for processing and analyzing data at locations far from data centers. For faster and more efficient processing, it is important to limit data transfers and process them near their place of production. Thus, it is essential to find a balance between using centralized platforms to pool computing resources and decentralized models to process data locally.

This thesis focuses on the implementation of several mechanisms aimed at designing scheduling algorithms for multiple agents with distinct objectives on a platform composed of computing resources.

First, we investigate the use of an external mechanism (oracle) to reveal uncertainties related to the execution times of sequential tasks to be executed on a computing resource. We propose algorithms that guarantee optimal average performance on the classic sum flowtime objective and study their performance, robustness, and the relevance of the oracle model on average through a simulation campaign. We also question the effectiveness of such a model if one wishes to guarantee good performance in the worst-case scenario.

Next, we study a bi-agent problem in which each agent is subject to different constraints and objectives. We propose an algorithm implementing resource augmentation (speed and rejection) and prove, using the dual fitting technique, a competitiveness ratio depending, among other things, on these augmentation parameters.

Finally, we address a more classic problem of scheduling a sequential task application for which we want to minimize the makespan on a hybrid multicores (CPU/GPU) platform. We propose an algorithm based on the study of the integrality gap of the relaxation of a linear programming, algorithm for which we provide a proof of performance guarantee in the form of a constant approximation ratio.

Résumé

Actuellement, la tendance est à la mutualisation des ressources informatiques. Cette centralisation offre plusieurs avantages, notamment une meilleure gestion des variations de charge de travail, l'évitement de pics d'utilisation extrêmes et une optimisation simplifiée des calculs avec les données déjà en place. Cependant, l'avènement de l'Internet des objets (IoT) a créé une augmentation des besoins de traitement et d'analyse de données à des endroits éloignés des data-centers. Pour un traitement plus rapide et efficace, il est important de limiter les transferts de données et de les traiter près de leur lieu de production. Ainsi, il est essentiel de trouver un équilibre entre l'utilisation de plateformes centralisées pour mutualiser les ressources de calcul et les modèles décentralisés pour traiter les données localement.

Cette thèse porte sur la mise en œuvre de plusieurs mécanismes visant à concevoir des algorithmes d'ordonnancement pour plusieurs agents ayant des objectifs distincts sur une plateforme composée de ressources de calcul.

Tout d'abord, nous nous interrogeons sur l'utilisation d'un mécanisme externe (oracle) pour révéler les incertitudes relatives aux durées d'exécution de tâches séquentielles à exécuter sur une ressource de calcul. Nous proposons des algorithmes qui garantissent des performances moyennes optimales sur l'objectif classique du "sum flowtime" et étudions, via une campagne de simulations, leurs performances, leurs robustesses et la pertinence du modèle d'oracle en moyenne. Nous nous interrogeons également sur l'efficacité d'un tel modèle si l'on souhaite garantir de bonnes performances au pire des cas.

Ensuite, nous étudions un problème bi-agent dans lequel chaque agent est soumis à des contraintes et des objectifs différents. Nous proposons un algorithme mettant en œuvre de l'augmentation de ressources (vitesse et rejet) et prouvons, grâce à la technique du dual fitting, un rapport de compétitivité dépendant, entre autres, de ces paramètres d'augmentation.

Enfin, nous abordons un problème plus classique d'ordonnancement d'une application de tâches séquentielles pour laquelle nous souhaitons minimiser le "makespan" sur une plateforme hybride multicœurs (CPU/GPU). Nous proposons un algorithme basé sur l'étude de "l'intégrality gap" de la relaxation d'une programmation linéaire, algorithme pour lequel nous donnons une preuve de garantie de performances sous la forme d'un rapport d'approximation constant.

Table des matières

1 Introduction

On observe actuellement une tendance à la mutualisation, notamment des ressources dans le domaine informatique, témoignant d'une prise de conscience croissante au sein de la population du partage de ressources (voitures partagées, vélos partagées, AirBnB, covoiturage...). L'essor de l'*open science* en est un exemple, mais cela va encore plus loin avec l'émergence de plateformes collaboratives. Parmi les illustrations de cette tendance, on peut citer la mutualisation de plateformes de calculs telles que Grid5000, ainsi que la popularité croissante de projets de cloud computing communautaires (Amadeus, Cmed...).

Cette mutualisation facilite l'optimisation de l'utilisation des ressources et potentiellement la réduction de l'impact environnemental, car elle permet de mieux gérer les variations de charge de travail et d'éviter les pics d'utilisation extrêmes qui conduisent souvent à un surdimensionnement des centres de calcul. Le *PUE* (*Power Usage Effectiveness* [AA12]) est un indicateur d'efficacité énergétique des centres de calcul qui mesure le rapport entre la consommation totale d'énergie d'une plateforme et sa consommation d'énergie dédiée uniquement aux ressources de calculs. La centralisation des ressources constitue actuellement un moyen efficace d'atteindre des PUE satisfaisants (proches de 1).

De plus, en ordonnançant des applications de tâches sur des plateformes centralisées, l'optimisation est plus simple car les données sont sur place, ce qui permet de réduire le temps de communication par rapport aux modèles décentralisés. Ainsi, la mutualisation des systèmes de calculs contribue à maximiser l'utilisation des machines construites et à adapter la taille des plateformes à leur utilisation réelle moyenne.

Néanmoins, l'avènement de l'Internet des objets (IoT) a engendré une hausse des besoins en matière de traitement et d'analyse des données à des endroits éloignés des centres de données. Les données produites par les capteurs nécessitent souvent un traitement immédiat et leur analyse doit être effectuée localement, à proximité des capteurs [Ngu+21]. La plupart de ces données n'ont qu'un intérêt dans l'environnement local où elles sont produites et leur durée de vie est courte. Ainsi, elles doivent être analysées immédiatement.

Ces nouveaux besoins en matière de traitement de données ont mis en lumière les limites des systèmes informatiques traditionnels et ont conduit à une augmentation du déploiement de capacités de calcul en périphérie d'Internet pour répondre à ces exigences. L'*Edge* et le *Fog Computing* en sont des exemples. Ils permettent de traiter les données plus proche de leur source, c'est-à-dire au plus près des objets connectés ou des capteurs, et de limiter les transferts de données vers les centres de données ; pour un traitement plus rapide et plus efficace.

Les approches centralisées et décentralisées sont toutes deux pertinentes. Toutefois, dans un contexte où la réutilisation des ressources est privilégiée plutôt que la construction de nouvelles infrastructures matérielles dédiées à une application, trouver un juste équilibre entre l'utilisation de plateformes centralisées pour mutualiser les ressources de calcul et les modèles décentralisés pour traiter les données localement est un défi de taille.

1.1 Un modèle intermédiaire

Pour réussir à gérer efficacement une plateforme composée de ressources centralisées et décentralisées, il est essentiel de développer des algorithmes de planification de tâches (d'*ordonnancement*) qui prennent en compte de nombreux aspects. Les usages des plateformes peuvent évoluer au fil du temps et s'éloigner de leur cas d'usage initial. Les éventuels changements ou divergences dans les besoins des utilisateurs, l'arrivée inattendue de nouveaux utilisateurs, ainsi que l'hétérogénéité des ressources disponibles, doivent être pris en compte. De plus, ces algorithmes doivent garantir de bonnes performances pour chaque application traitée et être robustes aux situations imprévues pour être utilisés efficacement dans la gestion de ces plateformes mutualisées.

Nous présentons un problème général d'ordonnancement bi-agent sur une plateforme de calcul dans la Section (1.3). Nous décomposons ce problème en plusieurs sous-problèmes plus spécifiques que nous introduisons tout d'abord brièvement dans cette introduction. Chaque chapitre de cette thèse aborde en détail chacun de ces sous-problèmes d'ordonnancement. Nous y proposons des algorithmes dont nous évaluerons leurs performances en proposant, selon les problèmes, tant des analyses théoriques (rapports d'approximation, efficacités en moyenne, preuves d'optimalité, règles de dominance) que des évaluations pratiques tournées vers l'étude de l'efficacité moyenne ou vers la robustesse de ces algorithmes.

Bien que les problèmes que nous étudions soient difficiles (au sens NP-difficile du terme), la résolution de ceux-ci peut parfois être simple pour la quasi-totalité des instances, et la difficulté réside dans quelques cas spécifiques (que nous appellerons "pathologiques") qui ne se présentent en pratique qu'avec une probabilité quasi-nulle. On définit ici un "cas pathologique" comme étant un pire cas, ou une instance du problème qui contiendrait un *motif* caractéristique d'un pire cas. Nous proposons des mécanismes pour chacun des problèmes dont nous évaluons les avantages et les limites pour aider à la conception de politiques d'ordonnancement appropriées sur ces plateformes de calcul partagées.

1.2 Mécanismes pour contrer les cas pathologiques

> *Comment allouer les ressources pour plusieurs applications ayant des objectifs divergents dans un contexte où il est aujourd'hui impossible de trouver un algorithme performant pour une seule de ces applications ?*

Nous examinons plusieurs problèmes qui sont intrinsèquement complexes, et pour certains d'entre eux (présentés dans les chapitres 2 et 3), il n'existe aucun algorithme connu qui puisse les résoudre avec un rapport d'approximation (*approximation ratio*) à facteur constant. Nous nous intéressons particulièrement à ce rapport qui mesure la qualité de la solution produite par l'algorithme en comparant son résultat avec la solution optimale, dans le pire des cas. Pour Π l'ensemble des instances I d'un problème de minimisation, le rapport d'approximation ρ peut être défini comme (voir [Hoc97]) :

$$\max_{I \in \Pi} \left\{ \frac{\text{résultat de l'algorithme } (I)}{\text{résultat optimal } (I)} \right\} \leq \rho$$

On dira pour un problème *offline* que toutes les tâches ainsi que leurs caractéristiques (p_j, r_j, ...) sont connues avant le début de la planification. En revanche un problème *online* est un problème pour lequel on ne dispose pas entièrement de l'information concernant les entités à traiter au début de l'exécution. Comme il est possible de découvrir de nouvelles tâches à traiter au cours de l'exécution, les décisions prises à l'instant t=0 par l'algorithme (par exemple) le sont sans avoir entièrement connaissance de l'instance qu'il est en train de traiter. On ne parle alors plus de

rapport d'approximation pour évaluer les performances de nos algorithmes mais de rapport de compétitivité ρ tel que :

$$\max_{I \in \Pi} \left\{ \frac{\text{résultat de l'algorithme online } (I)}{\text{résultat optimal offline } (I)} \right\} \leq \rho$$

Même si de nombreux problèmes sont considérés comme difficiles, des heuristiques simples permettent souvent d'obtenir de très bons résultats en pratique, les cas pathologiques ne se présentant que très rarement. Un des objectifs de cette thèse est de réussir à mettre le doigt sur ce qui rend un problème structurellement difficile, sur ce qui caractérise ces cas pathologiques, et de s'intéresser à des mécanismes pour faire baisser les bornes induites par ces pires cas.

En ce qui nous concerne, nous proposons des révisions des modélisations de certains problèmes pour donner des libertés ou des informations aux algorithmes que nous proposons, de façon à obtenir des résultats proches de ceux obtenus par un algorithme optimal qui ne bénéficierait pas de ces modifications), dans la lignée des idées développées par Tim Roughgarden dans *Beyond the Worst-Case Analysis of Algorithms* [Rou20] (particulièrement dans le chapitre 4). Nous nous penchons sur différentes techniques d'augmentation de ressources (*resource augmentation*) ainsi que sur l'utilisation d'oracles pour pallier aux cas pathologiques qui rendent ces problèmes structurellement difficiles.

L'augmentation de ressources (en tant que tel) est un sujet relativement récent dont la communauté scientifique s'est particulièrement emparé depuis les années 2000, qui peut prendre différentes formes. B. Kalyanasundaram et K. Pruhs décrivent le mécanisme d'augmentation de vitesse (*speed augmentation*) dans leur article intitulé *Speed is as powerful as clairvoyance* ([KP00]), l'augmentation du nombre de machines est proposée par C.A. Phillips *et al.* dans *Optimal time-critical scheduling via resource augmentation* ([Phi+97]), et le rejet de certains jobs est évoqué dans des travaux plus récents (2018) de A.R. Choudhury *et al.* : *Rejecting jobs to minimize load and maximum flow-time* ([Cho+18]).

Nous envisageons l'augmentation des ressources comme un outil théorique pour renforcer nos algorithmes et garantir des performances améliorées. Bien que cette approche puisse sembler artificielle, le gain en performances qu'elle entraîne peut dans certains cas modifier totalement notre point de vue sur le problème dans son ensemble. Par exemple :

— L'augmentation de la quantité de ressources pour l'ordonnancement peut s'assimiler au recours à des travailleurs temporaires (intérimaires) qui est une

pratique courante dans le monde des entreprises (depuis les années 50 en France).

— L'augmentation de vitesse des ressources de calcul est couramment utilisée en pratique : le cas du surcadençage (*overclocking*) d'une ressource de calcul.

— Le rejet de tâches ressemble à la minimisation le nombre de tâches en retard, indépendamment de la durée de leur retard.

A titre d'exemple, la mémoire cache aurait pu être qualifiée d'augmentation de ressources très artificielle (un modèle théorique où le temps de communication pour ré-accéder à une donnée "récemment utilisée" est quasi nul) ; mais est aujourd'hui démocratisée auprès du grand public (depuis les années 1968, sur la série IBM 360 [Lip68]).

Pour tenir compte de l'augmentation de ressources, nous pouvons le redéfinir le rapport compétitif de la façon suivante, en comparant le résultat de l'algorithme online qui dispose de ressources augmentées avec l'optimal offline qui n'en dispose pas. L'objectif est alors de concevoir des algorithmes qui garantissent un rapport-compétitif ρ minimal, tel que :

$$\max_{I \in \Pi} \left\{ \frac{\text{résultat de l'algorithme online augmenté } (I)}{\text{résultat optimal offline non augmenté}(I)} \right\} \leq \rho$$

Bien que cette comparaison oppose les algorithmes à de très fortes bornes, les libertés supplémentaires qui peuvent lui être accordées peuvent permettre de pallier l'écart entre online et offline. On peut remarquer qu'avec cette définition, il n'est pas impossible que ρ puisse être inférieur à 1, selon la puissance supplémentaire donnée à l'algorithme étudié.

Nous utilisons un modèle d'oracle qui peut être considéré comme une boite noire qui permet de révéler des données incertaines : ici le temps d'exécution d'une tâche testée. Le problème de l'ordonnancement sur une seule machine utilisant l'*incertitude explorable* a été introduit dans la théorie de l'ordonnancement par Dürr *et al.* [Dür+18]. L'utilisation d'un oracle n'entraîne pas directement de pénalité dans la fonction objectif, mais chaque appel à cet oracle est assimilé à une tâche "test" avec un temps de traitement à exécuter. Ainsi un appel à l'oracle révèle des incertitudes mais dégrade également l'ordonnancement par l'algorithme. Ce fonctionnement d'oracle a déjà été étudié par Dufossé *et al.* [Duf+22] ainsi que par Bampis *et al.* [Bam+21].

Cette thèse se concentre sur l'étude de différents problèmes pour lesquels nous proposons des mécanismes tels que l'utilisation d'un modèle de prédiction pour obtenir des informations supplémentaires sur les tâches et explorerons diverses

facettes de l'augmentation de ressources, en utilisant des techniques pour augmenter la vitesse des machines et autoriser le rejet de certaines tâches. Nous présentons des algorithmes qui s'appuient sur des méthodes d'optimisation combinatoire et de programmation linéaire pour améliorer les performances des systèmes.

1.3 Une modélisation bi-agent en bordure de réseau

Étant donné notre volonté de mutualiser les ressources, nous nous concentrons sur un modèle bi-agent. Dans ce travail, nous visons un système informatique composé de plusieurs unités de calcul connectées à des capteurs. Par exemple, imaginez un bâtiment intelligent avec un contrôle centralisé et des unités de traitement (une infrastructure Edge classique est composée de plusieurs de ces systèmes informatiques, mais nous nous concentrons sur un seul).

Dans ce type d'infrastructure, les ressources matérielles permettent de traiter les tâches produites par les capteurs. Lorsqu'aucune analyse n'est effectuée, les unités de calcul disponibles peuvent être utilisées pour des calculs externes ou délocalisés, tels que le calcul participatif (*volonteer computing*) [And04].

Le problème de la planification à deux agents a été introduit par Agnetis *et al.* dans [Agn+04]. Cet article fondateur était consacré à la planification de deux ensembles de tâches non préemptives en compétition pour être exécutées sur une seule machine, où chaque agent cherche à minimiser sa propre fonction objectif basée sur le temps d'achèvement de ses propres tâches (maximum, somme, dates d'échéance, etc.). Les auteurs ont considéré deux façons de résoudre le problème : premièrement, en minimisant une fonction objectif tout en maintenant la seconde sous un seuil et, deuxièmement, en trouvant l'ensemble de paires non dominées de valeurs objectifs sur le front de Pareto. Plus tard, Agnetis *et al.* [Agn+14] ont étendu ce travail à d'autres problèmes de planification multi-agents, y compris l'exécution sur des machines parallèles.

On peut trouver dans la littérature de nombreuses variantes du problème de planification à deux agents, par exemple en considérant une chaîne de tâches pour chaque agent [Agn+15], des tâches avec des dates d'échéance [Che+19 ; YWC20], ou avec un temps de configuration supplémentaire lorsqu'une tâche d'un agent est traitée directement après une tâche du deuxième agent [LCF16].

Dans un contexte similaire, Baker et Smith [BS03] ont étudié le problème de deux ou trois agents sur une seule machine, où chaque agent a sa propre fonction objectif

à minimiser. Cependant, ils ont considéré l'approche à objectif unique en minimisant une combinaison linéaire de l'objectif de chaque agent. Liu *et al.* [LGL19] ont également considéré le problème avec des tâches arrivant hors ligne où les tâches avaient une date d'arrivée, avec comme objectif de minimiser une combinaison linéaire du temps de completion maximal (*makespan*) de chaque agents.

Saule et Trystram [ST09] se sont concentrés sur le problème d'un nombre arbitraire d'agents planifiant des travaux sur des machines parallèles, où l'objectif d'un agent est soit la minimisation du makespan, soit la somme du temps d'achèvement de ses tâches. Ils ont proposé des bornes d'approximation et des algorithmes d'approximation avec des ratios de performance dépendant du nombre d'agents.

Nous nous concentrons ici sur la minimisation du *Flowtime total*, qui est une mesure de la performance dans la gestion des tâches d'un système de traitement de tâches. Formellement, pour une tâche j, on note C_j la date à laquelle la tâche j termine son exécution, et r_j la date à laquelle la tâche j arrive dans le système, le flowtime de la tâche j peut s'exprimer de la façon suivante :

$$F_j = C_j - r_j$$

Minimiser le flowtime total revient donc à minimiser la somme du flowtime de chaque tâche. On peut aussi l'interpréter comme étant la minimisation du temps moyen où une tâche est présente dans le système. Si l'on considère des machines parallèles (le temps d'exécution pour une tâche est le même peu importe la machine qui l'exécute), cela revient également à minimiser le temps d'attente moyen des tâches.

Dans un second temps, nous nous intéressons également à la propriété du *makespan*, qui est définie pour un ordonnancement comme la valeur maximale de C_j. Le makespan est intéressant car il peut être interprété comme le dual de l'utilisation : plus le makespan est petit, moins il y a d'inactivité sur les machines. Nous utilisons le makespan à la fois comme une échéance à ne pas dépasser dans le Chapitre 3 et comme un objectif à minimiser dans le Chapitre 4.

1.3.1 Problème Bi-Agent : Associer Tâches Locales et Globales

Nous nous concentrons sur un problème bi-objectif où nous cherchons à minimiser le flowtime total des tâches produites localement par des capteurs tout en essayant de terminer les tâches (globales) prévues à l'avance dans un délai raisonnable. Le problème est complexe car l'arrivée de tâches locales peut se produire de manière imprévisible.

Ce problème peut être considéré comme une extension du problème de minimisation du *Flowtime* total de tâches online sur machines parallèles, sous la contrainte supplémentaire de la planification offline des tâches d'un deuxième agent.

Les caractéristiques des deux types de tâches, chacun associé à un agent, sont les suivantes :

Le premier agent est en charge des tâches générées localement par les capteurs. Nous appelons ces tâches "locales" car elles sont produites localement par des capteurs et arrivent dynamiquement (de façon online). Elles doivent être traitées dès que possible. La date d'arrivée d'une tâche locale n'est pas connue à l'avance, mais son temps d'exécution est connu dès lorsqu'elle est libérée. L'objectif cible du premier agent est de minimiser le flowtime total de ces tâches. En raison de leur localité, de leur temps d'exécution et de leur consommation de mémoire généralement faible, on considère que la préemption (sans migration) de ces tâches est autorisée.

Le second type de tâches correspond aux tâches externes soumises par lot. Nous les appelons tâches "globales". Ces tâches (offline) sont disponibles au début de l'ordonnancement. Le temps d'exécution de ces tâches est connu, et l'objectif que nous modélisons est de terminer cet ensemble de tâches en un temps raisonnable, avant une date limite commune qui est fixée au préalable, typiquement avant l'arrivée du lot suivant. En raison de leur nature et d'une consommation de mémoire possiblement importante par rapport aux tâches locales, une tâche globale ne peut pas être préemptée. Cependant, nous autorisons le rejet d'une tâche globale pendant son exécution si cela est nécessaire, pour maintenir une bonne performance du système. Intuitivement, une tâche globale rejetée sera soumise à nouveau dans un lot ultérieur sur les mêmes ressources de calcul ou sur d'autres.

Pour résumer, les entrées du problème de planification correspondant sont les suivantes : m machines identiques, $n^{\mathcal{L}}$ tâches locales et $n^{\mathcal{G}}$ tâches globales, p_j le temps de traitement de la tâche j. Les tâches des deux agents sont séquentielles.

Le problème auquel on souhaite s'intéresser ici est uniquement de trouver une politique d'ordonnancement pour intercaler les deux ensembles de tâches de la

"meilleure" manière possible par rapport aux objectifs ciblés. Nous ne souhaitons pas que le problème d'ordonnancement des tâches locales seules (respectivement des tâches globales seules) soit un problème difficile. C'est aussi pour cette raison que nous nous permettons de modéliser le problème de cette façon pour les locales, et sans nous poser la question de la réalisabilité concernant la *date limite* globale ; nous considérerons même que nous avons un planning global préétabli dans lequel des tâches locales viendront s'intercaler.

Pour le problème de minimisation du flowtime total, il est bien connu que la politique SRPT (*Shortest Remaining Processing Time*) donne une planification optimale pour le problème avec r_j sur une seule machine (tout comme SPT (*Shortest Processing Time*) pour le cas sans r_j) [Smi56]. A chaque instant, l'algorithme SRPT choisit d'exécuter la tâche ayant le plus court temps de traitement restant à effectuer (voir Figure (1.1)) .

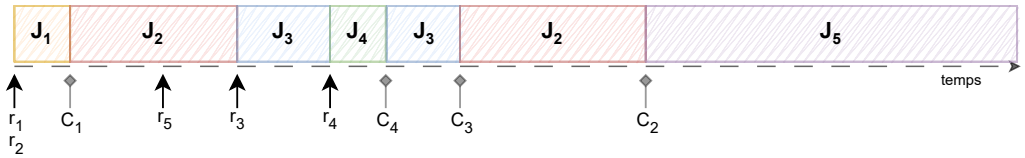


Figure 1.1. : Un exemple d'exécution de SRPT sur un exemple à 5 tâches (J_1, J_2, J_3, J_4, J_5). A chaque événement (fin d'exécution ou arrivée d'une nouvelle tâche), la décision d'exécuter la tâche avec le temps restant d'exécution minimum est prise.

Malheureusement, l'optimalité de SRPT ne s'applique pas dans notre contexte avec un deuxième agent, comme le montrera la Section (3.3). Lorsque l'on considère $m \geq 2$ machines, le problème devient NP-difficile [DLY90], et Leonardi et Raz [LR07] ont montré que SRPT est $O(\min\{\log(P), \log\frac{n}{m}\})$-compétitif, où P est le rapport entre le temps d'execution maximum et minimum des tâches, et n est le nombre de tâches. Ils ont également montré qu'aucun algorithme online ne pourrait atteindre un meilleur rapport de compétitivité.

De tels résultats forts sur l'approximation des problèmes d'ordonnancement ont motivé l'introduction du modèle d'augmentation de ressources, où un algorithme online reçoit plus de pouvoir lors de la comparaison à l'optimal.

Dans le contexte de l'ordonnancement sur une seule machine, Feng *et al.* [Fen+15] ont étudié le problème commun à deux agents d'Agnetis *et al.* avec la caractéristique supplémentaire que les tâches peuvent être rejetées avec une pénalité donnée à la valeur objectif de l'agent correspondant. Dans le cadre de l'approche online, Lucarelli *et al.* ont également introduit les modèles d'augmentation de vitesse et de rejet pour le problème de minimisation totale du temps d'écoulement (pondéré)

sur des machines non liées [Luc+16 ; Luc+18a ; Luc+18b] et liées [Luc+19]. Les auteurs y ont proposé plusieurs algorithmes dont la compétitivité a été prouvée en utilisant la technique du dual-fitting. Nous utiliserons la même approche dans l'analyse de notre algorithme dans la Section 3.4.

Contributions

L'étude de ce problème est détaillée dans le Chapitre 3.

Nous montrons que même en autorisant la rejet de certaines tâches et en autorisant une augmentation de la vitesse des ressources de calcul, à moins de rejeter plus de la moitié des tâches globales, il est impossible d'obtenir un *algorithme-compétitif* à rapport constant.

Nous montrons qu'à moins de rejeter plus de la moitié des tâches globales, le rapport-compétitif minimal qu'on pourrait espérer obtenir dépendra de $\mathcal{W}$, $\mathcal{W}$ étant le rapport entre la charge de travail totale des tâches globales et la charge de travail totale des tâches locales.

D'un point de vue positif, nous proposons un algorithme pour résoudre le problème de planification à deux agents sous le modèle d'augmentation des ressources, avec à la fois rejet et augmentation de la vitesse, pour lequel nous proposons une analyse basée sur la technique du dual-fitting, afin de pouvoir fournir un rapport compétitif. Celui ci dépend de la valeur de $\mathcal{W}$, ainsi que des paramètres utilisés pour l'augmentation de ressources.

1.3.2 Ordonnancement des tâches locales entre elles

En réalité, planifier l'exécution les tâches locales (entre elles) n'est pas aussi simple qu'il n'y paraît. Nous nous intéressons ici au sous-problème particulier où les tâches locales ont déjà été affectées à une machine et sont en attente d'être exécutées. Nous avons donc une file d'attente de tâches locales à exécuter sur une machine.

Dans la plupart des applications concrètes, les temps d'exécution ne sont pas précisément connus. Cependant les algorithmes les utilisent généralement, en particulier pour les fonctions objectifs basées sur les temps de complétion. Pour ce problème, quand les durées d'exécution sont connues, la politique SPT est optimale [Smi56]. Mais que faire lorsque ceux-ci ne sont pas connus ?

Cette question a été étudiée dans de nombreux travaux autorisant différents mécanismes, tels que l'usage de la préemption pour proposer différents algorithmes d'approximations tant sur ce problème classique à une machine (Round-Robin), que sur des variantes du problème (avec poids : [KC03], sur machines parallèles avec tâches *Malléables* [Bea+12],...) ou encore l'usage de prédictions avec erreurs ([LV18], [MNS07]), [PSK18], [Im+21], [LM22], [Bam+22]).

Nous nous intéressons ici à un problème d'ordonnancement de tâches indépendantes non-clairvoyantes, où les durées des tâches ne sont pas connues à l'avance, où la préemption n'est pas permise, et où toute tache commencée ne peut être arrêtée.

Nous supposons que nous avons accès à un oracle qui, lorsqu'on l'interroge, fournit la durée correcte d'une tâche spécifique. Cependant, cette requête prend un certain temps (noté p_T) et aucune autre tâche ne peut être traitée simultanément. L'objectif est de trouver un ordonnancement qui minimise le temps de complétion total des tâches attendu (min $\sum_j C_j$), en utilisant l'information fournie par l'oracle de manière efficace.

Cette approche "mixte" cherche à créer des algorithmes robustes aux incertitudes en s'autorisant un accès à une information supplémentaire sur les données, tout en étant parcimonieuse et en évitant d'utiliser l'analyse de toutes les données (comme en apprentissage automatique).

Notre étude se focalise sur un sous-problème particulier de l'ordonnancement local des tâches, où seulement deux types de tâches sont considérés pour simplifier le problème. Ces tâches sont des tâches courtes, constituées d'un nombre connu n_S de durée p_S, et des tâches longues, constituées d'un nombre également connu n_L de durée p_L.

Cette modélisation de l'oracle semble similaire à celle étudiée dans [Duf+22], mais diffère sur un point clé : les valeurs n_S et n_L sont considérées comme connues et fixes. Les idées développées dans [Duf+22] supposent un adversaire fournissant l'instance la plus défavorable. Dans notre cas, l'adversaire est quelque peu restreint dans ses décisions, car il ne peut pas choisir le nombre de tâches courtes et longues de l'instance, mais seulement leur ordre d'apparition (l'ordre dans lequel les tâches sont considérées), ce qui permet d'autres algorithmes plus fins, mais contredit certaines conjectures et règles de dominances précédemment établies. Cette hypothèse (n_S et n_L connus) complexifie le comportement de l'adversaire ainsi que l'analyse des algorithmes.

La problématique principale est de déterminer le compromis entre l'utilisation ou non de l'oracle, et dans le cas où il est utilisé, quand l'utiliser.

Contributions

Dans le Chapitre 2, nous nous intéressons à la conception d'algorithmes fournissant de bonnes performances en moyenne. Nous présentons deux variantes du problème, une *adaptative* : pour lesquelles on s'autorise à se reposer sur le contexte d'exécution pour décider d'utilisation de l'oracle, une *non-adaptative* : pour laquelle on ne l'autorise pas. Nous présentons des algorithmes optimaux dans les deux cas.
De plus, nous présentons une heuristique gloutonne, inspirée des deux algorithmes précédents, qui pour le cas adaptatif ne fournit pas un résultat optimal mais offre des performances extrêmement proches. Nous abordons brièvement la robustesse du système en cas de prédictions incorrectes ou d'incertitude sur le nombre de tâches.

Dans le Chapitre 5, nous travaillons sur le pire cas, pour fournir des algorithmes-compétitifs. La difficulté du problème résidant dans l'incertitude sur la durée des tâches, mais aussi leur ordre d'arrivée, nous nous intéressons à la définition offline de notre problème online. Nous y évoquerons également les critères de robustesse que nous ciblons dans la suite de nos travaux : le min max Regret.

Les résultats présentés dans ce chapitre sont des résultats préliminaires, les recherches sont toujours en cours. Pour l'heure, nous arrivons à caractériser certaines règles de dominance qui régissent le comportement de certains optimums pour pouvoir fournir une analyse approfondie de ceux ci.

1.3.3 Ordonnancement des tâches Globales entre elles, sur plateforme hybride CPU / GPU

Dans la Sous-Section 1.3.1, nous avons considéré que l'ordonnancement des tâches globales n'était pas un défi majeur et que nous pouvions partir d'un ordonnancement préétabli pour éviter cette difficulté. Une particularité de l'algorithme présentée dans le Chapitre 3, qui associe les tâches locales et les tâches globales est qu'il fonctionne tant que les tâches globales sont ordonnancées par un algorithme de Liste [Gra69]. Sans plus rentrer dans le détail, cet algorithme peut être modifié de façon à considérer des tâches globales ordonnancées différemment, à condition que leur ordonnancement respecte la date limite qui est fixée pour les tâches globales.

De cette façon, tant qu'on respecte cette date limite, il est tout à fait possible de leur créer un ordonnancement de façon à minimiser un certain critère pour les tâches globales (en gardant à l'esprit que celui ci va être légèrement perturbé par les tâches locales tel que défini dans le Chapitre 3). C'est pourquoi nous étudions à présent l'ordonnancement de ces tâches globales plus en détail, en nous focalisant sur le critère du *makespan* (donc de la date limite globale).

La mutualisation des plateformes implique souvent une hétérogénéité des ressources de calcul. Nous analysons un problème de planification d'*applications* de tâches sur un système de ressources hybride composé de m CPUs identiques et k GPUs identiques. Une application est décrite comme un ensemble de n tâches mono-processeur liées par des précédences décrites par un graphe orienté acyclique. Une arrête (i,j) signifie qu'il y a une relation de précédence entre la tâche i et la tâche j ; c'est à dire que commencer la tâche j nécessite que la tâche i ait terminé son exécution.

Nous n'imaginons aucune relation entre les temps d'exécution d'une tâche sur un CPU ($\overline{p_j}$) et celui sur un GPU ($\underline{p_j}$). Cela est justifié par les systèmes réels où les tâches effectuant, par exemple, des opérations de matrices peuvent être exécutées beaucoup plus efficacement sur un GPU, tandis que l'exécution des tâches qui doivent souvent communiquer avec le système de fichiers est plus rapide sur un processeur central. Par conséquent, dans l'intégralité du Chapitre 4 qui traite du sujet, sans perte de généralité, nous pouvons considérer que $m \geq k$.

Contributions

L'algorithme HLP présenté dans [KMT15] résout ce problème en garantissant un rapport d'approximation de 6, en priorisant l'affectation des tâches (sur CPU ou sur GPU) par rapport à leur planification (au sein des CPUs par exemple). Amaris *et al.* [Ama+17] montrent sur un exemple qu'en suivant cette affectation, ce rapport de 6 est atteint.

Nous proposons une modification de cet algorithme, basé sur une procédure d'arrondi différente d'un programme linéaire (qui est non optimale pour le problème seul de l'affectation), et sur une heuristique. Nous obtenons alors un rapport d'approximation de 5,83. Plus précisément, ce rapport vaut $3+2\sqrt{2}$ et tend vers 3 lorsque m/k est proche de 1.

La seconde contribution de ce chapitre est l'amélioration de la borne inférieure du rapport d'approximation (conditionnel) à 3 pour toute valeur de m/k en supposant une conjecture introduite par Bazzi et Norouzi-Fard [BN15].

Enfin, nous évoquons une perspective possible d'extension de ces travaux au problème $(R|\text{prec}|C_{max})$.

1.4 Synthèse

En résumé, cette thèse vise à proposer des algorithmes d'ordonnancement efficaces pour les systèmes distribués en utilisant des techniques de programmation linéaire et des mécanismes mathématiques récents telles que la prédiction ou l'augmentation de ressources (comme le rejet et l'augmentation de vitesse) pour contourner les aspects qui rendent les problèmes structurellement difficiles.

Nous nous concentrons sur trois cas d'application spécifiques et présentons différents algorithmes, y compris des algorithmes d'approximation et des algorithmes ayant un bon comportement en moyenne, avec une attention particulière à la complexité ; l'objectif étant de fournir des solutions raisonnées à faible coût pour résoudre ces problématiques.

Enfin, nous discutons quelques perspectives pour poursuivre ces travaux.

2 Un oracle pour un modèle non clairvoyant de tâches indépendantes

Ce chapitre présente plusieurs travaux en cours en collaboration avec Fanny Dufossé, Malin Rau et Denis Trystram.

Dans de nombreuses plateformes de calcul, la liste des tâches à effectuer est connue mais pas leur temps de traitement. L'objectif ici est d'étudier comment la connaissance de certains paramètres clés comme la durée de chaque tâche impacte l'obtention de meilleures performances : obtenir de telles informations à un coût qui n'est pas négligeable (coût d'entraînement d'un modèle prédictif, temps de transmission de l'information, sécurité du système), et la fiabilité de cette information peut avoir un impact non négligeable.

Nous nous intéressons ici à l'utilisation d'un oracle externe qui fournit à la demande la durée d'une tâche et examinons le problème de la planification avec oracle comme un problème d'optimisation en incluant le coût des requêtes dans le processus d'optimisation de la planification.

2.1 Introduction

Nous nous intéresserons ici à la modélisation introduite dans l'Introduction (1.3.1). L'ordonnancement pour minimiser la somme des temps de complétion est un problème classique dans le domaine de l'algorithmique [Pin12]. Il est bien connu que la politique qu consiste à ordonnancer la tâche la plus courte en premier (SPT) est optimale pour ce problème [CMM03]. Cependant, cet algorithme nécessite une connaissance complète de la durée des tâches.

Récemment, il y a eu un grand intérêt pour l'ordonnancement avec un oracle pour gérer les problèmes où des incertitudes surviennent. L'idée est de se fier à un processus externe (un oracle), quel qu'il soit, pour obtenir la valeur exacte de certains paramètres (ou seulement une approximation de la valeur exacte), comme

dans notre cas : les temps de traitement inconnus. C'est une façon d'améliorer la connaissance globale qui se présente naturellement dans de nombreuses applications, mais cela a un coût.

Cette idée d'oracles a été introduite dans [Kah91] pour résoudre des problèmes sous incertitudes. Le problème de la minimisation des requêtes d'oracles a été étudié dans des domaines tels que la logique (arbre booléen [Cha+02]) avec des requêtes sur l'affectation des variables, des problèmes d'ensembles finis (classement [Fed+00 ; KT01 ; Cha+21], sac à dos [Goe+15]) avec des requêtes sur des valeurs de l'ensemble, la géométrie 2D (détermination de l'enveloppe convexe [Bru+05]) avec des requêtes sur les positions, la théorie des graphes (arbre couvrant minimum [Erl+08 ; MMS17] et le plus court chemin [Fed+07]) avec des requêtes sur les poids des arêtes. Les différentes applications du modèle d'oracle sont résumées pour une enquête dans [EH15].

Dans le domaine de l'ordonnancement, on retrouve également les oracles dans la liste des outils théoriques utilisés. L'article fondateur (Kahan *et al.* [Kah91]) a introduit un modèle théorique pour étudier les problèmes de stockage avec des données évolutives. L'utilisation de l'oracle dans les problèmes d'ordonnancement par des algorithmes stochastiques a été étudiée par Levi *et al.* [LMS19]. Ici, les poids et les temps de traitement des tâches sont tirés de distributions connues et l'algorithme peut interroger ces paramètres. Dans ce modèle, tester une tâche ne raccourcit pas son temps d'exécution, cela fournit simplement de l'information pour l'ordonnanceur. Les résultats principaux sont de dériver des résultats structuraux sur les politiques optimales et des algorithmes efficaces basés sur la programmation dynamique où l'approche principale a été de chercher des algorithmes déterministes. Récemment, Dürr *et al.* [Dür+18] ont introduit le problème de l'ordonnancement sur une seule machine utilisant l'"incertitude explorable", qui est un autre nom pour l'oracle, à travers des arbres adverses de la théorie des jeux.

Ce travail a été amélioré par Dufossé *et al.* dans [Duf+22]. La valeur objectif de l'ordonnancement résultant est comparée à la valeur objectif optimale, qui est calculée à l'aide des informations complètes. Le but est de minimiser le rapport de compétitivité qui donne le coût des temps de traitement cachés. Ce travail a été étendu par Albers *et al.* dans [AE20] pour des temps de test non uniformes.

Avec l'augmentation de la complexité des systèmes numériques actuels, il existe de plus en plus d'incertitudes dans les durées de tâches qui doivent être exécutées. L'enjeu commun est donc désormais fournir des algorithmes robustes à ces incertitudes. Notre approche proposée ici est "mixte" ; nous cherchons à concevoir des algorithmes robustes aux incertitudes en ayant accès à des informations supplémentaires sur les

données, mais nous souhaitons que cette approche soit économe, et ne pas avoir recours à l'analyse de toutes les données (comme en apprentissage automatique).

2.1.1 Description informelle

Nous considérons le problème de l'ordonnancement d'un ensemble de n tâches indépendantes non clairvoyantes, c'est-à-dire où la durée des tâches n'est pas connue avant leur exécution.

Nous définissons un problème d'ordonnancement simplifié à partir des circonstances décrites ci-dessus comme suit. Nous disposons d'une seule machine et d'un ensemble $\mathcal{J}$ de n tâches. Chaque tâche $j \in \mathcal{J}$ a un des deux temps de traitement $p(j) \in p_S, p_L = p_S + \Delta$. Nous noterons n le nombre de tâches, le nombre de tâches courtes est noté n_S et celles-ci ont un temps de traitement noté p_S. De même, le nombre de tâches longues est noté $n_L = n - n_S$ et le temps de traitement d'une tâche longue est noté p_L.

Toutes ces valeurs (p_S, p_L, n_S, n_L et donc n) sont connues par l'ordonnanceur. Bien que nous connaissions le nombre total de tâches ainsi que le nombre de tâches courtes et longues, nous ne savons pas quelle tâche nécessite un temps de traitement court ou long. De plus, une fois lancées, les tâches ne peuvent pas être interrompues.

Notre objectif ici est de trouver un ordonnancement qui minimise le temps de complétion total des tâches. Dans ce Chapitre, nous nous intéresserons à des algorithmes permettant d'obtenir de bonnes performances en moyenne.

Nous avons accès à un Oracle qui, lorsqu'il est interrogé à propos d'une tâche j, nous indique si j appartient à l'ensemble des tâches courtes ou à celui des tâches longues. Interroger l'oracle prend un certain temps (p_T) et aucune autre tâche ne peut être traitée simultanément.

Dans ce chapitre, nous faisons plusieurs hypothèses pour aborder ce problème. Outre le fait de considérer que les nombres de tâches courtes et longues sont connus à l'avance, nous supposons tout d'abord que l'oracle a toujours raison.

De plus, nous choisissons de modéliser ce problème avec uniquement deux ensembles de tâches, motivés par l'argument de [Dür+18] indiquant que cela permet de maximiser le sur-coût induit par les erreurs faites dans notre ordonnancement (exécuter une tâche longue avant une tâche courte) par rapport à un ordonnancement optimal omniscient (SPT).

2.1.2 Contributions

La question principale abordée dans ce travail est "comment déterminer le compromis entre utiliser ou non un oracle ?" et, dans le cas où il est utile de l'utiliser, "quand l'utiliser ?". Nous présentons deux variantes du problème en fonction de la manière d'utiliser l'oracle. Pour les deux cas, les algorithmes sont basés sur une analyse approfondie.

— Premièrement, nous limitons l'analyse à un cadre non adaptatif où les requêtes à l'oracle sont déterminées à l'avance par l'algorithme. Le problème ici est de déterminer quelles tâches tester. Nous fournissons dans ce cas un algorithme optimal. Nous prouvons que la meilleure stratégie de test non adaptatif en ce qui concerne le temps total d'achèvement attendu est de tester les n_T premières tâches pour $n_T \in 0, \ldots, n-1$. Nous analysons ensuite son coût attendu.

— Deuxièmement, nous étudions le cas adaptatif où ces requêtes ne sont pas fixées à l'avance par l'algorithme et la question "tester ou non" dépend des tâches déjà rencontrées. Nous introduisons deux algorithmes. Le premier algorithme utilise la programmation dynamique et nous établissons qu'il est le meilleur algorithme possible de ce type. Sa complexité temporelle est en $\mathcal{O}(n^3)$. Le deuxième algorithme est une heuristique gloutonne inspirée des deux algorithmes précédents. Bien qu'il n'atteigne pas la solution optimale, des expériences menées montrent qu'il a une performance moyenne proche de la meilleure possible.

Les deux cas sont évalués par des simulations pour étudier l'influence des différents paramètres.

2.1.3 Organisation du chapitre

Nous décrivons formellement le problème dans la Section 2.2 et certaines propriétés préliminaires utiles sont discutées. La Section 2.3 présente une analyse complète du cas non-adaptatif, ce qui signifie que le nombre d'appels à l'oracle est fixé en tant qu'entrée du problème. Nous montrons comment déterminer de manière optimale ce nombre. Dans la Section 2.4.3, nous étendons notre étude à des stratégies adaptatives où les appels à l'oracle sont déterminés tâche par tâche. Nous présentons en particulier un algorithme qui a les meilleures performances possibles en moyenne. Enfin, des expériences basées sur des simulations sont menées et analysées dans la Section 2.5 pour mieux comprendre le problème et pour obtenir des informations sur le rôle de ses différents paramètres.

2.2 Modélisation du problème

Avant d'entamer la définition formelle du problème, nous allons nous baser sur sa description informelle pour énoncer des règles de dominance simples. Cette approche nous permettra de définir les paramètres clés du problème et de proposer une modélisation épurée qui ne comporte pas de paramètres superflus.

Il peut être déduit de la description du problème que tout algorithme conçu pour le résoudre doit uniquement choisir entre ces trois actions pour chaque tâche :

1. Tester la tâche courante j,
2. Exécuter j sans la tester,
3. Sélectionner une tâche déjà testée j et l'exécuter.

Propriété 1 : Une tâche courte détectée doit être planifiée immédiatement et une tâche longue détectée doit être planifiée "à la fin" de la planification.

Il est connu, d'après [Pin12], que planifier les tâches dans l'ordre croissant de leurs temps de traitement fournit la solution optimale lorsqu'on vise à minimiser le total de leur temps de réalisation. Puisque nous ne savons pas quelle tâche a quel temps de traitement, cette heuristique gloutonne ne peut pas être utilisée directement. Cependant, nous savons qu'une tâche testée j devrait être planifiée immédiatement après le test si elle est courte, et qu'elle devrait être reportée si elle est longue. Ainsi, pour tous les algorithmes considérés, nous supposons que les tâches testées sont traitées immédiatement après leurs tests, tandis que les tâches longues sont placées dans une file d'attente et planifiées une fois qu'il ne reste plus aucune tâche non testée.

Propriété 2 : Il est inutile de tester la dernière tâche inconnue.

Une autre observation est la suivante. Au cours d'une planification, lorsqu'il ne reste qu'une seule tâche qui n'a ni été exécutée ni testée, cela signifie que la durée des autres $n - 1$ tâches est connue. Ainsi, la durée de cette dernière tâche peut être déduite des autres tâches. Quoi qu'il arrive précédemment, la dernière tâche est exécutée sans être testée dans une planification optimale.

Propriété 3 : Tester est une mauvaise idée si le coût d'un test est trop élevé

Enfin, nous notons qu'il est raisonnable de ne tester que si $p_L > p_S + p_T$, c'est-à-dire si $\Delta > p_T$. Cela est évident car si le temps total nécessaire pour tester et traiter une tâche courte est plus long ou aussi long que le temps total nécessaire pour traiter une tâche longue, il est préférable de simplement traiter la tâche longue avant la tâche courte plutôt que de tester une tâche.

Propriété 4 : Aucun algorithme ne pourra avoir de meilleures performances que SPT.

Lemme 2.2.1. *Pour chaque instance composée de n tâches (n_S tâches courtes et $n_L = n - n_S$ tâches longues), et pour une fonction objectif $\sum_j C_j$, l'ordonnancement omniscient optimal a un temps total de completion :*

$$SPT(I) = \frac{n(n+1)}{2} p_S + \frac{n_L(n_L+1)}{2} \Delta$$

Preuve. Il est connu que l'algorithme SPT (Shortest Processing Time First) fournit l'ordonnancement avec le temps total d'achèvement optimal [Pin12]. Dans cet ordonnancement, les n_S tâches courtes sont traitées en premier, puis les n_L longues. Par conséquent, le temps total d'achèvement peut être écrit comme suit :

$$\frac{n_S(n_S+1)}{2} p_S + n_L \cdot n_S \cdot p_S + \frac{n_L(n_L+1)}{2} p_L = \frac{n(n+1)}{2} p_S + \frac{n_L(n_L+1)}{2} \Delta \quad (2.1)$$

□

2.2.1 Définition formelle et notations

Au vu de ces résultats préliminaires, nous pouvons donc nous contenter de cette formulation formelle du problème.
Entrée : une Instance $I = (n_L, n_S, p_L, p_S, p_T)$ dont les 5 paramètres sont des entiers connus, supérieurs ou égaux à 0.

- n_L correspond au nombre de tâches longues de l'instance I,
- n_S au nombre de tâches courtes,
- p_L correspond à la durée d'exécution d'une tâche longue,
- p_S à la durée d'exécution d'une tâche courte,
- et p_T correspond "au temps d'exécution" de l'oracle.

On notera $\mathcal{T}$ l'ensemble des tâches de l'instance I (tel que $|\mathcal{T}| = n_S + n_L$). L'objectif est de minimiser $\sum_{\forall j \in \mathcal{T}} C_j$ où C_j est la date de complétion de j, c'est à dire la date où l'exécution de la tâche j termine.

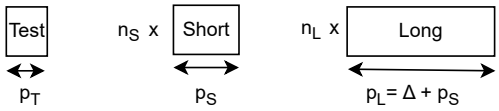


Figure 2.1. : Schéma récapitulatif présentant les choix de notations qui définissent une Instance. (Notons que la permutation entre les tâches longues et courtes ne fait pas partie de notre définition d'une instance).

Pour une Instance $I = (n_L, n_S, p_L, p_S, p_T)$, on considérera qu'il existe $\binom{n_S+n_L}{n_L}$ permutations différentes entre tâches longues et courtes. On pourra considérer une permutation comme étant la suite des tirages aléatoires qui définissent quelle est la prochaine tâche à considérer par l'ordonnanceur.

Nous noterons $\mathrm{SPT}(I)$ le temps de complétion total donné par un algorithme optimal omniscient (qui applique la politique Shortest Processing Time : c'est à dire qui exécute les tâches dans l'ordre croissant de leur durée) pour une Instance $I = (n_L, n_S, p_L, p_S, p_T)$.

Pour tout algorithme A et instance I, nous noterons $E(A(I))$ la moyenne des coûts attendus sur toutes les permutations possibles de l'instance I :

$$E(A(I)) = \frac{1}{\binom{n_S+n_L}{n_L}} \sum_{\forall \text{ permutation } P \text{ de l'instance } I} (\text{l'algorithme } A \text{ joué sur } P)$$

Définition du surcoût SC

Voici une autre notation qui va être utilisée dans les différentes preuves de ce chapitre. Pour un algorithme A, nous utilisons souvent, pour simplifier les calculs et le raisonnement, la différence (qu'on notera SC) entre les sommes des temps de complétion données respectivement par A et par la politique SPT, de manière à ce que

$$E(A(I)) = \mathrm{SPT}(I) + SC(I)$$

Nous définissons SC_j comme la contribution (ou l'impact) de chaque tâche j ($\in [1; n]$) dans ce SC. Intuitivement, ce surcoût qu'on associe à une tâche est composée du retard induit par un test ainsi que par l'impact d'exécuter une tâche longue avant des tâches courtes.

Figure 2.2. : Exemple d'application de la définition du surcoût. Nous considérons 3 ordonnancements différents d'une instance composée de 2 tâches longues et 3 tâches courtes. Notons que l'ordonnancement 1) correspond à un ordonnancement donnée par la politique SPT.

Le deuxième ordonnancement définit par la Figure (2.2) exécute tout d'abord une tâche courte (ce qui correspond à la politique SPT), puis décide d'exécuter une tâche longue. Enfin, les 3 tâches restantes sont exécutées en respectant la politique SPT, c'est à dire les deux courtes d'abord et une longue ensuite. La seule décision prise ici contraire à la règle SPT a été d'exécuter une tâche longue avant deux tâches courtes ; ce faisant, si on se compare avec l'ordonnancement 1) donné par SPT, la date de fin de la tâche longue a été réduite de $2p_S$, alors que la date de complétion de chacune des deux tâches courtes a été augmentée de p_L. Cette décision a donc causé au final une dégradation de l'objectif (somme des temps de complétion) de 2Δ. Le choix est fait d'imputer cette dégradation à la tâche longue mal positionnée : la valeur de SC_j (avec j correspondant à cette tâche mal positionnée) est égale à 2Δ, alors que toutes les autres tâches j (bien positionnées) auront une valeur $SC_j = 0$.

Le troisième ordonnancement définit par la Figure (2.2) met en évidence le surcoût induit par un test. Ici, il a été décidé de tester la tâche longue (qui était mal positionnée précédemment). Cette décision a retardée de la valeur p_T la date de fin de complétion de chacune des tâches, sauf celles dont l'exécution était terminée au moment du test : donc les tâches inconnues mais également , si il y en avait eu, les tâches longues détectées, repoussées à la fin de l'ordonnancement. Cette différence entre l'objectif obtenu par l'ordonnancement 3) et SPT est de $4p_T$. Cette dégradation a été causée par le test de la tâche longue, cette dégradation est donc imputée à cette tâche longue, pour laquelle SC_j vaut $4p_T$.

Avec notre définition, SC_j est composé de deux éléments : l'impact des tests (0 si j n'est pas testé, sinon tout le retard ajouté au planning à cause du test) ; et le coût d'une erreur (0 si une tâche courte est planifiée immédiatement, $p_L - p_S$ pour toutes les tâches courtes exécutées après j). Avec cette formulation, nous avons $\sum_{j\in\mathcal{T}} SC_j + \mathrm{SPT}(I) = \sum_{j\in\mathcal{T}} \mathsf{C}_j$.

Plus formellement, pour l'instant, on se contentera de dire que :

Pour chaque tâche testée j :

$$\begin{aligned} SC_j =& p_T \cdot (|\text{tâches exécutées après } j \text{ (inclus)}|) \\ =& p_T \cdot (n - |\text{tâches déjà exécutées}|) \end{aligned} \tag{2.2}$$

Et pour chaque tâche non testée j (planifiée immédiatement) :

$$\text{Si } j \text{ est courte : } SC_j = 0 \tag{2.3}$$

$$\text{Si } j \text{ est longue : } SC_j = \Delta \cdot |\text{tâches courtes après } j| \tag{2.4}$$

("après" à prendre au sens strict, j est non incluse car j est longue)

2.3 Approche Non Adaptative

Dans la littérature, les algorithmes de décision non contextuels, appelés algorithmes non adaptatifs, sont bien documentés. Cette section présentera un algorithme naïf de base, qui exécute toutes les tâches sans en tester aucune. Nous présentons également un algorithme ($\mathcal{A}_{NA}$) pour lequel nous démontrons son optimalité, dans un cadre non adaptatif. Nous présentons également ses performances attendues (moyenne, meilleur et pire des cas).

2.3.1 Politique d'ordonnancement naïve : ne rien tester

Nous examinons le coût de cette politique dans le pire des cas ainsi qu'en moyenne. Bien que cette politique soit naïve, elle est intéressante car elle choisit de ne jamais tester les tâches. En d'autres termes, elle équivaut à planifier les tâches dans un ordre aléatoire, ce qui est la seule option possible pour notre modélisation si nous n'avions pas accès à l'Oracle. Les valeurs obtenues peuvent servir de référence pour évaluer la pertinence de notre modèle (c'est-à-dire la mise en place d'un système d'oracle et la possibilité de choisir de tester). Nous retrouverons les résultats obtenus par cette politique aléatoire dans la partie expérimentale qui remet en cause certaines hypothèses du modèle (2.6).

Théorème 2.3.1. *Pour toute permutation de l'instance $I = (n_L, n_S, p_L, p_S, p_T)$, on a*

$$A_{\text{aucunTest}}(I) \leq \text{SPT}(I) + n_L \cdot n_S \cdot \Delta$$

Preuve. Dans le pire des cas, toutes les tâches longues sont sélectionnées avant les tâches courtes. Donc la valeur de $\sum_{\forall j \in \mathcal{T}} C_j$ vaut est d'au plus :

$$\frac{n_L(n_L+1)}{2} p_L + \frac{n_S(n_S+1)}{2} p_S + n_S \cdot n_L \cdot p_L = \text{SPT} + n_L \cdot n_S \cdot \Delta$$

□

Ensuite, on considère le coût donné par l'algorithme en moyenne si on ne teste aucune tâche.

Théorème 2.3.2. *Le coût moyen $E(A_{aucunTest})$ pour une instance I, définie telle que $I = (n_L, n_S, p_T, p_L, p_S)$, généré par l'algorithme $A_{aucunTest}$ est donné par*

$$E(A_{aucunTest}(I)) = \frac{n(n+1)}{2}p_S + \frac{n_L(n+1)}{2}\Delta = \mathrm{SPT}(I) + \frac{n_L n_S}{2}\Delta$$

Preuve. *On pourra noter que cette preuve est redondante avec le Theorème (2.3.6), qu'il suffit d'appliquer avec le paramètre k=0 pour obtenir les mêmes résultats.*

La durée entre les dates de fin de deux tâches exécutées consécutivement d_j est égale au temps d'exécution de la tâche correspondante.
Ainsi $E(d_j) = \left(p_S \cdot \frac{n_S}{n} + p_L \cdot \frac{n_L}{n}\right)$. Il y a $(n+1-j)$ tâches pour lesquelles la date de fin d'exécution sera retardée par d_j. Donc on obtient :

$$\begin{aligned} E(A_{aucunTest}(I)) &= \left(\sum_{j=1}^{n} E(d_j) \cdot (n+1-j)\right) \\ &= \sum_{j=1}^{n} \left((p_S \cdot \frac{n_S}{n} + p_L \cdot \frac{n_L}{n}) \cdot (n+1-j)\right) \\ &= \frac{(p_S n_S + p_L n_L)}{n} \frac{n(n+1)}{2} \\ &= \frac{n(n+1)}{2} p_S + \frac{n_L(n+1)}{2}(p_L - p_S) \end{aligned}$$

□

Par conséquent, nous nous attendons à ce que le coût moyen supplémentaire (à SPT) soit réduit de moitié par rapport au pire des cas pour cet algorithme.

2.3.2 En quête de l'algorithme optimal : $\mathcal{A}_{NA}$

> *Dans ce monde merveilleux "Non Adaptatif", votre talent vous a plongé au plus profond de ce problème complexe. Vous devrez résoudre les énigmes algorithmiques pour découvrir les mystérieuses règles de dominance, menant au secret de l'algorithme optimal.*
>
> **— Une thèse dont vous êtes le personnage principal**

On prouve tout d'abord qu'un algorithme optimal applique la politique suivante : tester les taches jusqu'à un certain indice (qu'on notera n_T) puis exécuter le reste sans tester. Ensuite nous donnons la meilleure valeur possible pour n_T qui dépend des paramètres de l'instance I. Enfin, les performances de l'algorithme sont analysées en termes d'espérance, de meilleur et de pire cas.

Lemme 2.3.3. *Si l'algorithme optimal décide de tester des tâches, alors ces tâches seront les premières de la permutation.*

Esquisse de la preuve Nous démontrons par un argument d'échange que : en supposant qu'il soit pertinent de tester une tâche J_i après avoir exécuté une tâche J_{i-1} sans la tester ; alors il est préférable de tester d'abord la tâche J_i, puis d'exécuter ensuite sans la tester la tâche J_{i-1}.
On pourra ensuite déduire directement de cette règle de dominance que dans une solution optimale, si une tâche J_k est testée, alors toutes les tâches J_i (avec $i \leq k$) le sont aussi.

Les détails des calculs de ce lemme sont dans l'Annexe A.

Preuve. Supposons l'existence d'un ordonnancement de n tâches, pour lequel la $(i-1)^{\text{ème}}$ tâche n'est pas testée. On rejoue cet ordonnancement post-mortem, et on se positionne au moment où on s'intéresse à la tâche J_{i-1}. A cet instant précis, on note k le nombre de tâches ayant déjà été exécutées, le nombre de tâches longues ayant été détectées vaut donc $i-2-k$.

Pour simplifier les démonstrations à venir, nous utilisons la notation suivante pour définir la valeur de la fonction objectif résultante de l'ordonnancement tel que décrit dans la Figure "x" :

$$\sum C_i$$

Figure 2.x

Tout d'abord, nous étudions l'impact sur l'ordonnancement d'avoir pris la décision de tester la tâche J_i (ordonnancement représenté avec nos notations Figure 2.4), ou d'avoir pris la décision de ne pas tester J_i (représenté Figure 2.3).

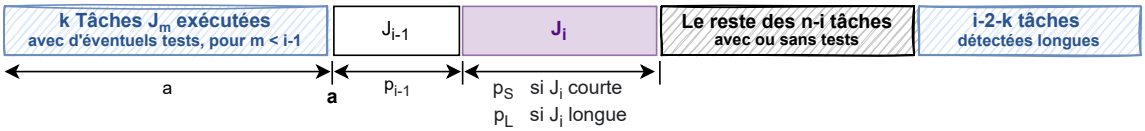


Figure 2.3. : Un ordonnancement si on décide de ne tester ni la tâche J_{i-1}, ni la tâche J_i.

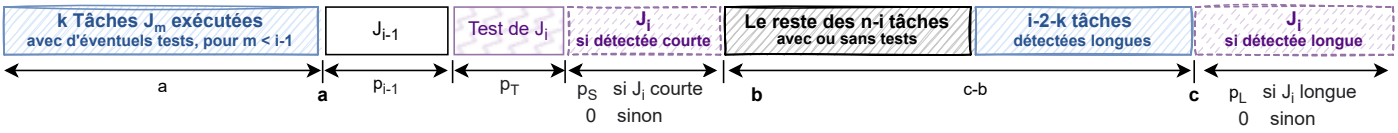


Figure 2.4. : Le même ordonnancement mais on décide d'executer la tâche J_{i-1} sans la tester puis de tester la tâche J_i.

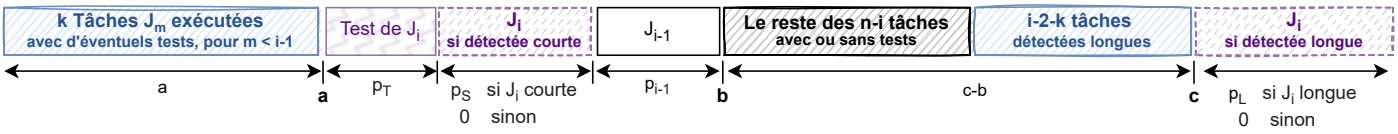


Figure 2.5. : Le même ordonnancement mais cette fois, on décide d'inverser J_{i-1} et J_i, c'est à dire tout d'abord de tester la tâche J_i, puis ensuite d'exécuter sans la tester la tâche J_{i-1}.

On souhaite tout d'abord se placer dans le contexte où : il est plus rentable de tester J_i que de ne pas le faire, donc où :

$$\underbrace{\sum C_i}_{\text{Figure 2.4}} \leq \underbrace{\sum C_i}_{\text{Figure 2.3}}$$

On obtient alors l'inéquation suivante :

$$p_T(n-k-1+d\frac{n_L}{n}) \leq \Delta\frac{n_S n_L}{n(n-1)}(n-i) \tag{2.5}$$

avec d le nombre de tests qui seront faits sur les tâches encore inconnues.

On s'intéresse à présent à la possibilité d'échanger J_{i-1} et J_i.

$$\begin{aligned}\sum_{\text{Figure 2.5}} \mathsf{C}_j - \sum_{\text{Figure 2.4}} \mathsf{C}_j &= p_T - \frac{n_S}{n}\frac{n_L}{n-1}\Delta \\ &\leq \Delta\frac{n_S n_L}{n(n-1)}\left(\frac{n-i}{(n-k-1+d\frac{n_L}{n})} - 1\right) \\ &\leq \Delta\frac{n_S n_L}{n(n-1)}\frac{(-i+k+1-d\frac{n_L}{n})}{(n-k-1+d\frac{n_L}{n})} \leq 0\end{aligned}$$

La dernière inégalité est correcte car $k \leq i-2$ et $n \geq k+2$. Donc si on veut tester J_i, il est préférable de l'échanger avec sa tâche précédente (J_{i-1}) si celle ci n'est pas testée. On peut en déduire récursivement que si on décide de tester m tâches (et d'exécuter sans tester les $n-m$ restantes), le meilleur coût moyen est obtenu en testant les m premières. □

Lemme 2.3.4. *Pour toute instance $I = (n_L, n_S, p_T, p_L, p_S)$ et pour un algorithme qui teste uniquement les n_T premières tâches, la meilleure valeur de n_T est*

$$\left\lfloor \frac{n \cdot n_S n_L \Delta - (n-1)(n^2+n_S)p_T)}{n_S(n_L\Delta - (n-1)p_T)} \right\rfloor$$

si $n_L n_S/n^2 \geq p_T/\Delta$, et 0 sinon.

Preuve. La définition de SC_i (donnée dans la Section (2.2)) nous donne :

Pour toute tâche i testée :

$$\begin{aligned}SC_i =& p_T \cdot (n \text{ - tâches déjà exécutées}) \\ =& p_T \cdot (n \text{ - nombre de tâches courtes détectées avec } (i-1) \text{ tests}) \\ \text{d'où } \mathbb{E}(SC_i) =& p_T \cdot (\text{n - } \mathbb{E}(\text{nombre de tâches courtes détectées avec } (i-1) \text{ tests})) \\ =& p_T \cdot \left(n - (i-1)\frac{n_s}{n}\right) \qquad (2.6)\end{aligned}$$

Et pour toute tâche i non testée (donc exécutée immédiatement), si i est longue :

$$\begin{aligned}\mathbb{E}(SC_i) =& \Delta \cdot (\text{nombre de tâches après i}) \cdot \mathbb{P}(\text{une tâche soit courte } | i \in \mathbb{L}) \\ \mathbb{E}(SC_i) =& \Delta \cdot (n-i)\frac{n_s}{n-1} \qquad (2.7)\end{aligned}$$

On peut noter que les deux composantes possibles de $\mathbb{E}(SC_i)$ pour une tâche J_i sont linéaires par rapport à i. Pour déterminer la valeur optimale du nombre de tâches à tester (n_t), nous devons trouver à partir de quelle valeur le surcoût de tester (donné par l'équation (2.6)) devient plus élevé que le coût induit par la probabilité d'exécuter une tâche longue avant une tâche courte (donnée par les formules (2.3) et (2.7)).

On cherche la plage de i ($\in [1, n]$) pour laquelle tester les tâches J_i est plus rentable que ne pas les tester.

$$p_T \cdot \left(n - (i-1)\frac{n_s}{n}\right) \leq \frac{n_l}{n} \cdot \left(\Delta(n-i)\frac{n_s}{n-1}\right) + \frac{n_S}{n} \cdot (0)$$
$$p_T(n-1)(n^2+n_s) - \Delta(n_l n_s n) \leq i\,(p_T(n-1)n_s - n_s n_l \Delta) \tag{2.8}$$

Si $p_T(n-1)n_s - n_s n_l \Delta > 0$; (2.8) nous donne :

$$i \geq \frac{p_T(n-1)(n^2+n_s) - \Delta(n_l n_s n)}{(p_T(n-1)n_s - n_s n_l \Delta)} = n + \frac{p_T(n-1)(n^2+n_s-n_s n)}{p_T(n-1)n_s - n_s n_l \Delta} > n$$

On en déduit que dans ce cas que les tâches qu'il est pertinent de tester sont celles dont l'indice est supérieur à n, comme $\{i | i > n, i \in [1, n]\} = \emptyset$, la meilleure décision est de ne tester aucune tâche. $n_T = 0$.

Si $p_T(n-1)n_s - n_s n_l \Delta \leq 0$; nous devons tester toutes les tâches J_i telles que

$$i \leq \frac{p_T(n-1)(n^2+n_s) - \Delta(n_l n_s n)}{p_T(n-1)n_s - n_s n_l \Delta}$$

Dans ce cas, si $p_T(n-1)n_s \geq n_s n_l \Delta$, on en déduit la valeur :

$$n_T = \left\lfloor \frac{p_T(n-1)(n^2+n_s) - \Delta(n_l n_s n)}{p_T(n-1)n_s - n_s n_l \Delta} \right\rfloor, \text{ et 0 sinon.}$$

Notons qu'on ne veut tester que si $n_T \geq 1$. Il est simple de vérifier que $n_T \geq 1$ si et seulement si $n_S n_L \Delta \geq n^2 p_T$.

En conséquence de cela, il y a deux conditions pour lesquelles le nombre optimal de tests est 0 : ($\Delta n_L - (n-1)p_T \leq 0$) et ($n_S n_L \Delta < n^2 p_T$). Cependant, $n_S n_L \Delta \geq n^2 p_T$ implique que $\Delta n_L - (n-1)p_T > 0$ car $n^2 > (n-1)n_S$. Donc $n_S n_L \Delta < n^2 p_T$ est une condition suffisante pour ne tester aucune tâche. □

Algorithme 1 : L'algorithme $A_{\mathrm{NA}}(n_L, n_S, p_T, p_L, p_S)$.

1. Q : la queue des tâches longues à exécuter à la fin de l'ordonnancement
2. **si** $n_L n_S/n^2 < p_T/\Delta$: $n_T = 0$
3. **sinon** $n_T = \left\lfloor \frac{n \cdot n_S n_L \Delta - (n-1)(n^2+n_S)p_T)}{n_S(n_L\Delta-(n-1)p_T)} \right\rfloor$
4. **pour**$\{i = 1, i \leq n_T, i++\}$
   a) Tester la prochaine tâche
   b) **si** le test dit "Court" : exécuter la tâche immédiatement
   c) **sinon** mettre la tâche dans Q
5. **pour**$\{i = n_T + 1, i \leq n, i++\}$
   a) Exécuter immédiatement la prochaine tâche sans la tester.
6. Exécuter toutes les tâches se trouvant dans Q

Théorème 2.3.5. *L'algorithme A_{NA} est optimal.*

Preuve. Ce résultat peut être directement obtenu grâce aux différentes règles de dominance décrites précédemment dans 2.2 ainsi que dans les deux précédents lemmes (2.3.4) et (2.3.4). □

Théorème 2.3.6. *Pour toute instance $I = (n_L, n_S, p_T, p_L, p_S)$ et un nombre donné de tests k ($\in [0, n]$), la valeur moyenne du temps total de complétion générée par un algorithme qui teste les k premières tâches de l'ordonnancement est donnée par la formule :*

$$\mathbb{E}(C(k)) = \mathrm{SPT}(I) + k(n - \frac{n_s}{2n}(k-1)) \cdot p_T + \frac{n_l \cdot n_s \cdot (n-k)}{n(n-1)} \frac{n-k-1}{2} \Delta$$

Preuve. Depuis (2.6), (2.3) et (2.7), on obtient directement :

$$\begin{aligned} E(C(k)) - \mathrm{SPT}(I) &= \sum_{i \in \mathcal{T}} \mathbb{E}(SC_i) \\ &= \sum_{i=1}^{k} (n - \frac{(i-1)n_s}{n}) p_T + \sum_{i=k+1}^{n} \left(\frac{n_l}{n} \Delta \frac{n_s}{n-1} (n-i) \right) \\ &= \cdot\, k(n - \frac{n_s}{n} \frac{k-1}{2}) p_T + \frac{n_l \cdot n_s \cdot (n-k)}{n(n-1)} \frac{n-k-1}{2} \Delta \end{aligned}$$

□

Meilleur cas pour l'algorithme A_{NA}

On cherche la permutation pour laquelle l'algorithme A_{NA} donne la solution avec la plus petite valeur de la fonction objectif, pour une instance I donnée. Avec des arguments d'échange dans la lignée de ceux présentés dans le Lemme (2.3.2), on peut montrer que le meilleur cas pour cet algorithme est celui où toutes les tâches courtes sont considérées avant les tâches longues. En effet, pour minimiser la valeur de la fonction objectif, il s'agit de minimiser la valeur de SC_i tant pour les tâches testées que pour les tâches non testées. De cette façon $SC_i = 0$ pour les tâches non testées, et cet ordonnancement permet de minimiser le nombre de tâches retardées par les tests (en maximisant au plus tôt le nombre de tâches (courtes) exécutées.

La valeur de la fonction objectif A_{NA} sur cette permutation est

$$\begin{aligned} \sum C_j &= \mathrm{SPT} + p_T \cdot n_T \left(n - \frac{n_T}{2} + \frac{1}{2} \right) && \text{si } (n_S \geq n_T) \\ &= \mathrm{SPT} + p_T \frac{n_S(n_S+1)}{2} + n_L \cdot n_T \cdot p_T && \text{si } (n_S < n_T) \end{aligned}$$

Pire cas pour l'algorithme A_{NA}

On cherche le pire cas pour l'algorithme, donc la pire permutation. Il est intéressant de montrer ici qu'il y a un équilibre à trouver ici. Avec un argument d'échange, il est facile de montrer que si l'adversaire décide de positionner une tâche longue parmi les n_T tâches testées, alors il devrait le faire au début (pour maximiser les coûts induits par les tests). Et pour toutes les tâches non testées, la meilleure façon de maximiser le surcoût est de positionner les tâches longues le plus tôt possible.

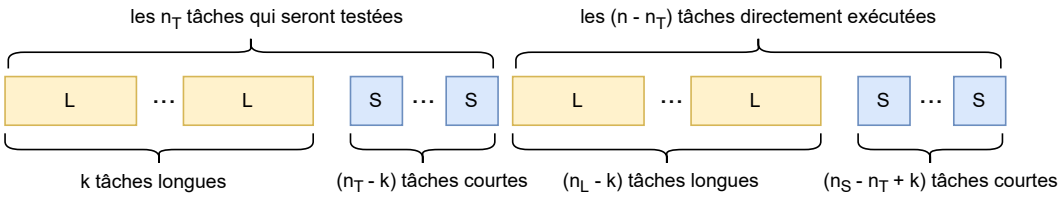


Figure 2.6. : Le pire Adversaire pour A_{NA}.

A partir de ces arguments, on peut déduire que la pire permutation est de la forme présentée dans la Figure (2.6), et que le seul "choix" pour l'adversaire est de décider combien de tâches longues doivent se trouver parmi les tâches testées dans la permutation.

Si on appelle F(k) la valeur de $\sum C_i$ pour cette permutation avec k le nombre de tâches longues présentes pendant la phase de test :

$$\begin{aligned} F(k) = & -k^2(\Delta + \frac{p_T}{2}) + k\left(p_T(n_T - \frac{1}{2}) + \Delta(n_L - n_S + n_T)\right) \\ & + \left(p_T n_T(n - \frac{n_T - 1}{2}) + \Delta(\frac{n_L^2}{2} + \frac{n_L}{2} - n_T \cdot n_L + n_S \cdot n_L) + p_S(\frac{n^2 + n}{2})\right) \end{aligned}$$

Pour trouver la valeur de k qui maximise $F(k)$, il suffit d'analyser

$$F(k) - F(k-1) = -2k(\Delta + \frac{p_T}{2}) + (p_T n_T + \Delta(n_L - n_S + n_T + 1))$$

et la valeur de k du pire cas est

$$\left\lfloor \frac{p_T n_T + \Delta(n_L - n_S + n_T + 1)}{p_T + 2\Delta} \right\rfloor$$, ou 0 si négatif

2.4 Approche adaptative

Après avoir étudié le cas non adaptatif, nous étudions à présent le cas où il est considéré comme possible de s'appuyer sur le contexte d'exécution pour prendre des décisions. Nous présentons deux algorithmes. Le premier algorithme ($OptimumPrime$) optimal est basé sur une programmation dynamique avec une complexité tant en temps qu'en mémoire en (o(n^3)). Le second algorithme est un algorithme glouton, inspiré des deux algorithmes optimums précédents (A_{NA} et $OptimumPrime$), qui, bien qu'il ne soit pas optimal, offre expérimentalement de très bonnes performances.

2.4.1 Notations spécifiques aux cas adaptatifs

Nous définissons un état s comme un triplet $(u_L(i), u_S(i), q(i))$ où

— l'instant i correspond à l'instant où on se demande si on doit ou non tester la $i^{\text{ème}}$ tâche de l'ordonnancement.

— $u_L(i)$ est le nombre de tâches inconnues longues restantes à exécuter au moment i : le nombre de tâches longues déjà exécutées à cet instant i vaut $n_L - u_L(i) - q(i)$.

— $u_S(i)$ est le nombre de tâches inconnues courtes restantes à exécuter au moment i : le nombre de tâches courtes déjà exécutées à l'instant i vaut $n_S - u_S(i)$.

— $q(i)$ est le nombre de tâches (longues) présentes dans la queue au moment i.

2.4.2 Algorithme optimal par Programmation Dynamique

Nous réutilisons les formules du surcoûts présentées dans la Sous-Section (2.2.1). Minimiser la somme des temps de complétion de toutes les tâches revient à minimiser la somme de tous les surcoûts (SC) induits par toutes ces tâches.

On note $SC(u_L(i), u_S(i), q(i))$ la valeur de la somme des surcoûts d'ordonnancer les $(u_L(i) + u_S(i))$ tâches restantes inconnues, en sachant qu'il y a $q(i)$ tâches longues dans la queue. On peut exprimer la valeur de $SC(u_L(i), u_S(i), q(i))$ en fonction des sous-problèmes $SC(u_L(i), u_S(i) - 1, q(i))$, $SC(u_L(i) - 1, u_S(i), q(i) + 1)$ et $SC(u_L(i) - 1, u_S(i), q(i))$ en considérant le passage d'un état à l'autre tel que défini dans la figure 2.7.

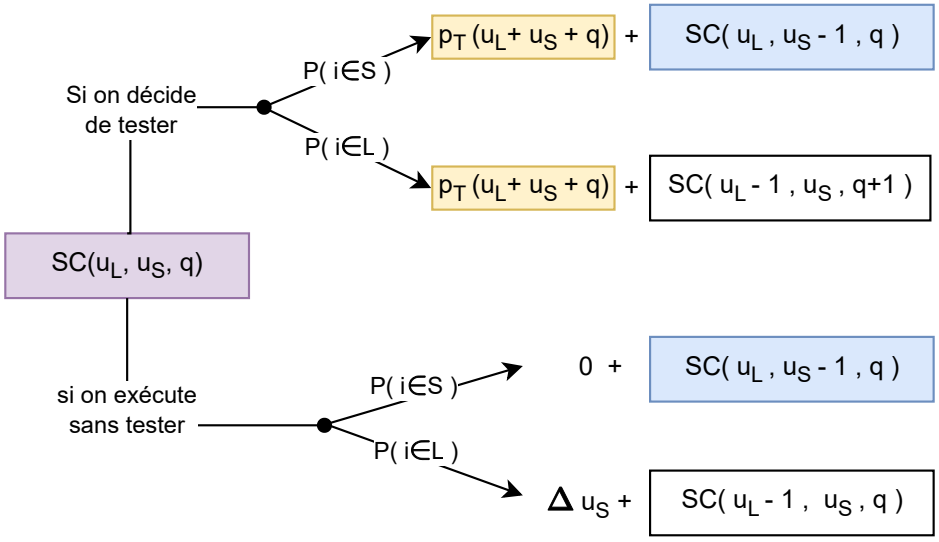


Figure 2.7. : Décomposition du problème adaptatif en sous-problèmes.

On peut écrire la formule de récurrence suivante qui donne la solution optimale pour le problème de minimisation en moyenne, il s'agit de choisir le minimum entre le fait de tester et de ne pas tester :

$$
\begin{aligned}
SC(u_L, u_S, q) = min \Bigg\{ \Bigg(& p_T \cdot (u_L + u_S + q) + \frac{u_S}{u_L + u_S} SC(u_L, u_S - 1, q) \\
& + \frac{u_L}{u_L + u_S} SC(u_L - 1, u_S, q + 1) \Bigg) \; ; \\
& \Bigg(\frac{u_S}{u_L + u_S} SC(u_L, u_S - 1, q) \\
& + \frac{u_L}{u_L + u_S} \left(\Delta \cdot u_S + SC(u_L - 1, u_S, q) \right) \Bigg) \Bigg\}
\end{aligned}
\tag{2.9}
$$

avec $\forall x, z \in (0, n_L)$, $y \in (0, n_S)$: $SC(x, 0, z)$ et $SC(0, y, z) = 0$

Algorithme 2 : Optimum Prime$(n_L, n_S, p_T, p_L, p_S)$

Initialisation des valeurs de $SC[i][j][k]$ à 0 et de $Tester[i][j][k]$ à Faux
pour u_L de 1 à n_L **faire :**

pour u_S de 1 à n_S **faire :**

pour q de 0 à $n_L - u_L$ **faire :**

$a \leftarrow p_T \cdot (u_L + u_S + q) + \frac{u_S}{u_S+u_S} SC[u_L][u_S - 1][q] + \frac{u_L}{u_S+u_S} SC[u_L - 1][u_S][q+1]$

$b \leftarrow \frac{u_S}{u_S+u_S}(0 + SC[u_L][u_S - 1][q]) + \frac{u_L}{u_S+u_S}(\Delta \cdot u_S + SC[u_L - 1][u_S][q])$

si $a < b$ alors
$SC[u_L][u_S][q] \leftarrow a$
$Tester[u_L][u_S][q] \leftarrow Vrai$
sinon
$SC[u_L][u_S][q] \leftarrow b$

retourne Tester

Cet algorithme retourne un tableau (assimilable à un arbre de décision) qu'il suffit de parcourir au fur et à mesure de l'exécution pour prendre la décision optimale qui minimise le temps de complétion total moyen.

2.4.3 Une heuristique gloutonne

Algorithme 3 : L'algorithme $A_{\mathrm{A}}(n_L, n_S, p_T, p_L, p_S)$.

1. Q : la queue des tâches longues à exécuter à la fin de l'ordonnancement
2. **pour**$\{i = 1, i \leq n_T, i++\}$
   a) tester = $(\frac{p_T}{\Delta} < \frac{u_L}{u_S+u_L} \cdot \frac{u_S}{(u_S+u_L+q))})$
   b) **si** tester :
      i. **si** le test dit "court" : exécuter immédiatement
      ii. **sinon** : mettre la tâche dans Q
   c) **sinon** : exécuter la tâche immédiatement sans la tester.
3. Exécuter toutes les tâches se trouvant dans Q

On peut faire le rapprochement entre cette heuristique et les deux algorithmes optimums précédents.

Tout d'abord, cet algorithme est l'adaptation de A_{NA} pour le cas adaptatif. Si on adapte le problème initial pour dire qu'il est possible au départ d'avoir des tâches longues dans la queue ; cette heuristique est similaire à relancer l'algorithme A_{NA} à chaque tâche pour se poser uniquement la question "dois-je tester ou non la tâche suivante ?" (équivalent à "$n_T \neq 0$?").

On peut aussi remarquer que le critère de test $(\frac{p_T}{\Delta} < \frac{u_L}{u_S+u_L} \cdot \frac{u_S}{(u_S+u_L+q))})$ est le même que celui de Optimum Prime si on ne tient pas compte des différents sous-problèmes dans la formule (2.9) et qu'on considère uniquement (ce qui est inexact) :

$$SC(u_L, u_S, q) = min\left\{(p_T \cdot (u_L + u_S + q)) \; ; \; \left(\frac{u_L}{u_L + u_S}\Delta \cdot u_S\right)\right\}$$

Intuitivement, la détection d'une tâche longue "augmente le coût des tests futurs" (car elle augmente la taille de la queue). Il s'agit de cette "augmentation" qui n'est pas prise en compte par l'heuristique gloutonne, par rapport à l'algorithme de programmation dynamique.

Dans les situations presque équilibrées entre "tester" ou "ne pas tester", faire un test aurait tendance à réduire le nombre de tests futurs. Nous pouvons en conjecturer que la différence entre le moyen de tests de nos deux algorithmes est très faible. Réussir à estimer (ou borner) correctement ce nombre pourrait permettre de trouver une borne de l'écart entre l'heuristique et l'optimal adaptatif.

2.5 Simulations et Performances des algorithmes

L'objectif de cette section est de comparer les résultats obtenus par les algorithmes proposés précédemment ; et plus particulièrement de pouvoir constater l'écart entre les modèles adaptatifs et non adaptatifs.

Partant du fait qu'aucun algorithme (pouvant tester) ne pourra avoir une meilleure valeur moyenne que $OptimumPrime$, on s'intéresse à la qualité donnée par nos heuristiques gloutonnes par rapport à celui ci.

Pour des paramètres donnés (p_T, p_L, p_S, n_L et n_S) on simule les valeurs moyennes de $\sum C_j$ donné par chacun de nos algorithmes sur l'ensemble des permutations possibles des tâches courtes / longues.

Les instances testées sont tous les 5-uplets (p_T, p_L, p_S, n_L et n_S) avec :
- $p_T \in [1 : 300]$
- $p_L \in [1 : 12000]$
- $p_S = 0$
- $n_L \in [0 - 100]$
- $n_S = 100 - n_L$

2.5.1 Description du choix des instances considérées

Nous pouvons remarquer que pour nos 3 algorithmes, les composantes du surcoût ne dépendent pas indépendamment de p_S et p_L, mais toujours de $p_L - p_S = \Delta$. Le terme p_S apparaît (indépendamment de p_L) uniquement dans la valeur d'optimal donné par SPT ; composante qui apparaît à la fois au numérateur et au dénominateur du rapport entre valeur de l'algorithme et valeur optimale. Comme on souhaite mettre en évidence les pires rapport entre nos algorithmes, nous choisissons d'imposer $p_S = 0$ afin d'obtenir les rapports moyens les plus mauvais pour nous.

2.5.2 Présentation des résultats

Pour chacun des deux algorithmes gloutons présentés (A et NA) plus tôt, nous comparons sa valeur objectif moyenne obtenue avec celle obtenue grâce à l'algorithme optimal adaptatif $OptimumPrime$ en o(n^3), et présentons ces résultats dans des cartes de chaleur, où à chaque point (qui correspond à une instance $I = (n_L, n_S, p_L, p_S, p_T)$) est associée une couleur représentant la valeur du rapport moyen entre l'algorithme et l'optimal adaptatif.

Il est intéressant de noter qu'il est possible de formuler ces deux algorithmes sous la forme d'une programmation dynamique de la même façon que l'algorithme $OptimumPrime$, avec un critère de test différent, propre à chaque heuristique. En appelant ces différentes programmations dynamique avec les bons paramètres, il est possible de réutiliser les résultats de la programmation dynamique d'une instance pour dans les calculs d'une autre instance (avec p_S, p_L, p_T constants). Plus concrètement, si on regarde le pseudo-code Algorithme (2) en faisant varier tout d'abord u_L entre 0 et 100, et u_S entre 0 et $100 - u_L$, chacun des trois algorithmes (A, NA, $OptimumPrime$) génère ses valeurs pour une ligne entière des cartes de chaleurs suivantes.

Rapports moyens de l'algorithme NA

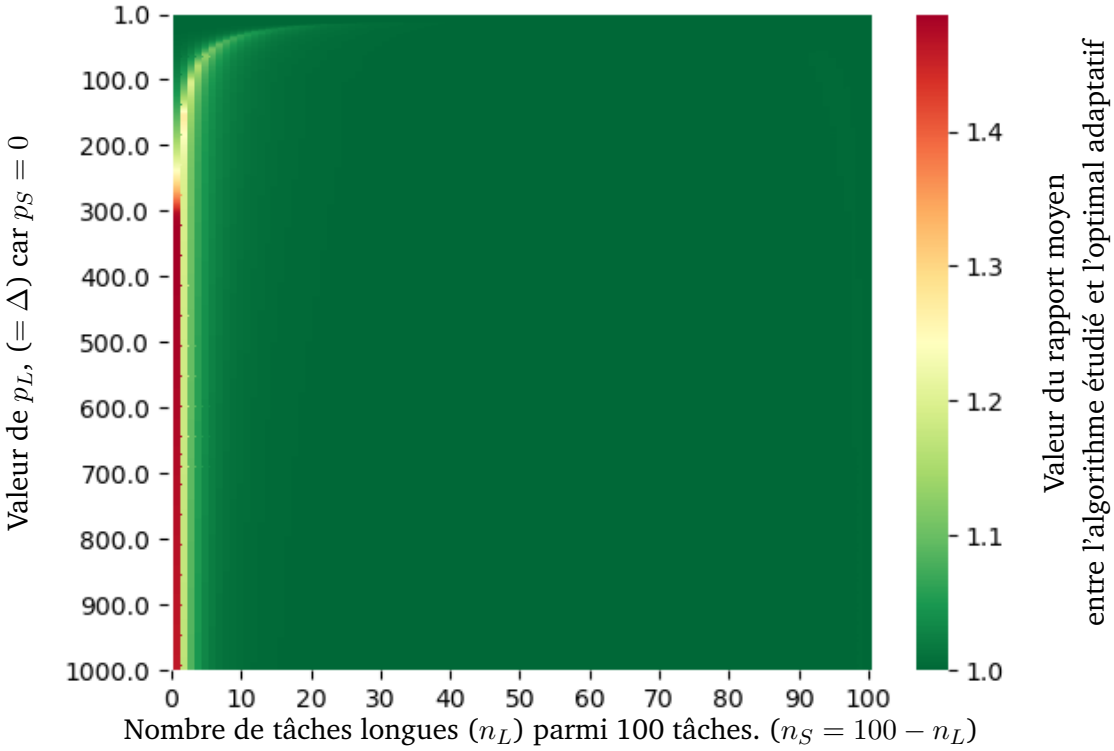


Figure 2.8. : Carte de chaleur du rapport trouvé entre la valeur de $\sum(C_j)$ trouvé par l'algorithme NA et celle trouvée par $OptimumPrime$, en faisant varier le nombre de tâches longues (n_L) parmi 100 tâches ainsi que la valeur de Δ entre 1 et 1000, pour $p_S = 0$ et $p_T = 3$.

On peut remarquer, sur la campagne de simulations réalisée avec $p_S = 0$, que le pire rapport entre l'algorithme non adaptatif et l'algorithme optimal est obtenu pour $n_L = 1$ et avec des valeurs de $\frac{\Delta}{p_T} \approx 119.91$. Avec ce rapport $\frac{\Delta}{p_T}$, le rapport entre l'algorithme non optimal et l'algorithme optimal semble tendre asymptotiquement vers 1.5.

Rapports moyens de l'algorithme A

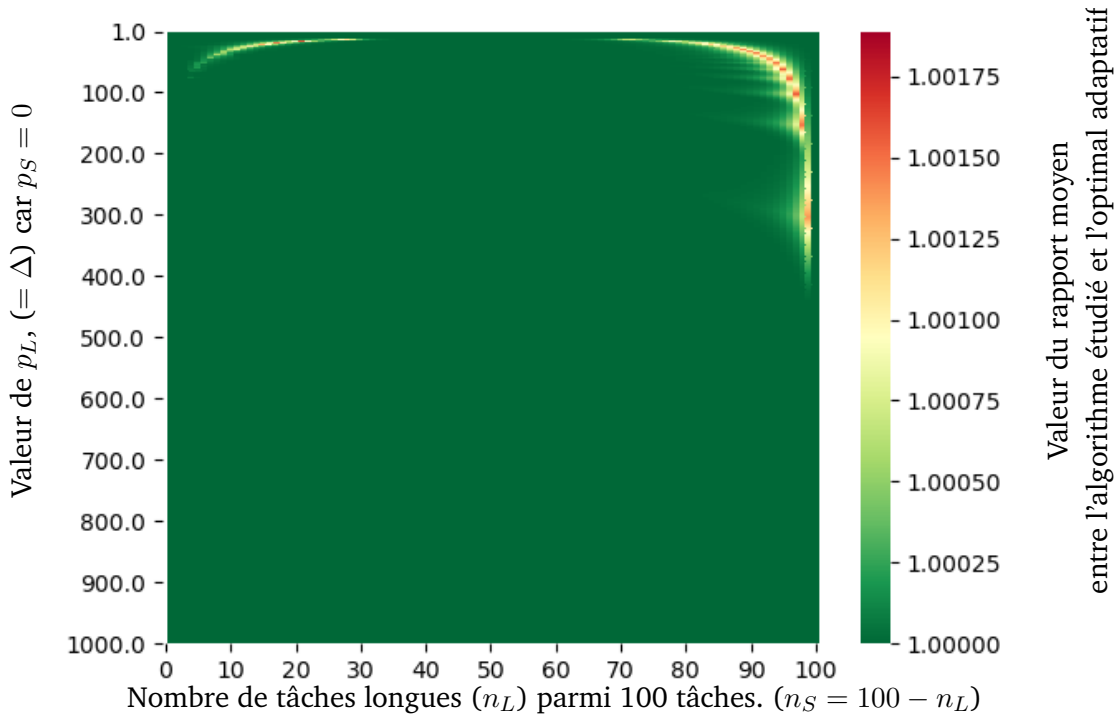


Figure 2.9. : Carte de chaleur du rapport trouvé entre la valeur de $\sum(C_j)$ trouvé par l'algorithme A et celle trouvée par $OptimumPrime$, en faisant varier le nombre de tâches longues (n_L) parmi 100 tâches ainsi que la valeur de Δ entre 1 et 1000, pour $p_S = 0$ et $p_T = 3$.

Cette carte de chaleur permet d'avoir une vue d'ensemble du comportement de l'algorithme A par rapport à $OptimumPrime$. Afin d'avoir une meilleure vision des résultats, nous nous concentrons sur la partie supérieure de cette carte de chaleur, pour de faibles valeurs de Δ, dans la figure 2.10.

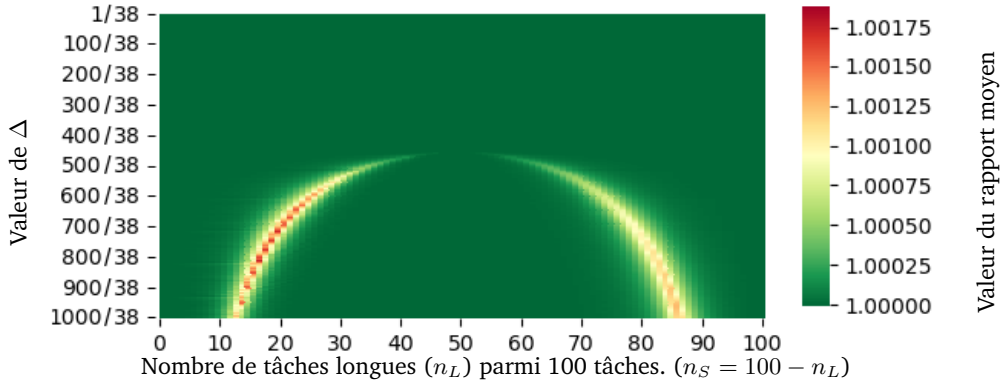


Figure 2.10. : Zoom x38 sur les résultats affichés par la carte de chaleur 2.9. Au vu des arguments présentés dans 2.5.2, ces résultats peuvent également être obtenus de la même façon que la Figure 2.9 en considérant $p_T = 3 * 38 = 114$.

Sur l'intégralité des simulations faites, la pire valeur du rapport moyen entre les résultats de l'heuristique adaptative et l'optimal adaptatifs semblent être atteints pour une valeur de $\frac{\Delta}{p_T}$ proche de 8,65. pour ce cas de figure, le rapport etre les deux algorithmes semble se rapprocher asymptotiquement de la valeur 1,002.

Bien que l'heuristique ne soit pas optimale, elle permet en pratique avec une complexité moindre d'obtenir des résultats très proches d'un algorithme optimal qui a une complexité en temps et en mémoire en o(n^3).

2.5.3 Discussions à propos des résultats

Intuitivement, bien que ces trois algorithmes aient une façon différente de décider quelles tâches tester, au vu de la définition des surcoûts faites dans la Section (2.2.1), la valeur de la fonction objectif obtenue par chaque algorithme peut s'écrire sous la forme :

$$\mathrm{SPT} + ap_T + b\Delta$$

Comme nous choisissons $p_S = 0$, donc $p_L = \Delta$, la valeur de SPT peut aussi s'écrire comme une fonction de n_L, n_S et p_L, on obtient alors que l'objectif de chaque algorithme peut s'écrire sous la forme suivante :

$$ap_T + b'\Delta$$

Le rapport entre les objectifs de deux algorithmes ($\mathcal{A}_1$ et $\mathcal{A}_2$) peut s'écrire

$$\frac{a_1 p_T + b_1 \Delta}{a_2 p_T + b_2 \Delta} = \frac{a_1 + b_1 \frac{\Delta}{p_T}}{a_2 + b_2 \frac{\Delta}{p_T}}$$

C'est pourquoi l'accent est particulièrement mis sur le paramètre $\frac{\Delta}{p_T}$. Il s'agit d'une analyse qui est valable car nous considérons $p_S = 0$ afin d'obtenir les pires rapports possibles.

Le pire scénario pour l'algorithme NA est facilement explicable : il se produit lorsqu'il y a une seule tâche longue très importante, poussant NA et $OptimumPrime$ à effectuer de nombreux tests pour la détecter, mais NA continue de tester même lorsque toutes les incertitudes ont été levées de manière dynamique. Au-delà d'une certaine limite pour Δ très élevé, toutes les tâches (sauf la dernière) sont testées par NA ; et l'augmentation de la valeur de Δ ne contribue qu'à "gommer" l'impact des p_T dans le rapport.

En revanche, nous n'expliquons pas le pire scénario pour l'algorithme A. Le cas observé expérimentalement ne semble pas avoir de caractéristiques particulières.

Être en mesure de caractériser plus en détail ces pires cas est un point manquant majeur pour pouvoir effectuer une analyse théorique approfondie des deux algorithmes adaptatifs, dans la mesure où nous disposons d'une analyse complète de l'algorithme non adaptatif, particulièrement si nous souhaitons étendre notre analyse pour répondre à la question suivante.

2.6 Robustesse et remise en cause des hypothèses

> *A partir de quel point la modélisation de ce problème devient obsolète (comparée à un algorithme ordonnancement aléatoirement) si l'oracle n'est pas parfait ?*

L'étude précédente repose sur deux hypothèses fortes, formulées dans la Section (2.1.1) ; les nombres de tâches courtes (n_S) et longues (n_L) sont connus et l'oracle a toujours raison. Nous portons notre attention sur les performances de nos algorithmes dans les cas où ces hypothèses soient inexactes.

2.6.1 Si l'oracle n'a pas raison dans 100% des cas.

Il est important de préciser que l'algorithme "croit" que l'oracle a toujours raison. Bien que ces questions soient pertinentes et pourraient être l'objet d'une extension théorique de ces travaux, nous n'explorerons pas ici ces différentes voies :

1. Si l'oracle se trompe dans 100% des cas, l'algorithme pourrait faire l'inverse de ce qu'indique l'oracle. Comment détecter à la volée le pourcentage d'erreur de l'oracle sachant que l'erreur (une tâche longue est détectée courte) peut s'identifier immédiatement car cette tâche est directement exécutée, mais que l'erreur (une tâche courte est détectée longue) ne s'identifie que trop tard ?
2. L'erreur de l'oracle peut ou non être stochastique : deux tests de la même tâche donnent-ils toujours le même résultat ? Auquel cas une politique optimale pourrait par exemple considérer tester plusieurs fois une tâche, ou confirmer avec un second test un résultat positif.

Pour nos algorithmes précédents, clarifier la façon dont peut se tromper l'oracle n'est pas pertinent dans la mesure où chaque tâche n'y est testée au plus qu'une seule fois. Nous considérons qu'à chaque appel, l'oracle a un certain pourcentage de chance de se tromper. On peut alors étudier plus en détail les différents cas qui sont détaillés dans la Figure (2.11).

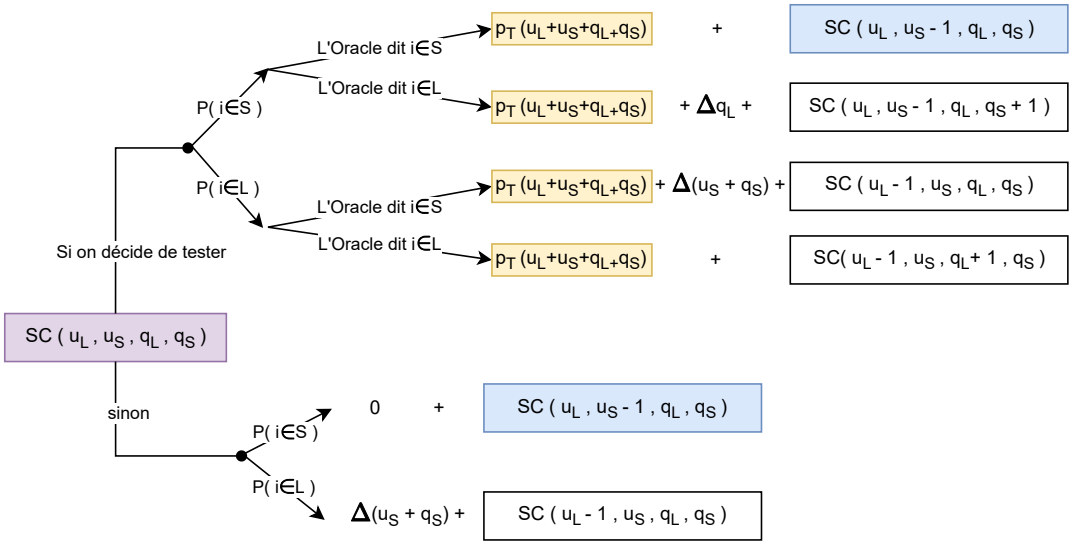


Figure 2.11. : Programmation Dynamique pour calculer la valeur moyenne de la fonction objectif de $OptimumPrime$ dans le cas où l'oracle puisse faire des erreurs dans ses prédictions avec une certaine probabilité.

Un point qui n'a pas été précisé au préalable. On considère ici que l'algorithme $OptimumPrime$ ajoute les nouvelles tâches détectées comme longues à la fin de la queue. Au moment de l'exécution des tâches longues (hormis celles se trouvant dans la queue) : on paie le décalage de toutes les tâches courtes suivantes. Si une tâche courte se trouve dans la queue après des tâches longues : on paie le décalage de la tâche courte par ces tâches longues au moment où la tâche courte est ajoutée dans la queue.

La programmation dynamique issue de la Figure (2.11) permet uniquement de calculer la valeur de l'objectif en considérant des erreurs. L'algorithme $OptimumPrime$ calcule toujours les valeurs de $Tester[u_L, u_S, q]$ en utilisant la procédure décrite dans l'Algorithme (2). A tout instant, l'algorithme $OptimumPrime$ "croit" que l'oracle à toujours raison, donc que :

— la queue contient $(q_S + q_L)$ tâches longues,

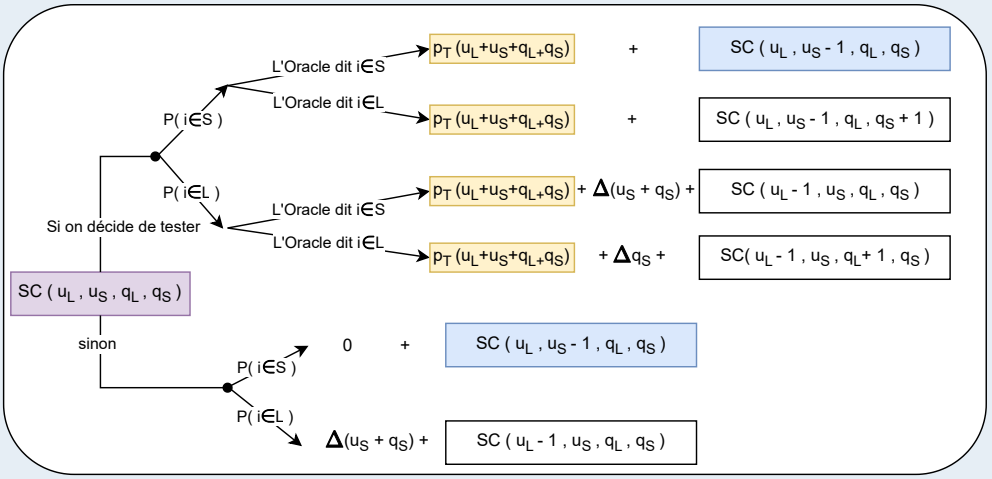
— le nombre de tâches courtes inconnues restantes vaut $(u_S + q_S)$,

— le nombre de tâches longues inconnues restantes vaut $(u_L - q_S)$.

Afin de calculer la valeur de l'objectif avec erreurs de l'oracle, il faudra d'une part calculer le tableau $Tester$ issu de l'Algorithme (2) pour construire l'arbre de décision qu'aurait décidé de suivre $OptimumPrime$, et s'en servir pour choisir laquelle des deux branches emprunter dans la Figure (2.11). Il s'agira donc de suivre la décision de $Tester[(u_L - q_S), (u_S + q_S), (q_S + q_L)]$.

Note de l'auteur :

Tout comme l'illusion de silhouette de Nobuyuki Kayahara, ces équations peuvent sembler peu naturelles au premier abord.
Si vous vous sentez plus à l'aise avec l'autre approche, vous pouvez considérer que l'algorithme ajoute les tâches au début de la queue, et en déduire ce schéma de récursion, qui peut sembler plus naturel ; et qui donne le même résultat que celui issu de la Figure (2.11) si on l'applique à une Instance du problème.



Attention, comme la décomposition faite ici n'est pas la même que dans la Figure (2.11), la signification du terme SC s'en retrouve modifiée, et la valeur d'une étape intermédiaire du calcul fait ici ne doit pas être confondue (et n'est pas comparable) avec la valeur intermédiaire du calcul issu de la Figure (2.11).

Il est temps d'étudier les performances de nos algorithmes lorsque l'oracle a une certaine probabilité de se tromper. Comme les algorithmes présentés ne testent qu'une seule fois chaque tâche, notre modèle n'est pertinent que pour un oracle ayant une fiabilité supérieure à 50%. Nous testerons la fiabilité de nos algorithmes pour des oracles ayant 100%, 95%, 90%, ... jusque 50% de fiabilité.

On expose les résultats obtenus sur les deux instances mises en évidence dans la Section (2.5.2) en utilisant les rapports de $\frac{\Delta}{p_T}$ des pires instances détectées pour les algorithmes NA et A :

— $p_T = 100$ et $\Delta = 19991$: pire valeur pour l'algorithme non adaptatif,

— $p_T = 100$ et $\Delta = 865$: pire valeur pour l'algorithme adaptatif.

Par soucis de lisibilité, les valeurs des objectifs obtenus par les trois algorithmes sur ces instances (avec erreurs de l'oracle) sont chacune divisées par la valeur d'$OptimumPrime$ avec oracle à 100% fiable. Chaque courbe (bleue) représentant l'optimum commencera donc au point (fiabilité=100% , rapport = 1).

Pire valeur pour l'algorithme non adaptatif NA **:** $p_T = 100$ **et** $\Delta = 19991$ **:**

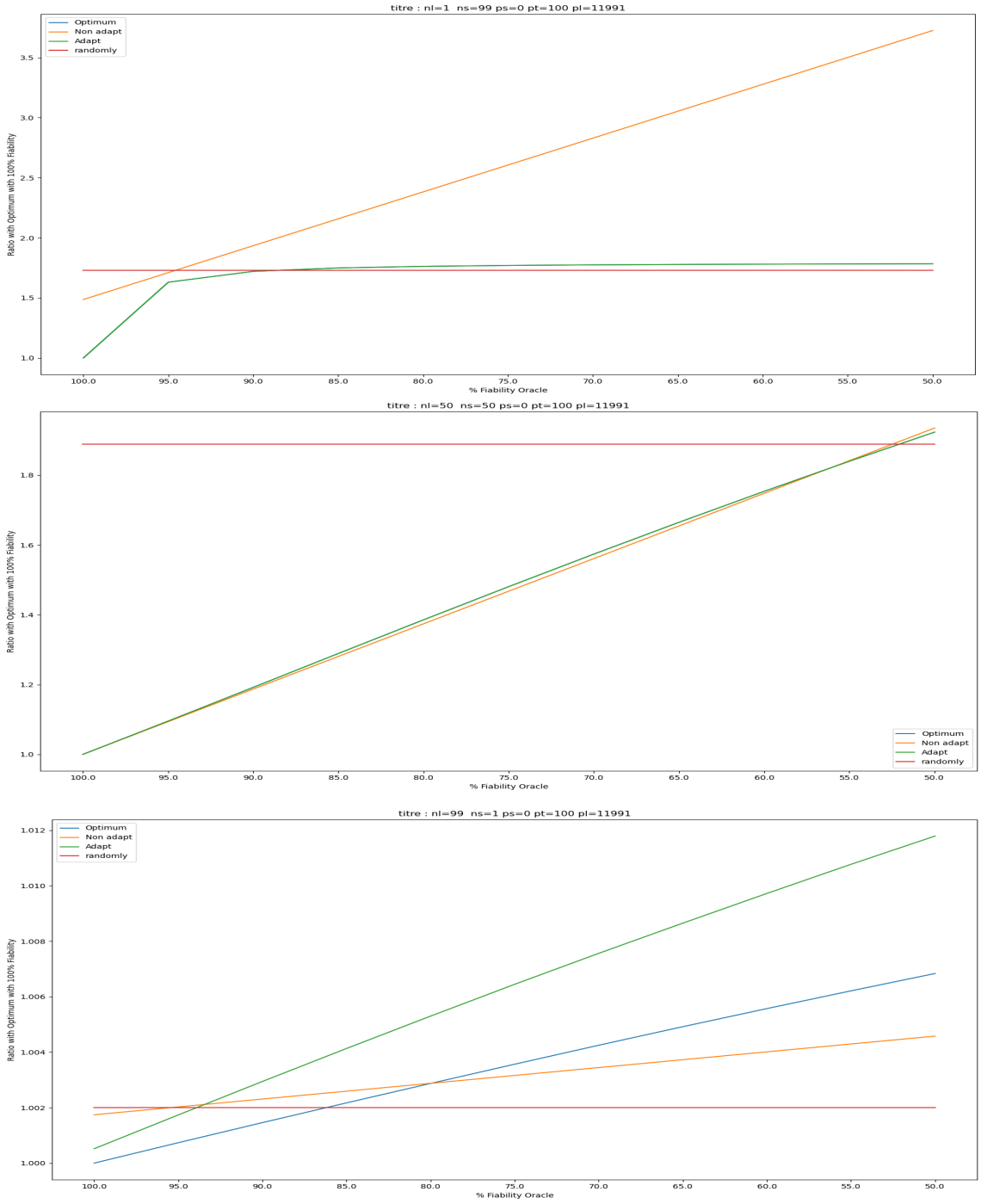


Figure 2.12. : Évolution du rapport issu des trois algorithmes en fonction du pourcentage d'erreur de l'oracle : $OptimumPrime$ en bleu, A en vert, NA en Orange, $\mathcal{A}_{\text{aucunTest}}$ en rouge (horizontale) ; pour $p_T = 100$, $\Delta = 19991$ et pour
1) $n_L = 1$, $n_S = 99$
2) $n_L = 50$, $n_S = 50$
3) $n_L = 99$, $n_S = 1$

Pire valeur pour l'algorithme non adaptatif NA **:** $p_T = 100$ **et** $\Delta = 865$ **:**

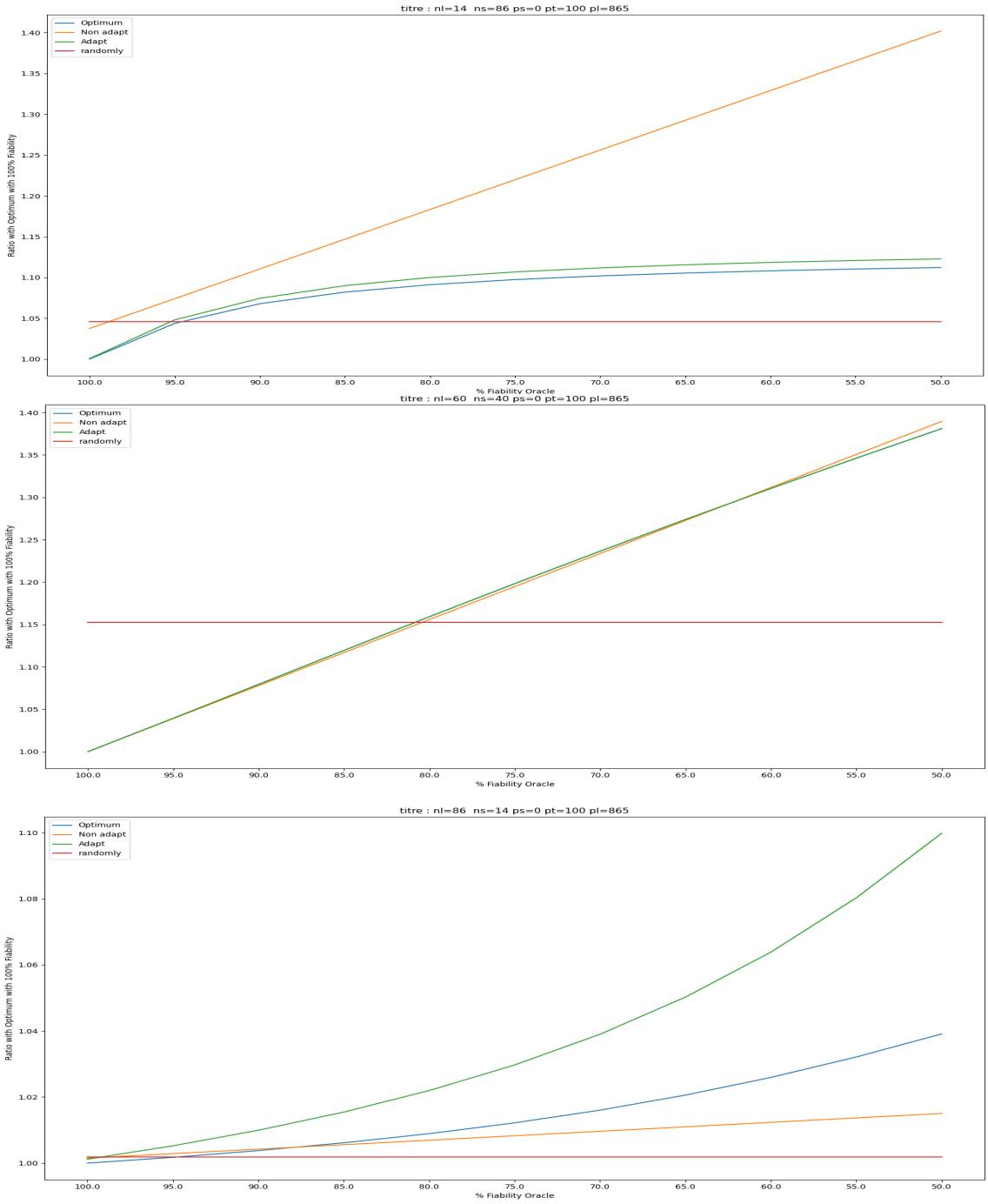


Figure 2.13. : Évolution du rapport entre l'objectif des trois algorithmes avec erreurs d'oracles par rapport à $OptimumPrime$ sans erreur, en fonction du pourcentage d'erreur de l'oracle : $OptimumPrime$ en bleu, A en vert, NA en Orange, $\mathcal{A}_{\text{aucunTest}}$ en rouge (horizontale) ; pour $p_T = 100$, $\Delta = 865$ et pour
1) $n_L = 14$, $n_S = 86$
2) $n_L = 60$, $n_S = 40$
3) $n_L = 86$, $n_S = 14$

2.6.2 Si le nombres de tâches longues et courtes est méconnu

Nous étudions désormais la résilience de nos algorithmes en cas de violation de notre deuxième hypothèse, à savoir la connaissance exacte du nombre de tâches longues. Pour ce faire, pour les algorithmes adaptatifs, il s'agit encore une fois d'utiliser une case "erronée" du tableau Tester généré par la version programmation dynamique des deux algorithmes ; et pour l'algorithme non adaptatif, il suffit de calculer de "mauvaises" valeurs de n_T.

Nous utilisons les mêmes scénarios défavorables que précédemment pour l'une ou l'autre des deux heuristiques .

Pire valeur pour l'algorithme non adaptatif A **:** $p_T = 100$ **et** $\Delta = 19991$ **:**

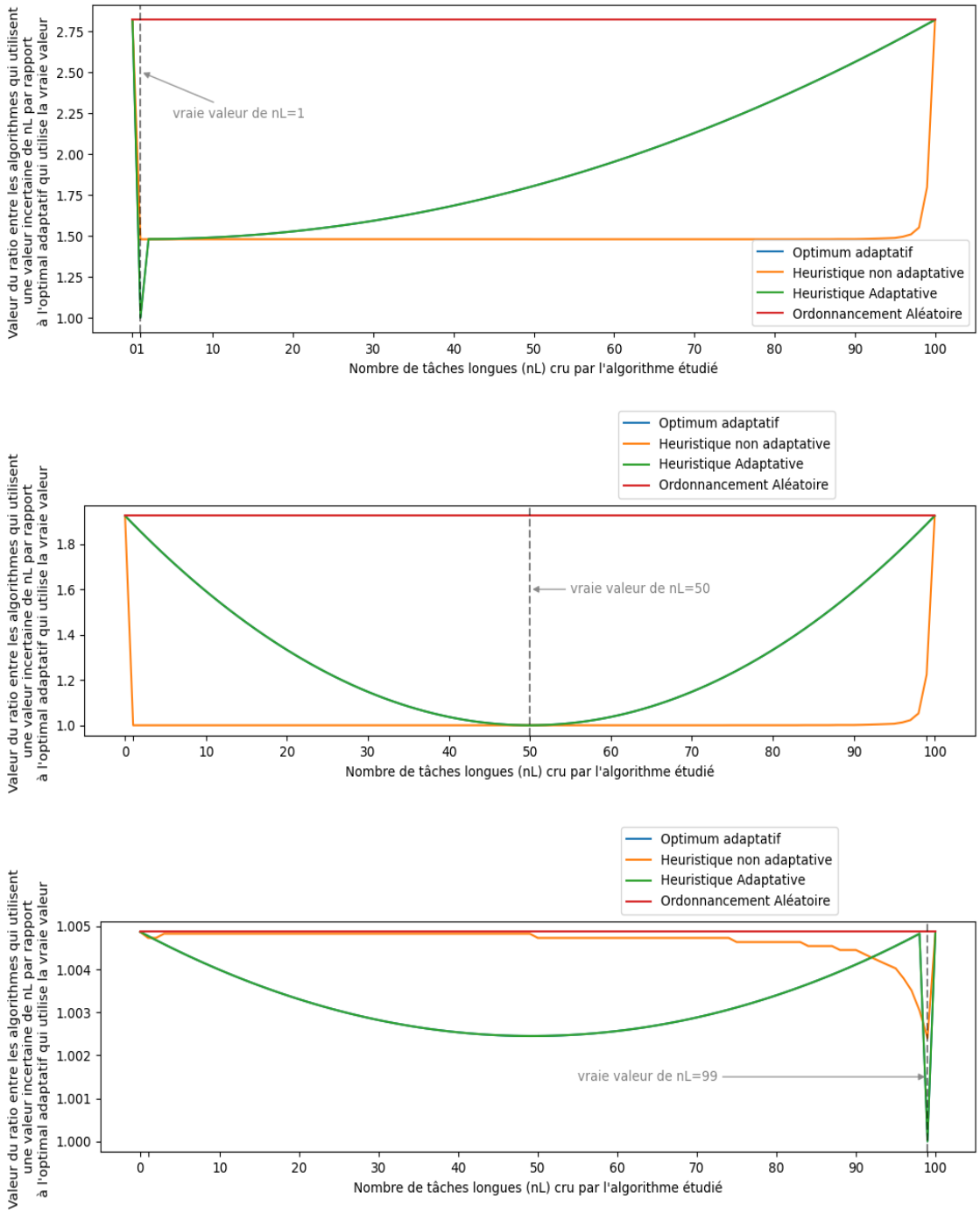


Figure 2.14. : Evolution du rapport entre l'objectif des trois algorithmes avec erreurs sur n_L par rapport à $OptimumPrime$ avec connaissance exacte de n_L, pour $p_T = 100$, $\Delta = 865$ et pour
1) $n_L = 1$, $n_S = 99$
2) $n_L = 50$, $n_S = 50$
3) $n_L = 99$, $n_S = 1$

Pire valeur pour l'algorithme adaptatif A **:** $p_T = 100$ **et** $\Delta = 865$ **:**

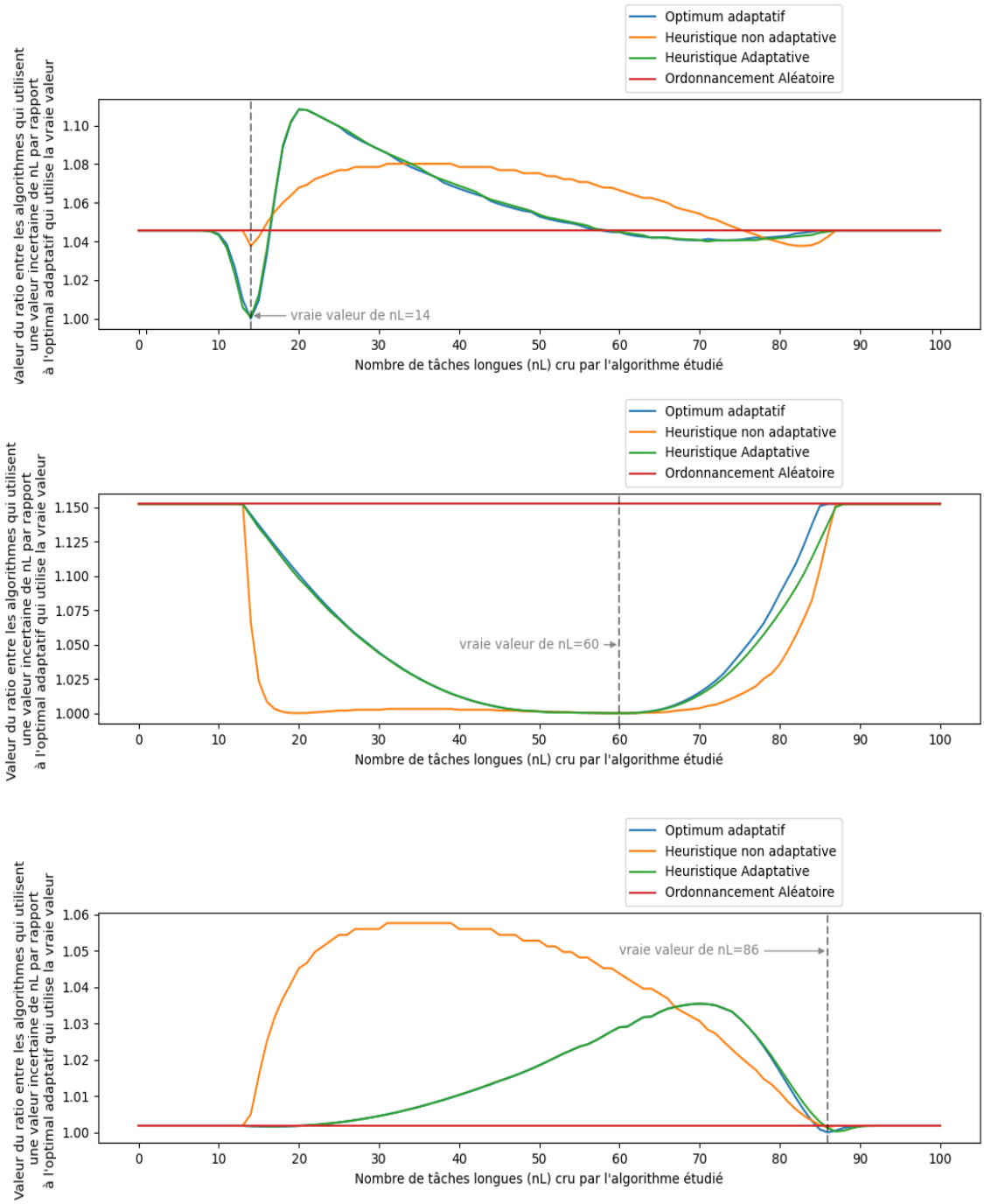


Figure 2.15. : Evolution du rapport entre l'objectif des trois algorithmes avec erreurs sur n_L par rapport à $OptimumPrime$ avec connaissance exacte de n_L, pour $p_T = 100$, $\Delta = 865$ et pour
1) $n_L = 14$, $n_S = 86$
2) $n_L = 60$, $n_S = 40$
3) $n_L = 86$, $n_S = 14$

En conclusion, l'étude de la robustesse de nos algorithmes montre des résultats mitigés. En ce qui concerne les erreurs d'oracle, le comportement des trois algorithmes semble similaire, que l'on soit par défaut dans des cas plus défavorables à l'heuristique non adaptative ou à l'heuristique adaptative, avec un avantage conséquent pour NA pour de grandes valeurs de n_L. Bien que nos algorithmes adaptatifs obtiennent des résultats moins bons qu'un ordonnanceur aléatoire si l'oracle a une grande probabilité d'erreur ; notre hypothèse initiale d'un oracle parfait ne semble pas trop restrictive si on la met en perspective des performances des modèles d'IA utilisés dans la pratique.

Cependant, les résultats sont beaucoup moins clairs en ce qui concerne les erreurs de proportion sur le nombre de tâches longues, en dehors de l'effet de bord de ne jamais effectuer de tests si l'on croit qu'aucune tâche longue n'est présente dans l'ordonnancement. Les rapports observés s'éloignent en revanche des valeurs de performances présentées en premier lieu dans la Section (2.5).

> *Voila pourquoi le futur work est super intéressant.*
>
> **— Fanny Dufossé, 25/04/2023**

2.7 Conclusions et perspectives

Nous présentons trois algorithmes : un algorithme optimal non adaptatif dont nous prouvons l'optimalité, un algorithme optimal adaptatif basé sur une programmation dynamique, ainsi qu'une heuristique gloutonne performante dans le cadre adaptatif. Bien que nous ayons fourni une analyse complète pour l'algorithme non adaptatif, des questions subsistent quant aux valeurs des fonctions objectif des algorithmes adaptatifs.

Nous montrons brièvement que l'hypothèse selon laquelle l'oracle ne commet pas d'erreur n'est pas très restrictive, mais que la connaissance des valeurs de n_S et n_L semble être un paramètre bien plus important.

Ce travail sur ce modèle ouvre plusieurs perspectives de recherche, notamment du point de vue de la protection de la vie privée, où il faut également minimiser le nombre de recours à l'oracle, donc rentrer sur les sentiers de l'optimisation multi-objectif. La conception d'un algorithme optimal via programmation dynamique (en $o(n^4)$) semble étonnamment plus simple que la recherche d'heuristiques gloutonnes, car les raisons qui rendaient difficile l'analyse de nos algorithmes adaptatifs semblent devenir des critères qui compliquent la conception de telles heuristiques.

Une piste prometteuse serait d'étudier les différentes manières pour l'oracle de se tromper, en incluant les aspects tels que les faux positifs, les vrais négatifs, etc., dans les pourcentages de fiabilité de l'oracle. Intégrer également un aspect stochastique à l'oracle (le fait de tester deux fois la même tâche peut donner deux résultats différents) ouvre des perspectives intéressantes dans la conception d'algorithmes.

Dans le chapitre (5), nous nous intéressons à la minimisation du pire cas (sur la pire permutation) des dates de complétion cumulées, pour des modèles proches de celui-ci.

3 Problématique Bi-agent : Intégrer des tâches prioritaires en gardant le contrôle sur la qualité de service

Ce chapitre est basé sur des travaux en collaboration avec Clément Mommessin, Giorgio Lucarelli et Denis Trystram, publiés dans Europar [Fag+22].

Ce chapitre de thèse se concentre sur les systèmes informatiques composés de plusieurs unités connectées à des capteurs tels que présentés dans l'introduction (Sous-Section (1.3.1)). Il existe deux types de tâches à exécuter, celles produites par les capteurs et celles provenant de l'extérieur, pour lesquelles nous avons des contraintes et des objectifs différents. Les données produites par les capteurs devraient être traitées et analysées localement et n'ont d'intérêt que pour leur environnement local. Le problème est ici de combiner des tâches inconnues à traiter en urgence dans un planning préétabli. Nous présentons un algorithme compétitif pour lequel nous autorisons une augmentation de la vitesse et qui sera autorisé à rejeter certaines tâches, qui seront exécutées sur une plateformes extérieure ou remises dans un batch ultérieur. Nous analysons également les performances de cet algorithme en utilisant une approche de *dual fitting* (qui est présentée Sous-Section (3.5.1)). Son rapport de compétitivité dépendra des paramètres de l'augmentation de vitesse et du rejet des tâches, qui doivent être fixés par l'utilisateur.

3.1 Introduction

Dans le cas d'un seul agent, le problème de minimisation du temps d'écoulement online peut être résolu de manière optimale sur une seule machine si les préemptions sont autorisées, mais il est difficile de l'approximer avec $m \geq 2$ machines [LR07], ou dans le cas d'un calcul offline si les préemptions ne sont pas autorisées [KTW99 ; CKZ01]. Ces résultats d'inapproximabilité ont conduit à analyser le problème dans

le contexte de l'augmentation des ressources, où plus de puissance est donnée à l'algorithme, comme par exemple en autorisant l'algorithme à utiliser plus de processeurs ou une vitesse plus élevée que l'optimal. Dans le cas de l'augmentation de vitesse, nous supposons que l'algorithme utilise des machines avec une vitesse de $1+\epsilon$, tandis que la solution optimale est basée sur une vitesse de machine de 1, où $\epsilon \geq 0$ est une constante. En utilisant ces modèles d'augmentation des ressources, nous pouvons obtenir des garanties de performance qui dépendent de la valeur de ϵ [Phi+97 ; KP00 ; Cho+18].

Avec l'introduction de tâches provenant de l'agent global, nous montrons dans ce chapitre qu'il est difficile d'approximer l'objectif du flowtime total pour les tâches locales, même si leur préemption et l'augmentation des ressources est utilisée. Plus précisément, tout algorithme onligne qui utilise des machines à vitesse $(1+\epsilon_s)$ devrait rejeter au moins k tâches globales pour obtenir un rapport de compétitivité en $O\left((1-\frac{2k}{n^g})\mathcal{W}\right)$, où $\mathcal{W}$ est le rapport entre la charge de travail totale des tâches globales et la charge de travail totale des tâches locales.

D'un point de vue positif, nous proposons un algorithme pour résoudre le problème de planification à deux agents sous le modèle d'augmentation des ressources, avec à la fois rejet et augmentation de la vitesse.

Ensuite, nous analysons le rapport de compétitivité de l'algorithme en utilisant l'approche du dual fitting [AGK12] qui sera vulgarisée dans la Sous-Section 3.5.1 suivante. En particulier, nous prouvons que notre algorithme est à vitesse $(1+\epsilon_s)$ et est $\max\left\{\frac{\mathcal{W}}{\epsilon_s\epsilon_r}+\frac{1+\epsilon_s}{2\epsilon_s}, \frac{1+\epsilon_s}{\epsilon_s}\right\}$-compétitif en rejetant une fraction de tâches globales en fonction d'un paramètre ϵ_r, où $\epsilon_s > 0$ et $0 < \epsilon_r < 1$.

L'organisation de ce chapitre est la suivante : nous commençons par définir formellement le problème de planification à deux agents dans la section 3.2, en présentant les notations utilisées tout au long du chapitre. Ensuite, la section 3.3 met en évidence les limites du problème, en montrant qu'il existe un rapport d'inapproximation qui dépend des paramètres d'augmentation choisis. Dans la section 3.4, nous proposons un algorithme pour résoudre ce problème et nous l'analysons en termes de compétitivité en utilisant l'approche de dual fitting dans la section 3.5. La section 3.6 marque la fin de ce chapitre avec des observations, des discussions et quelques suggestions pour les travaux futurs.

3.2 Définition formelle

input : un nombre m de machines parallèles (identiques), un ensemble de tâches globales noté $\mathcal{G}$, un ensemble de tâches locales noté $\mathcal{L}$, deux paramètres : ϵ_r et ϵ_s tels que $0 < \epsilon_r < 1$ et $\epsilon_s > 0$.
L'objectif est de minimiser $\sum_{j \in \mathcal{L}} C_j$.

Les tâches locales arrivent dynamiquement (de façon online). Leur date d'arrivée (r_j) n'est pas connue à l'avance, mais leur temps d'exécution p_j est connu dès lorsqu'elles sont libérées. Toutes ces tâches arrivent avant une certaine date $d^{\mathcal{G}} \geq r_j$. Les tâches locales peuvent être préemptées, mais leur migration n'est pas autorisée.

Les tâches globales sont offline et disponibles dès le début de l'ordonnancement (pas de relation de précédence, $r_j = 0$). Le temps d'exécution p_j de ces tâches est connu. Chacune de ces tâches doit avoir une date de complétion inférieure à la valeur $d^{\mathcal{G}}$ (deadline globale). Une tâche globale ne peut être ni préemptée, ni migrée.

En raison des fortes bornes inférieures, nous proposons un algorithme qui utilise des mécanismes d'augmentation des ressources pour résoudre le problème, comme abordé dans (1.3.1). Le rapport de compétitivité ρ de cet algorithme augmenté y est défini de la manière suivante, avec Π l'ensemble des instances I pour ce problème :

$$\max_{I \in \Pi} \left\{ \frac{\text{résultat de l'algorithme online augmenté } (I)}{\text{résultat optimal offline non augmenté}(I)} \right\} \leq \rho$$

Il est important de noter que ces deux algorithmes (l'algorithme augmenté et l'algorithme optimal) ne jouent pas selon les mêmes règles. Alors que l'algorithme optimal doit respecter scrupuleusement toutes ces contraintes (décrites dans 3.2), à l'exception de celle concernant les tâches locales (qui sont considérées hors ligne par l'optimal), l'algorithme augmenté bénéficie de certaines libertés (ou pouvoirs) supplémentaires.

1) L'algorithme augmenté a le droit de *rejeter* un certain nombre de tâches globales : c'est à dire qu'il n'est pas obligé de toutes les exécuter (ce nombre sera défini en fonction du paramètre ϵ_r).
2) L'algorithme augmenté bénéficie d'une "accélération matérielle". Les machines qu'il utilise sont plus rapides que celles utilisées par l'optimal auquel on se compare : une tâche j (exécutée en un temps p_j par l'algorithme optimal) sera exécutée en un temps $\frac{p_j}{1+\epsilon_s}$ par l'algorithme augmenté.

3.2.1 Définitions et notations

Les caractéristiques des deux agents et de la plateforme considérées dans cette modélisation ont été justifiées dans la Sous-Section1.3.1. Plus formellement, nous considérons un ensemble $\mathcal{M}$ de m machines identiques sur lesquelles sont exécutées deux ensembles de tâches séquentielles ayant des paramètres différents.

Le premier ensemble $\mathcal{L}$ est composé de $n^{\mathcal{L}}$ tâches locales qui sont soumises de façon online, tandis que le second ensemble $\mathcal{G}$ est composé de $n^{\mathcal{G}}$ tâches globales qui sont disponibles (offline). Seules les tâches locales peuvent être interrompues (préemptées), mais les migrations entre les machines ne sont pas autorisées.

Pour une tâche donnée j, nous notons r_j sa date d'arrivée dans le système et p_j son temps de traitement sur une machine, connus uniquement au moment où la tâche arrive dans le système. On pourra dire que la date d'arrivée dans le système de toutes les tâches globales est 0, car elles sont offline.

Dans un ordonnancement, nous notons F_j le *flowtime* d'une tâche locale j, défini comme la différence entre sa date de complétion et sa date d'arrivée. L'objectif est de minimiser la somme des flowtime de toutes les tâches locales, sous la contrainte que toutes les tâches globales soient terminées avant une échéance commune $d^{\mathcal{G}}$. Nous considérons également que la date de début de toutes les tâches locales est bornée par $d^{\mathcal{G}}$.

Il s'agit de choisir une deadline pour les tâches globales qui soit réaliste avec le modèle proposé. Nous ne souhaitons pas que l'ordonnancement des tâches globales (seules) soit un problème difficile.

Dans *Bounds on Multiprocessing Timing Anomalies* [Gra69], Graham étudie des algorithmes de liste sur des ressources parallèles. Un algorithme de liste fonctionne de la façon suivante : toutes les tâches disponibles sont mises dans une liste, et dès qu'une ressource est disponible (n'exécute aucune tâche), l'algorithme planifie l'exécution de la première tâche de la liste sur cette machine. Ainsi, à moins que la liste des tâches à traiter ne soit vide, toutes les ressources sont occupées. Il existe de nombreux algorithmes de liste (SPT, LPT, EDD, ...) qui reposent sur des règles de priorité au sein de la liste (mettre les tâches les plus courtes au début de la liste, à la fin,...). Graham prouve que tous les algorithmes de liste ont un rapport d'approximation de $2-\frac{1}{m}$ (avec m le nombre de ressources parallèles) si on considère l'objectif de la minimisation du *makespan*.

En se basant sur la preuve du Théorème 1 ([Gra69]), et en considérant qu'il n'existe pas de relation de précédence entre les tâches (le chemin critique correspond à la

tâche la plus longue), on peut s'arrêter à un état intermédiaire des calculs de la preuve de cette borne $(2 - \frac{1}{m})$ et affirmer que :

$$C_{max}^{\text{Algorithme de Liste}} \leq \frac{1}{m} \cdot \left(\sum_{\forall\, j} p_j \; - \max_{\forall\, j} p_j \right) + \max_{\forall_j} p_j. \tag{3.1}$$

Ainsi, on peut en déduire que si un algorithme de liste ordonnance les tâches sur une plateforme de ressources parallèles, nous avons la garantie que toutes ces tâches termineront leur exécution avant la deadline donnée par la borne (3.1).

Nous utilisons la formule suivante pour définir cette échéance :

$$d^{\mathcal{G}} = \frac{1}{m} \cdot \left(\sum_{j \in \mathcal{G}} p_j \; - \max_{j \in \mathcal{G}} p_j \right) + \max_{j \in \mathcal{G}} p_j + \sum_{j \in \mathcal{L}} p_j.$$

Cela garantit qu'un calendrier qui intercale des tâches locales et globales est toujours faisable. Cependant, étant donné que les tâches locales sont lancées au fil du temps, la valeur de $d^{\mathcal{G}}$ n'est pas connue à l'avance par l'algorithme.

La formule choisie pour la deadline globale simplifie la compréhension de la modélisation. Il s'agit de la borne de Graham à laquelle on ajoute le retard maximal qu'une tâche globale peut subir (en supposant que son exécution ait été retardée par toutes le tâches locales). Cela permet d'utiliser un algorithme de liste pour les tâches globales. Nous discutons plus en détail de cette borne dans la conclusion (3.6) pour montrer que sans perte de généralité, on peut considérer un ordonnancement plus précis pour les tâches globales, voire même, les considérer online.

Dans le modèle d'augmentation des ressources, nous introduisons les coefficients d'augmentation de la vitesse $\epsilon_s > 0$ et de rejet $0 < \epsilon_r < 1$. Un algorithme pour résoudre le problème abordé peut utiliser des machines avec une vitesse $1 + \epsilon_s$ fois plus rapide que celle d'un adversaire optimal et peut rejeter un nombre de tâches globales limité par une fraction ϵ_r du nombre de tâches locales.

Nous définissons ci-dessous quelques notations supplémentaires utilisées dans les prochaines sections. Dans un planning partiel, $\mathcal{Q}_i^{\mathcal{L}}(t)$ désigne l'ensemble des tâches locales affectées à la machine i, mais non terminées au moment t.

Cet ensemble comprend toutes les tâches locales en attente d'exécution sur la machine i, ainsi que la tâche en cours d'exécution à l'instant t, si elle est locale.

Notez que, nous supposons qu'une tâche j qui arrive à l'instant r_j que l'on décide d'affecter à la machine i va immédiatement être ajoutée à $\mathcal{Q}_i^{\mathcal{L}}(r_j)$.

Comme la préemption des tâches locales est autorisée, $p_j^{rem}(t)$ désigne le temps de traitement restant de la tâche j à l'instant t.

Enfin, pour une instance donnée du problème, nous désignons par $\mathcal{W} = \frac{\sum_{j\in\mathcal{G}} p_j}{\sum_{j\in\mathcal{L}} p_j}$ le rapport entre la charge de travail totale des tâches globales et la charge de travail totale des tâches locales.

3.3 Résultats négatifs : une borne inférieure

Théorème 3.3.1. *Soient $\epsilon_s \leq \frac{1}{3} \cdot \frac{\mathcal{W}-3}{3\mathcal{W}+3}$ et $k \in 1, 2, \ldots, \frac{n^{\mathcal{G}}}{2}$, où $n^{\mathcal{G}}$ est le nombre de tâches globales. Tout algorithme online qui utilise des machines à vitesse $(1+\epsilon_s)$ devrait rejeter au moins k tâches globales pour avoir un rapport de compétitivité de $O\left((1-\frac{2k}{n^{\mathcal{G}}})\mathcal{W}\right)$.*

Preuve.

Nous considérons une instance avec une seule machine, comprenant $2Z$ tâches globales qui sont réparties également en deux ensembles de tâches ayant des durées de traitement respectives de $X/3$ et $2X/3$. Nous divisons le temps en Z phases, chacune de longueur $X+1$: la phase ℓ correspond à l'intervalle de temps $[(\ell-1)(X+1), \ell(X+1))$, pour chaque $\ell = 1, 2, \ldots, Z$.

Nous notons $b_\ell = (\ell-1)(X+1)$, $1 \leq \ell \leq Z$, le début de la phase ℓ. Pour bien distinguer les phases, un burst de Y tâches locales de durée de traitement 0 est soumis de façon online à chaque instant $\ell(X+1)$, $1 \leq \ell \leq Z$. De plus, dans chaque phase, une seule tâche locale de durée d'exécution 1 arrive de façon online ; et dont la date d'arrivée dans le système sera définie plus tard. Nous avons deux cas à considérer :

Cas 1 : Si l'algorithme en ligne décide d'exécuter une tâche globale de durée de traitement $X/3$ (disons G_1) et une tâche globale de durée d'exécution $2X/3$ (disons G_2) dans chaque phase ℓ, $1 \leq \ell \leq Z$.

Si l'algorithme décide d'exécuter G_1 avant G_2, alors le moment le plus tôt auquel l'exécution de G_1 peut se terminer dans le calendrier de l'algorithme est $b_\ell + \frac{X}{3(1+\epsilon_s)}$, car l'algorithme a accès à une machine qui exécute les tâches à la vitesse $1+\epsilon_s$. Après la fin de G_1, l'algorithme démarrera G_2 à un instant $t \geq b_\ell + \frac{X}{3(1+\epsilon_s)}$.

Ensuite, l'adversaire décide de libérer la tâche locale L à l'instant $b_\ell + \frac{2X}{3}$ et le flowtime de L sera au moins $\frac{X-2X\epsilon_s+3}{3(1+\epsilon_s)}$ si G_2 n'est pas rejetée ; sinon, le flowtime de L sera $\frac{1}{1+\epsilon_s}$. La solution optimale exécute les tâches dans l'ordre G_2, L et G_1, ce qui a un flowtime de 1.

Si l'algorithme décide d'exécuter G_2 avant G_1, alors on définit $t \geq b_\ell$ comme étant le début de l'exécution de G_2. Ensuite, l'adversaire décide de libérer la tâche locale L au temps $\frac{X}{3}$ et le flowtime de L sera d'au moins $\frac{X-X\epsilon_s+3}{3(1+\epsilon_s)}$ si G_2 n'est pas rejeté ; sinon le flowtime de L sera de $\frac{1}{1+\epsilon_s}$. La solution optimale exécute les tâches dans l'ordre G_1, L et G_2, résultant un flowtime de 1.

En supposant que l'algorithme décide de rejeter la tâche correspondante G_2 exactement dans k phases, alors son flowtime total est au moins $\frac{k}{1+\epsilon_s} + (Z-k)\frac{X-2X\epsilon_s+3}{3(1+\epsilon_s)} = (Z-k)\frac{\mathcal{W}(1-2\epsilon_s)}{3(1+\epsilon_s)} + \frac{Z}{1+\epsilon_s}$, étant donné que $\mathcal{W} = X$. D'autre part, le flowtime total de l'optimal est de Z.

En rejetant k tâches globales, le rapport de compétitivité sera au moins $\frac{Z-k}{Z}\frac{\mathcal{W}(1-2\epsilon_s)}{3(1+\epsilon_s)} + \frac{1}{1+\epsilon_s} = \Omega\left(\frac{Z-k}{Z}\mathcal{W}\right)$.

Cas 2 : Sinon, il y a une phase pendant laquelle deux tâches globales de durée de traitement $2X/3$ sont partiellement exécutées.

Il existe donc une phase ℓ, au début de laquelle une tâche globale démarrée lors de la phase précédente $\ell - 1$ sera exécutée pendant au moins $q = \frac{1}{1+\epsilon_s}\frac{4X}{3} - (X+1)$ temps.

Ensuite, le groupe de tâches locales arrivées au temps $(\ell-1)(X+1)$ aura un flowtime total d'au moins qY. Notez que dans ce cas, l'arrivée de tâches locales de durée d'exécution 1 n'est pas importante et nous pouvons supposer sans perte de généralité que l'algorithme les exécutera toutes dès leur arrivée, obtenant un flowtime total pour elles égal à Z. D'autre part, la solution optimale aura comme dans le cas précédent un flowtime total égal à Z.

Par conséquent, le rapport de performance dans ce cas sera $\Omega(Y)$, avec Y pouvant être choisi suffisamment grand pour que le premier cas domine.

□

3.4 Un Algorithme Augmenté pour ce problème Bi-Agent

L'algorithme est désigné par *Augmenté*, il est décrit comme suit.

Initialisation : Au départ, nous avons une file d'attente de tâches globales $\mathcal{Q}^{\mathcal{G}}$ initialisée avec toutes les tâches globales triées dans un ordre arbitraire. Pour chaque machine $i \in \mathcal{M}$, nous initialisons une file d'attente de tâches locales vide $\mathcal{Q}_i^{\mathcal{L}}$.

Allocation des tâches locales : Lors de la soumission d'une tâche locale j, nous l'allouons à la machine i qui minimise

$$\lambda_{ij} = \sum_{l \in \mathcal{Q}_i^{\mathcal{L}}(r_j):\ p_l^{rem}(r_j) \leq p_j} p_l^{rem}(r_j) + \sum_{l \in \mathcal{Q}_i^{\mathcal{L}}(r_j):\ p_l^{rem}(r_j) > p_j} p_j \tag{3.2}$$

et notons par λ_j cette quantité. Intuitivement, nous mettons la tâche j sur la machine qui minimise l'augmentation du flowtime total due uniquement aux tâches dans $\mathcal{Q}_i^{\mathcal{L}}$.

Exécution des tâches : Pour chaque machine i, les tâches dans $\mathcal{Q}_i^{\mathcal{L}}$ sont exécutées dans l'ordre du temps restant le plus court. (*Shortest Remaining Processing Time*)
Notez que si une tâche arrivée récemment est plus courte que la tâche locale actuellement en cours d'exécution, une préemption a lieu et la partie restante de la tâche courante est mise à nouveau dans $\mathcal{Q}_i^{\mathcal{L}}$.
Si $\mathcal{Q}_i^{\mathcal{L}}$ devient vide, nous enlevons une tâche de $\mathcal{Q}^{\mathcal{G}}$ et l'exécutons sur i, jusqu'à ce qu'il n'y ait plus de tâches globales restantes.
En n'introduisant pas de temps d'inactivité dans le planning avant que toutes les tâches globales soient traitées, nous nous assurons que la dernière tâche globale se terminera avant la date limite commune $d^{\mathcal{G}}$.
Notez que les tâches globales sont enlevées de la file d'attente dans un ordre arbitraire.

Politique de rejet pour les tâches globales : Si à tout moment il y a plus de $\frac{1}{\epsilon_r}$ tâches locales dans la file d'attente locale d'une machine exécutant actuellement une tâche globale, nous décidons de rejeter cette tâche globale. Notez qu'un rejet d'une tâche globale ne peut survenir qu'à un moment où une tâche locale est soumise dans le système. De cette façon, nous nous assurons qu'il n'y aura pas plus de $\epsilon_r \cdot n^{\mathcal{L}}$ tâches globales rejetées au total.

3.5 Analyse de *Augmenté* par le dual fitting

Nous analysons cet algorithme *Augmenté* en utilisant la méthode de dual fitting [AGK12]. Le principe de cette méthode est expliqué dans la Sous-Section suivante (3.5.1).

Nous fournissons d'abord une expression du temps total atteint par l'algorithme dans la Section 3.5.2 et donnons une formulation linéaire de notre problème dans la Section 3.5.3, ainsi que le programme dual relâché correspondant. Ensuite, nous proposons une affectation des variables duales en fonction des choix effectués par notre algorithme dans la Section 3.5.4. Dans la section 3.5.5, nous prouvons que cette affectation satisfait les contraintes duales et donnons le rapport de compétitivité de l'algorithme.

3.5.1 Un bref aperçu du Dual Fitting

Le dual fitting est une méthode utilisée pour prouver les rapports d'approximation (ou de compétitivité) d'algorithmes.

Il consiste à trouver une borne inférieure de la valeur optimale, que l'on nommera $B(I)$, pour chaque instance I d'un problème donné, plutôt que de chercher une borne supérieure de la performance de l'algorithme. Cela permet de calculer un rapport de compétitivité en comparant la solution donnée par l'algorithme $Algo(I)$ à cette borne inférieure. En effet, on cherchera alors à trouver une borne supérieure du terme de droite de cette relation $\frac{Algo(I)}{OPT(I)} \leq \frac{Algo(I)}{B(I)}$.

Le processus de dual fitting se déroule en trois étapes :

— formuler le modèle sous la forme d'un programme linéaire (dont la solution optimale est $OPT(I)$,

— écrire son dual (qui est un problème de maximisation pour un problème de minimisation),

— construire une solution réalisable $B(I)$ (qui n'a pas besoin d'être optimale) pour ce dual (on a bien $B(I) \leq OPT(I)$ quelle que soit l'instance I).

La construction de cette solution particulière du dual $B(I)$ est guidée par les décisions prises par l'algorithme de façon à pouvoir obtenir une valeur de $\frac{Algo(I)}{B(I)}$ la plus petite possible.

Dans cette étude, le programme linéaire utilisé (dans la Sous-Section 3.5.3) ne modélise pas exactement le problème tel que définit formellement dans la Section (3.2)

mais modélise plutôt une estimation inférieure de l'objectif d'une version relaxée du problème initial (à un facteur près). Les détails de cette approche seront fournis plus amplement dans le descriptif du programme linéaire. Il est important de noter que cela n'affecte pas la validité de la démarche.

3.5.2 Le flowtime de l'Algorithme *Augmenté*

Tout d'abord, définissons Δ_j comme étant l'augmentation du flowtime total de la solution actuelle de notre algorithme induit par la affectation d'une tâche locale j à la machine i. Cette valeur correspond au flowtime de la tâche j, ainsi qu'à l'augmentation du flowtime de toutes les tâches (locales, affectées à la machine i) qui sont retardées par l'arrivée de j. En suivant nos politiques de planification et de rejet, et en supposant qu'il y avait une tâche k en cours d'exécution à l'instant r_j, la définition détaillée de Δ_j est exprimée comme suit :

$$\Delta_j = \begin{cases} \text{Si k}(\in \mathcal{G}) \text{ est rejetée :} \\ \quad - \dfrac{p_k^{rem}(r_j)}{1+\epsilon_s} \cdot |\mathcal{Q}_i^{\mathcal{L}}(r_j) - 1| + \displaystyle\sum_{\substack{l \in \mathcal{Q}_i^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) \leq p_j}} \frac{p_l^{rem}(r_j)}{1+\epsilon_s} + \sum_{\substack{l \in \mathcal{Q}_i^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) > p_j}} \frac{p_j}{1+\epsilon_s} \\ \text{Si k}(\in \mathcal{G}) \text{ n'est pas rejetée :} \\ \quad \dfrac{p_k^{rem}(r_j)}{1+\epsilon_s} + \displaystyle\sum_{\substack{l \in \mathcal{Q}_i^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) \leq p_j}} \frac{p_l^{rem}(r_j)}{1+\epsilon_s} + \sum_{\substack{l \in \mathcal{Q}_i^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) > p_j}} \frac{p_j}{1+\epsilon_s} \\ \text{Si k}(\in \mathcal{L}) \text{ est préemptée :} \\ \quad \dfrac{p_j}{1+\epsilon_s} + \displaystyle\sum_{\substack{l \in \mathcal{Q}^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) > p_j}} \frac{p_j}{1+\epsilon_s} \\ \text{Si k}(\in \mathcal{L}) \text{ n'est pas préemptée :} \\ \quad \displaystyle\sum_{\substack{l \in \mathcal{Q}_i^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) \leq p_j}} \frac{p_l^{rem}(r_j)}{1+\epsilon_s} + \sum_{\substack{l \in \mathcal{Q}^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) > p_j}} \frac{p_j}{1+\epsilon_s} \end{cases}$$

Notez que, en raison de l'augmentation de vitesse dont bénéficie l'algorithme *Augmenté*, tous les temps d'exécution pour notre algorithme sont divisés par $(1+\epsilon_s)$.

Par définition, le flowtime total des tâches locales réalisé par l'algorithme est égal à la somme de Δ_j pour toutes les tâches locales.

En raison de la politique de rejet, l'impact des tâches globales sur le temps total de flux des tâches locales est limité. Il ne peut y avoir plus de $\frac{1}{\epsilon_r}$ tâches locales dans la file d'attente pendant l'exécution d'une tâche globale, de sorte qu'une tâche globale j contribuera au maximum à $\frac{p_j}{\epsilon_r(1+\epsilon_s)}$ sur le flowtime total produit par l'algorithme. Par conséquent, nous avons ceci :

$$\sum_{j\in\mathcal{L}} F_j = \sum_{j\in\mathcal{L}} \Delta_j \leq \sum_{j\in\mathcal{L}} \left(\sum_{\substack{l\in\mathcal{Q}_i^{\mathcal{L}}(r_j):\\ p_l^{rem}(r_j)\leq p_j}} \frac{p_l^{rem}(r_j)}{1+\epsilon_s} + \sum_{\substack{l\in\mathcal{Q}_i^{\mathcal{L}}(r_j):\\ p_l^{rem}(r_j)> p_j}} \frac{p_j}{1+\epsilon_s} \right) + \sum_{j\in\mathcal{G}} \frac{p_j}{\epsilon_r(1+\epsilon_s)} \tag{3.3}$$

3.5.3 Une Formulation d'un Programme Linéaire qui modélise le comportement de l'Optimal

Nous définissons une variable de décision $x_{ij}(t)$ qui vaut 1 si la tâche $j \in \mathcal{L}\cup\mathcal{G}$ est en cours d'exécution sur la machine $i \in \mathcal{M}$ à l'instant $t \geq 0$, et 0 sinon. Par convention, nous considérerons $x_{ij}(t) = 0$ pour une tâche locale j lorsque $t < r_j$.

Considérons le programme linéaire suivant, avec la constante $\Gamma \geq \frac{1}{2}$:

$$\text{min.} \quad \sum_{i\in\mathcal{M}}\sum_{j\in\mathcal{L}} \int_{r_j}^{\infty} \left(\frac{(t-r_j)}{p_j} + \Gamma \right) x_{ij}(t)dt$$

$$\text{s.t.} \sum_{i\in\mathcal{M}} \int_{r_j}^{\infty} x_{ij}(t)dt \geq p_j \qquad \forall j \in (\mathcal{L}\cup\mathcal{G}) \tag{3.4a}$$

$$\sum_{j\in(\mathcal{L}\cup\mathcal{G})} x_{ij}(t) \leq 1 \qquad \forall i \in \mathcal{M},\ \forall t \geq 0 \tag{3.4b}$$

$$\sum_{i\in\mathcal{M}} \int_{0}^{\infty} \left(\frac{t}{p_j} + \frac{1}{2} \right) x_{ij}(t)dt \leq d^{\mathcal{G}} \qquad \forall j \in \mathcal{G} \tag{3.4c}$$

$$x_{ij}(t) \in \{0;1\} \qquad \forall i \in \mathcal{M},\ \forall j \in (\mathcal{L}\cup\mathcal{G}),\ \forall t \geq 0$$

Dans la contrainte (3.4a), nous vérifions que chaque tâche $j \in \mathcal{L}\cup\mathcal{G}$ est exécutée pendant au moins p_j unités de temps. La contrainte (3.4b) indique qu'une machine ne peut exécuter qu'une tâche à un moment donné $t \geq 0$, et le côté gauche de la contrainte (3.4c) correspond à la date de complétion d'une tâche globale j (d'après

la formule (3.5)). Si on considère que la tâche globale j ne peut pas être préemptée ; et que l'on note S_j l'instant où la tâche j commence,

$$\begin{aligned}\int_0^\infty \left(\frac{t}{p_j}+\frac{1}{2}\right) x_j(t)dt &= \left[\frac{t^2+p_j\cdot t}{2p_j}\right]_{S_j}^{S_j+p_j} = \frac{(S_j+p_j)^2-S_j^2+p_j(S_j+p_j-S_j)}{2p_j} \\ &= \frac{(S_j^2+2S_j\dot{p}_j+p_j^2)-S_j^2+p_j^2}{2p_j} \\ &= S_j+p_j = \text{Completion Time}_j \end{aligned} \tag{3.5}$$

Notez que la quantité $\int_{r_j}^\infty \frac{(t-r_j)}{p_j} x_{ij}(t)dt$ dans la fonction objectif correspond à une formulation bien connu du *flowtime fractionnel* de la tâche j [AGK12]. C'est une borne inférieure pour le flowtime de j. De plus, avec $\Gamma = \frac{1}{2}$, la valeur objectif d'une solution optimale pour le programme linéaire est au plus égale au flowtime d'un planning optimal pour notre problème [AGK12]. Pendant notre analyse, nous utiliserons une valeur plus grande de Γ et la valeur objectif du programme sera au plus égale à $(\Gamma+\frac{1}{2})$ de la valeur optimale du problème.

Vous pourrez trouver des calculs détaillés dans l'annexe (B) utilisés pour saisir l'idée derrière la formulation choisie pour l'optimal. Bien que les références cités dans la description de la fonction objectif suffisent à justifier le fait qu'il s'agisse bien d'une borne inférieure du flowtime, les quelques calculs détaillés de l'annexe permettent de mettre en lumière les limites de cette formulation dans le contexte de notre analyse au vu des écarts entre nos inégalités.

Notez également que la formulation ci-dessus ne modélise pas exactement le problème abordé, car elle n'interdit pas la préemption des tâches globales ou leur migration entre les machines. Au lieu de cela, la formulation est une relaxation (qui donne plus de "pouvoirs" à l'optimal auquel on se compare), mais elle est suffisante pour analyser l'algorithme et prouver son rapport de competitivité via le dual fitting et l'augmentation de ressources.

3.5.4 Une solution particulière du Dual

Après avoir relâché la contrainte d'intégrité sur les variables $x_{ij}(t)$ dans le programme primal, son programme dual peut être exprimé comme suit.

$$\max. \sum_{j \in \mathcal{L} \cup \mathcal{G}} \alpha_j p_j - \sum_{i \in \mathcal{M}} \int_0^\infty \beta_i(t) dt - \sum_{j \in \mathcal{G}} d^{\mathcal{G}} \gamma_j$$

$$\text{s.t. } \alpha_j - \beta_i(t) - \left(\frac{t - r_j}{p_j} + \Gamma \right) \leq 0 \qquad \forall i \in \mathcal{M},\ \forall j \in \mathcal{L},\ \forall t \geq r_j \tag{3.6a}$$

$$\alpha_j - \beta_i(t) - \left(\frac{t}{p_j} + \frac{1}{2} \right) \gamma_j \leq 0 \qquad \forall i \in \mathcal{M},\ \forall j \in \mathcal{G},\ \forall t \geq 0 \tag{3.6b}$$

$$\alpha_j \geq 0 \qquad \forall j \in (\mathcal{L} \cup \mathcal{G})$$

$$\beta_i(t) \geq 0 \qquad \forall t \geq 0,\ \forall i \in \mathcal{M}$$

$$\gamma_j \geq 0 \qquad \forall j \in \mathcal{G}$$

Pour l'analyse par dual fitting, nous définissons les variables du programme dual en fonction des décisions prises par l'algorithme *Augmenté*.

Nous définissons la première variable duale α_j pour chaque tâche locale j comme $\alpha_j = \frac{\lambda_j}{(1+\epsilon_s)p_j} + \Gamma$. Notez que la valeur de α_j est uniquement définie à l'arrivée de j, et ne change pas par la suite.

Pour les autres variables, nous définissons $\alpha_j = 0$ et $\gamma_j = 0$ pour toute tâche globale j ; et $\beta_i(t) = \frac{|\mathcal{Q}_i^{\mathcal{L}}(t)|}{1+\epsilon_s},\ \forall t \geq 0$.

3.5.5 Analyse de compétitivité

Avec les deux lemmes suivants, nous montrons que notre définition des variables duales mène à une solution réalisable du programme dual.

Lemme 3.5.1. *Pour toute machine $i \in \mathcal{M}$, chaque tâche locale $j \in \mathcal{L}$ et pour tout temps $t \geq r_j$, la contrainte duale (3.6a) est respectée, c'est à dire :*

$$\alpha_j - \beta_i(t) - \left(\frac{t - r_j}{p_j} + \Gamma \right) \leq 0$$

Preuve. La première chose à noter est que le terme constant Γ dans α_j est compensé par le Γ de la contrainte.

En multipliant tous les autres termes par $(1+\epsilon_s)p_j$, nous devons prouver :

$$\lambda_j - |\mathcal{Q}_i^{\mathcal{L}}(t)| \cdot p_j - (1+\epsilon_s)(t-r_j) \leq 0$$

Si d'autres tâches locales arrivent après j, la quantité $|\mathcal{Q}_i^{\mathcal{L}}(t)|$ augmentera et la contrainte sera plus facile à satisfaire. C'est pourquoi, nous supposons qu'il n'y a pas d'autre arrivée de tâches après j. Ainsi, $|\mathcal{Q}^{\mathcal{L}}i(t)|$ ne peut que diminuer au fil du temps, lorsqu'une tâche termine son exécution. De plus, en raison de la politique de distribution des tâches locales, nous avons $\lambda_j \leq \lambda ij$.

Supposons qu'une tâche k soit actuellement exécutée à l'heure r_j et se termine à l'heure $t' = r_j + \frac{p_k^{rem}(r_j)}{1+\epsilon_s}$. Pour tout temps $t \in [r_j, t')$, $|\mathcal{Q}_i^{\mathcal{L}}(t)|$ restera constant et il suffit de prouver la contrainte au début de l'intervalle, lorsque $t = r_j$. Nous avons :

$$\lambda_{ij} \leq \sum_{\substack{l \in \mathcal{Q}_i^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) \leq p_j}} p_j \quad + \sum_{\substack{l \in \mathcal{Q}_i^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) > p_j}} p_j \quad = |\mathcal{Q}_i^{\mathcal{L}}(r_j)| \cdot p_j$$

et ainsi, la contrainte est satisfaite pour tout temps t dans cet intervalle.

Il est maintenant nécessaire de définir k' comme étant la première tâche dans $\mathcal{Q}^{\mathcal{L}}i(r_j)$ qui suit k, de sorte que lorsque k finit son exécution à l'heure t', k' débute son exécution dans l'intervalle $\left[t', t' + \frac{pk'^{rem}(r_j)}{1+\epsilon_s}\right)$. Encore une fois, il suffit de vérifier la contrainte au début de l'intervalle, à l'instant $t' = r_j + \frac{p_k^{rem}(r_j)}{1+\epsilon_s}$. Nous avons

$$\begin{aligned}\lambda_{ij} &\leq p_k^{rem}(r_j) + \sum_{\substack{l \in \mathcal{Q}_i^{\mathcal{L}}(r_j) \setminus \{k\}: \\ p_l^{rem}(r_j) \leq p_j}} p_j \quad + \sum_{\substack{l \in \mathcal{Q}_i^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) > p_j}} p_j \\ &= p_k^{rem}(r_j) + |\mathcal{Q}_i^{\mathcal{L}}(t')| \cdot p_j\end{aligned}$$

et

$$p_k^{rem}(r_j) - (1+\epsilon_s)(r_j + \frac{p_k^{rem}(r_j)}{1+\epsilon_s} - r_j) \leq 0.$$

Ainsi, la contrainte est satisfaite pour tout t dans l'intervalle.

Il est possible de faire un raisonnement similaire pour chaque intervalle de temps pendant l'exécution de toute tâche locale dans $\mathcal{Q}_i^{\mathcal{L}}(r_j)$. Notez que, si une tâche globale a été rejetée lors de l'arrivée de la tâche locale j, cela n'affecterait pas le raisonnement ci-dessus. Par conséquent, la contrainte duale (3.6a) est satisfaite pour tout temps $t \geq 0$. □

Lemme 3.5.2. *Pour chaque machine $i \in \mathcal{M}$, pour chaque tâche globale j et pour chaque temps $t \geq 0$, la contrainte duale (3.6b) est satisfaite, c'est à dire :*

$$\alpha_j - \beta_i(t) - (\frac{t}{p_j} + \frac{1}{2})\gamma_j \leq 0$$

Preuve. La valeur la plus petite possible de $\beta_i(t)$ est obtenue lorsque la file d'attente des tâches locales est vide. Ainsi, pour tout t, nous avons $\beta_i(t) \geq 0$. Étant donné que $\alpha_j = 0$ et $\gamma_j = 0$, la contrainte duale (3.6b) est satisfaite. □

Rappelons que F_j désigne le flowtime de la tâche j dans l'ordonnancement produit par l'algorithme. Le lemme suivant montre la relation entre le flowtime des tâches locales obtenu par l'algorithme et la valeur objectif du programme dual.

Lemme 3.5.3. *Avec nos définitions de α_j, $\beta_i(t)$ and γ_j, la valeur de la fonction objectif du dual vérifie :*

$$\sum_{j \in \mathcal{L} \cup \mathcal{G}} \alpha_j p_j - \sum_{i \in \mathcal{M}} \int_0^\infty \beta_i(t)dt - \sum_{j \in \mathcal{G}} d^{\mathcal{G}} \gamma_j \geq \frac{\epsilon_s}{1+\epsilon_s} \sum_{j \in \mathcal{L}} F_j$$

Preuve. Avec nos définitions de α_j (rappelons $\alpha_j = 0, \forall j \in \mathcal{G}$), nous avons :

$$\begin{aligned}
\sum_{j \in \mathcal{L} \cup \mathcal{G}} \alpha_j p_j = \sum_{j \in \mathcal{L}} \Bigg(& \sum_{\substack{l \in \mathcal{Q}_{i^*}^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) \leq p_j}} \frac{p_l^{rem}(r_j)}{1+\epsilon_s} + \sum_{\substack{l \in \mathcal{Q}_{i^*}^{\mathcal{L}}(r_j): \\ p_l^{rem}(r_j) > p_j}} \frac{p_j}{1+\epsilon_s} + \Gamma \cdot p_j \Bigg) \\
& + \frac{1}{1+\epsilon_s} \left(\sum_{j \in \mathcal{G}} \frac{p_j}{\epsilon_r} - \sum_{j \in \mathcal{G}} \frac{p_j}{\epsilon_r} \right) \\
& \geq \sum_{j \in \mathcal{L}} F_j + \Gamma \cdot \sum_{j \in \mathcal{L}} p_j - \sum_{j \in \mathcal{G}} \frac{p_j}{\epsilon_r(1+\epsilon_s)},
\end{aligned}$$

pour laquelle l'inégalité est respectée en raison de l'équation (3.3).

Avec notre définition de $\beta_i(t)$, nous avons :

$$\sum_{i \in \mathcal{M}} \int_0^\infty \beta_i(t)dt = \sum_{i \in \mathcal{M}} \int_0^\infty \frac{|\mathcal{Q}_i^{\mathcal{L}}(t)|}{1+\epsilon_s} dt = \frac{1}{1+\epsilon_s} \sum_{j \in \mathcal{L}} F_j$$

Rappelons la notation $\mathcal{W} = \frac{\sum_{j\in\mathcal{G}} p_j}{\sum_{j\in\mathcal{L}} pj}$, qui est le rapport entre la charge de travail totale des tâches globales et la charge de travail totale des tâches locales. En choisissant

$$\Gamma = \max\left\{\frac{\mathcal{W}}{\epsilon_r(1+\epsilon_s)}, \frac{1}{2}\right\},$$

et sachant que $\gamma_j = 0, \forall j \in \mathcal{G}$, la valeur de la fonction objectif du dual peut être exprimée de la façon suivante :

$$\begin{aligned}
&\sum_{j\in\mathcal{L}\cup\mathcal{G}} \alpha_j p_j - \sum_{i\in\mathcal{M}} \int_0^\infty \beta_i(t)dt - \sum_{j\in\mathcal{G}} d^{\mathcal{G}}\gamma_j \\
&\quad\geq \sum_{j\in\mathcal{L}} F_j + \Gamma\cdot\sum_{j\in\mathcal{L}} p_j - \sum_{j\in\mathcal{G}} \frac{p_j}{\epsilon_r(1+\epsilon_s)} - \frac{1}{1+\epsilon_s}\sum_{j\in\mathcal{L}} F_j \\
&\quad= \frac{\epsilon_s}{1+\epsilon_s}\sum_{j\in\mathcal{L}} F_j + \Gamma\cdot\sum_{j\in\mathcal{L}} p_j - \sum_{j\in\mathcal{G}} \frac{p_j}{\epsilon_r(1+\epsilon_s)} \\
&\quad\geq \frac{\epsilon_s}{1+\epsilon_s}\sum_{j\in\mathcal{L}} F_j
\end{aligned}$$

□

Nous avons maintenant suffisamment d'éléments pour conclure sur le rapport de compétitivité de notre algorithme.

Théorème 3.5.4. *Pour tout $0 < \epsilon_r < 1$ et $\epsilon_s > 0$, l'algorithme $\mathcal{A}$ qui propose une $(1+\epsilon_s)$-augmentation de vitesse a un rapport de compétitivité de $\max\left\{\frac{\mathcal{W}}{\epsilon_s\epsilon_r} + \frac{1+\epsilon_s}{2\epsilon_s}, \frac{1+\epsilon_s}{\epsilon_s}\right\}$ et rejette au moins $(\epsilon_r \cdot n^{\mathcal{L}})$ tâches globales.*

Preuve. D'après les trois lemmes précédents, nous savons que les variables duales telles que nous les avons définies forment une solution réalisable du programme dual et que la valeur objectif de ce programme est une borne supérieure du flowtime de l'algorithme multiplié par un facteur constant. Selon la politique de rejet, une tâche globale est rejetée la première fois qu'il y a plus de $\frac{1}{\epsilon_r}$ tâches locales en attente dans la file d'attente. Ainsi, pas plus de $\epsilon_r \cdot n^{\mathcal{L}}$ tâches globales sont rejetées. Par définition, l'algorithme utilise une machine à une vitesse $(1+\epsilon_s)$ fois plus rapide que celle d'une solution optimale.

Il reste à exprimer la relation entre le flowtime de notre algorithme et celui d'une solution optimale. Comme $\Gamma \geq \frac{1}{2}$, la valeur de la fonction objectif du programme

primal est au plus $\Gamma + \frac{1}{2}$ fois le flowtime d'une solution optimale, notée OPT. Enfin, en utilisant le théorème de dualité, nous pouvons écrire la relation :

$$\frac{\epsilon_s}{1+\epsilon_s} \cdot \sum_{j \in \mathcal{L}} F_j \leq (\Gamma + \frac{1}{2}) \cdot OPT$$
$$\frac{\sum_{j \in \mathcal{L}} F_j}{OPT} \leq \max \left\{ \frac{\mathcal{W}}{\epsilon_s \epsilon_r} + \frac{1+\epsilon_s}{2\epsilon_s}, \frac{1+\epsilon_s}{\epsilon_s} \right\}$$

et le théorème est vérifié. □

3.6 Conclusion et perspectives

Dans ce chapitre, nous avons introduit un nouveau problème de planification avec deux agents visant chacun un objectif différent. À notre connaissance, c'était la première fois qu'un cadre mixte online / offline est étudié pour ce problème. Nous avons fourni une borne inférieure sur le rapport de compétitivité de tout algorithme pour résoudre le problème et proposé un algorithme avec une preuve basée sur le dual fitting dans le cadre de l'augmentation de ressources (vitesse et rejet).

L'objectif des tâches globales est pris comme une contrainte, tandis que sa valeur est spécifiée par la politique de planification des tâches globales. En utilisant le List Scheduling dans notre cas, la date limite commune a été prise comme la borne classique de Graham plus la charge de travail totale des tâches locales. Il est possible de raffiner la valeur de la date limite changeant de la politique de planification des tâches globales dans l'algorithme $\mathcal{A}$: en utilisant n'importe quel algorithme $\mathcal{X}$ pour $P||C_{max}$ avec un rapport d'approximation connu ρ pour construire une affectation fixe des tâches globales aux machines avant la soumission de toute tâche locale, et il est garanti que la valeur de la deadline globale $d^{\mathcal{G}} = C^{\mathcal{X}}_{max} + \sum_{j \in \mathcal{L}} p_j$ sera respectée. En faisant cela, la vision "dynamique" de l'allocation des tâches globales est perdue, mais cela nous donne une vision "bi-objectif" de notre problème : la minimisation du flowtime total des tâches locales ainsi que de la deadline globale.

Les autres perspectives que nous mettions en avant dans [Fag+22] (étendre le modèle au cas pondéré, à des machines liées/non liées, ou encore remplacer la fonction objectif et s'intéresser à la minimisation du *Stretch* ($\sum F_j/p_j$)) ne semblent au final pas si prometteuses, car utiliser une méthode similaire à celle présentée dans ce chapitre ajoute de nombreuses complications sur la borne du nombre de tâches globales rejetées. En revanche, réduire les écarts entre la solution modélisée

par le LP (3.5.3) et le meilleur résultat obtenu par l'optimal permettrait d'améliorer l'analyse de l'algorithme. Une piste qui pourrait se montrer fructueuse (dans une moindre mesure) serait la prise en compte des écarts notés a_j dans (B) dans le choix des variables duales α_j.

Nous pensons également qu'il et possible de supprimer l'augmentation de vitesse du modèle, qui n'est utilisée que dans l'analyse de l'algorithme, comme cela a été réussi dans une autre étude sur un problème à agent unique avec des tâches online [Luc+18a].

4 Plateforme Hétérogène : CPUs / GPUs

Ce chapitre est basé sur des travaux en collaboration avec Imed Kacem, Giorgio Lucarelli et Bertrand Simon, publiés dans Latin [Fag+20].

De nos jours, de plus en plus de plateformes de calcul haute performance utilisent des processeurs à usage spécifique en conjonction avec des unités centrales de traitement classiques (CPUs) pour accélérer certaines opérations et améliorer leurs performances. Un exemple typique est l'utilisation de GPUs modernes qui peuvent accélérer les opérations de vecteur et de matrice.

En raison de l'hétérogénéité introduite par ce type d'accélérateurs, le problème de planification sur ces plateformes hybrides devient plus difficile. Plusieurs résultats expérimentaux ainsi que des limites inférieures théoriques [Ama+17] montrent que la décision d'allocation d'une tâche à un type de processeur est cruciale pour les performances du système. En particulier, les politiques gloutonnes classiques, telles que l'ordonnancement de liste de Graham [Gra69], qui montrent de bonnes performances sur une plateforme composée de ressources identiques, n'offrent en général pas de bonne performances dans de cas de plateformes hybrides. Pour cette raison, tous les algorithmes connus pour les plateformes hybrides [Ama+17 ; Can+18 ; CYZ14 ; KMT15] choisissent le type de ressource pour chaque tâche avant de décider de son ordonnancement dans l'horizon temporel.

4.1 Introduction

Dans ce chapitre, nous nous concentrons sur le problème de planification d'une application sur une plateforme hybride composée de m CPUs identiques et k GPUs identiques, tel que présenté dans la sous-section 1.3.3 de l'introduction. Une application est décrite comme un ensemble de n tâches mono-processeur V qui peuvent être liées par des relations de précédence, décrites par un graphe dirigé acyclique $G = (V, E)$.

Cela signifie qu'une tâche ne peut être exécutée que lorsque toutes ses tâches précédentes sont terminées. Le temps de traitement de la tâche j sur un CPU (resp. sur un GPU) est noté $\overline{p_j}$ (resp. $\underline{p_j}$), et nous n'assumons aucune relation entre $\overline{p_j}$ et $\underline{p_j}$. Cela est justifié par les systèmes réels, où les tâches effectuant, par exemple, des opérations de matrice peuvent être exécutées beaucoup plus efficacement sur un GPU, tandis que l'exécution de tâches qui ont souvent besoin de communiquer avec le système de fichiers est plus rapide sur un CPU. Par conséquent, nous pouvons sans perte de généralité supposer que $m \geq k$.

Nous nous intéressons à la conception d'algorithmes en temps polynomial avec de bonnes garanties de performance dans le pire des cas. Comme mesure de performance, nous utilisons le rapport d'approximation (bien connu) qui compare la solution d'un algorithme et la solution optimale, par rapport à une fonction objectif. Dans ce papier, nous étudions l'objectif *makespan*, c'est-à-dire que nous visons à minimiser la date de fin de la dernière tâche exécutée. En étendant la notation de Graham, nous noterons ce problème $(Pm, Pk) \mid prec \mid C_{\max}$.

Pour ce problème, un algorithme avec un rapport d'approximation de 6, nommé HLP (Programme linéaire hétérogène), a été proposé par Kedad-Sidhoum *et al.* [KMT15]. Cet algorithme a deux phases. Dans la première phase, une "bonne" allocation pour chaque tâche est décidée : soit sur le côté CPU, soit sur le côté GPU. Cette décision est basée sur un programme linéaire en nombres entiers qui utilise une variable de décision binaire x_j pour chaque tâche j : x_j est égal à 1 si j est affecté au côté CPU et à 0 sinon. Ce programme linéaire entier ne modélise pas tout le problème de planification mais seulement la décision de l'allocation, en essayant de rééquilibrer la charge moyenne sur les CPU et GPU ainsi que la longueur du chemin critique. La relaxation fractionnelle de ce programme est résolue et l'allocation de chaque tâche j est déterminée par une règle d'arrondi simple : elle est affectée aux GPU si $x_j < 1/2$, et aux CPU sinon. Dans la seconde phase, un algorithme de liste est utilisé pour planifier les tâches en respectant les contraintes de précédence et l'allocation définie dans la première phase.

4.1.1 Contribution

Dans [KMT15], les auteurs prouvent que la valeur de $1/2$ choisie est la meilleure par rapport au programme linéaire utilisé dans la première phase pour procéder à un arrondi. En quelque sorte, ils prouvent que l'écart d'intégralité du relâchement du programme linéaire est 2. De plus, compte tenu de cette règle de arrondi simple basée sur $1/2$, Amaris *et al.* [Ama+17] présentent un exemple (*tight*) d'HLP qui

atteint asymptotiquement un rapport d'approximation de 6, même si un autre algorithme de planification est utilisé dans la seconde phase. Malgré ces deux résultats négatifs précédents, nous montrons que HLP peut obtenir un meilleur rapport d'approximation en utilisant une procédure d'arrondi différente. En effet, même si nous utilisons un arrondi qui n'est pas le meilleur possible par rapport au problème d'affectation résolu dans la première phase, cet arrondi nous permet d'obtenir des garanties plus fortes sur la phase de planification et donc d'améliorer le rapport d'approximation. La principale différence avec HLP est que nous affectons la tâche j au type de processeur le plus rapide si x_j est proche de $1/2$ dans la solution du fractionnaire. Nous obtenons alors un rapport d'approximation inférieur à $3 + 2\sqrt{2}$ et qui tend vers 3 lorsque m/k est proche de 1.

Le meilleur rapport d'inapproximation connu est le même que pour les machines identiques, c'est-à-dire $4/3$[LR78], mais peut être amélioré à 2 en supposant une variante de la conjecture des jeux uniques (*unique game conjecture* [Sve11]). La seconde contribution de ce chapitre est d'améliorer cette borne inférieure du rapport d'approximation (conditionnel) à 3 pour toute valeur de m/k en supposant une conjecture plus forte introduite par Bazzi et Norouzi-Fard [BN15]. Ce rapport de bornes conditionnel est donc serré lorsque $m = k$.

Dans la section 4.1.2, nous donnons un aperçu de la littérature en positionnant ce problème par rapport à des problèmes similaires et en présentant plusieurs résultats d'approximation connus. Dans la section 4.2, nous présentons notre algorithme adapté pour le problème de planification sur des plateformes hybrides ainsi que son analyse qui aboutit à un rapport d'approximation de 5,83. Dans la section 4.4, nous prouvons une borne inférieure conditionnelle de 3 sur le rapport d'approximation. Enfin, nous concluons dans la section 4.5.

4.1.2 Travaux connexes

Le problème de d'ordonnancement sur des plateformes hybrides composées de deux ensembles de processeurs identiques est une généralisation du problème classique de planification sur des processeurs parallèles identiques, noté $P \mid prec \mid C_{\max}$. D'un autre côté, notre problème est un cas particulier du problème de planification sur des processeurs non apparentés (noté $R \mid prec \mid C_{\max}$), où chaque tâche a un temps de traitement différent sur chaque processeur. De plus, dans le cas de la planification sur des processeurs à vitesses uniformes (noté $Q \mid prec \mid C_{\max}$), chaque processeur a sa vitesse spécifique et le temps de traitement de chaque tâche dépend de la vitesse du processeur assigné. Ce problème est plus général que $P \mid prec \mid C_{\max}$ dans le

sens que le temps de traitement d'une tâche est différent sur chaque processeur. Cependant, dans ce dernier problème, toutes les tâches sont accélérées ou décélérées par le même facteur lorsqu'elles sont exécutées sur un processeur spécifique, tandis que dans notre cas, deux tâches n'ont pas nécessairement le même comportement (accélération ou décélération) si elles sont planifiées sur un CPU ou un GPU.

Pour le problème $P \mid prec \mid C_{\max}$, l'algorithme glouton de liste proposé par Graham [Gra69] a un rapport d'approximation de $(2 - \frac{1}{m})$, où m est le nombre de processeurs. Svensson [Sve11] a prouvé que c'est le meilleur résultat d'approximation possible, en supposant $P \neq NP$ et une variante de la conjecture de jeux uniques introduite par Bansal et Khot [BK09]. Notez que ce résultat négatif est également valable pour notre problème plus général. Pour $Q \mid prec \mid C_{\max}$, une série d'algorithmes avec des rapports d'approximation logarithmiques sont connus (voir par exemple [CB98 ; CS97]), tandis que Li [Li17] a récemment proposé un algorithme $O(\log(m)/\log(\log(m)))$, qui propose le meilleur rapport d'approximation connu actuellement. Côté négatif, Bazzi et Norouzi-Fard [BN15] montrent qu'il n'est pas possible d'avoir un rapport d'approximation constant en supposant la dureté NP de certains problèmes sur des graphes k-parti. Aucun résultat n'est actuellement connu pour $R \mid prec \mid C_{\max}$. Cependant, il y a quelques algorithmes d'approximation pour des classes spéciales de graphes de précédence (voir par exemple [Kum+05]).

Pour le problème $(Pm, Pk) \mid prec \mid C_{\max}$, pour les plateformes hybrides, Kedad-Sidhoum *et al.* [KMT15] a présenté un algorithme d'approximation de rapport 6 en séparant les phases d'allocation et de planification. Amaris *et al.* [Ama+17] a proposé de petites améliorations pour les deux phases, sans améliorer le rapport d'approximation. Cependant, ils montrent qu'en utilisant la procédure d'arrondi proposé dans [KMT15], aucune politique de planification ne peut entraîner un rapport d'approximation strictement inférieur à 6. Dans l'absence de contraintes de précédence, un schéma d'approximation en temps polynomial a été proposé par Bleuse *et al.* [Ble+15].

Le problème de planification sur les plateformes hybrides a également été étudié dans le cas online. Si les tâches ne sont pas soumises à des relations de précédence, un algorithme compétitif de 3,85 a été proposé dans [CYZ14], et les auteurs montrent également qu'aucun algorithme en ligne ne peut avoir un rapport de compétitivité strictement inférieur à 2. En présence de contraintes de précédence, Amaris *et al.* [Ama+17] considèrent que les tâches arrivent dans un ordre online respectant les relations de précédence et ils donnent un algorithme $(4\sqrt{m/k})$-compétitif. Ce résultat a été amélioré par Canon *et al.* [Can+18] qui fournissent un algorithme $(2\sqrt{m/k} + 1)$-compétitif, et ils montrent qu'aucun algorithme online ne peut avoir

un rapport de compétitivité inférieur à $\sqrt{m/k}$. De récents travaux de Perotin *et al.* [PSR21] étudient entre autres une généralisation de ce problème en considérant des tâches moldables.

Définition et notation du problème

Étant donnée une plate-forme hybride composée de m CPU identiques et de k GPU identiques sur lesquels nous voulons ordonnancer une application décrite par un graphe acyclique orienté $G = (V, E)$. Les sommets de ce graphe correspondent à des tâches séquentielles, tandis que les arêtes indiquent les relations de priorité entre les tâches. Soit $\mathcal{T}$ l'ensemble des tâches et $n = |\mathcal{T}|$ leur nombre.

Pour toute paire de tâches $j, j' \in \mathcal{T}$, s'il existe un arc (j, j') entre j et j', nous disons que j précède j' et nous le notons $j \prec j'$. Dans ce cas, l'exécution de j' ne peut pas commencer avant l'achèvement de j. Soit $\mathcal{P}$ l'ensemble des chemins orientés du graphe G.

Le temps de traitement de chaque tâche $j \in \mathcal{T}$ dépend du type de machine sur laquelle elle sera exécutée : soit $\overline{p_j}$ le temps de traitement de j si la tâche est exécutée sur n'importe quel CPU et $\underline{p_j}$ son temps de traitement si elle l'est sur un GPU. L'objectif est de minimiser le temps d'achèvement de la dernière tâche, c'est-à-dire le *makespan* de l'ordonnancement.

4.2 L'algorithme HLP-b

Comme expliqué dans l'introduction, l'algorithme HLP-b est composé de deux phases : la phase d'allocation et la phase de planification. La phase d'allocation est basée sur un programme linéaire entier tandis que la phase de planification et basée sur un algorithme de List.

Amaris *et al.* [Ama+17] a montré qu'en utilisant la procédure d'arrondi proposée dans [KMT15] qui permet d'obtenir l'allocation optimale, aucune politique de planification ne peut entraîner un rapport d'approximation strictement inférieur à 6. En revanche, si on choisit de détériorer cette phrase d'allocation, on montre qu'il est possible de casser cette borne d'inapproximabilité de 6.

4.2.1 La phase d'allocation

Pour chaque tâche $j \in V$, nous définissons la variable de décision x_j qui est égale à 1 si la tâche j est affectée au côté CPU et 0 sinon. De plus, définissons la variable C_j correspondant au temps d'achèvement de la tâche j. Enfin, la variable C_{max} représente le makespan, c'est à dire le temps d'achèvement maximal parmi toutes les tâches. Pour plus de simplicité, nous ajoutons dans G une tâche fictive 0 avec $\overline{p_0} = \underline{p_0} = 0$ qui précède toutes les autres tâches. Considérons le programme linéaire en nombres entiers similaire à Kedad-Sidhoum *et al.* [KMT15].

$$\text{Minimiser } C_{\max}$$

$$\frac{1}{m}\sum_{j\in V}\overline{p_j}x_j \leq C_{max} \tag{4.1}$$

$$\frac{1}{k}\sum_{j\in V}\underline{p_j}(1-x_j) \leq C_{max} \tag{4.2}$$

$$C_i + \overline{p_j}x_j + \underline{p_j}(1-x_j) \leq C_j \qquad \forall (i,j) \in E \tag{4.3}$$

$$0 \leq C_j \leq C_{max} \qquad \forall j \in V \tag{4.4}$$

$$x_j \in \{0,1\} \qquad \forall j \in V \tag{4.5}$$

Les contraintes (4.1) et (4.2) impliquent que le makespan de tout ordonnancement ne peut être inférieur à la charge moyenne sur les côtés CPU et GPU, respectivement.

Les contraintes (4.3) et (4.4) constituent le *chemin critique* du graphe de précédence, c'est-à-dire le chemin de G ayant le temps total d'achèvement le plus long. Dans

n'importe quel ordonnancement, la longueur du chemin critique est une borne inférieure du makespan. Notez que le chemin critique de l'instance d'entrée ne peut être défini avant la décision d'allocation pour toutes les tâches car le temps de traitement exact d'une tâche dépend de cette allocation.

La contrainte (4.5) est la contrainte d'intégrité pour la variable de décision x_j. Dans ce qui suit, nous relâchons la contrainte d'intégrité et la remplaçons par $x_j \in [0, 1]$ pour chaque tâche j dans V, afin d'obtenir un programme linéaire que nous pouvons résoudre en temps polynomial.

Le programme linéaire entier ci-dessus n'est pas complètement équivalent à notre problème de planification, mais la valeur objectif de sa solution optimale est une borne inférieure de tout ordonnancement optimal.

La procédure d'arrondi de HLP-b est basée sur un paramètre $b \geq 2$. Nous montrerons dans la sous-section 4.3.2 que le meilleur choix est $b = 1 + \sqrt{\frac{2+k/m-1/m}{1-k/m}}$.

Soit x_j^R la valeur de la variable de décision pour j dans la solution optimale du programme linéaire précédent relaxé. Nous définissons x_j^A comme étant la valeur de la variable qui décide l'affectation de la tâche j pour notre algorithme, c'est-à-dire la valeur de la variable de décision obtenue par la procédure d'arrondi. La phase d'allocation de notre algorithme arrondit la solution relaxée optimale x_j^R pour obtenir une solution réalisable x_j^A de la manière suivante :

— si $x_j^R \geq 1 - \frac{1}{b}$, alors $x_j^A = 1$;
— si $x_j^R \leq \frac{1}{b}$, alors $x_j^A = 0$;
— si $\frac{1}{b} < x_j^R < 1 - \frac{1}{b}$ et $\overline{p_j} \geq \underline{p_j}$, alors $x_j^A = 0$;
— si $\frac{1}{b} < x_j^R < 1 - \frac{1}{b}$ et $\overline{p_j} < \underline{p_j}$, alors $x_j^A = 1$.

Intuitivement, si la solution du programme linéaire est proche d'un entier ($x_j \leq \frac{1}{b}$ ou $x_j \geq 1 - \frac{1}{b}$), nous suivons sa proposition, sinon nous choisissons le type de processeur ayant le temps de traitement le plus faible : si $\overline{p_j} < \underline{p_j}$: la tâche est affectée à un CPU (*c'est-à-dire* $x_j^A = 1$), et à un GPU sinon.

4.2.2 La phase de planification

Étant donnée l'attribution obtenue par la procédure précédente, HLP-b passe à la phase de planification. Un algorithme classique de liste est appliqué en respectant l'attribution x_j^A et les contraintes de précédence : les tâches sont attribuées au processeur disponible le plus tôt ; processeur du type correspondant à l'affectation, dans un ordre topologique.

4.3 Analyse de l'algorithme

Dans cette section, nous présentons une analyse de l'algorithme HLP-b pour le problème $(Pm, Pk) \mid prec \mid C_{\max}$. Nous rappelons ici exhaustivement les définitions et les étapes pour la compréhension complète de notre analyse, bien que certains ingrédients nécessaires aient déjà été présentés dans [KMT15] et [Mon14].
Par soucis de cohérence avec la littérature, certaines définitions et noms de variables (comme T_{1C}, T_2, W_{CPU}...) seront directement tirés des deux documents cités précédemment (de la même manière que le choix de cette notation de Graham $(Pm, Pk) \mid prec \mid C_{\max}$ pour notre problème).

4.3.1 Une borne sur la valeur du makespan

Cette sous-section présente tout d'abord une amélioration de la borne du *makespan* (de l'ordre de $\frac{1}{m}$) produit par notre algorithme en suivant les mêmes arguments et les mêmes notations que présentés dans [Ama+17 ; KMT15]. On pourra retrouver dans la littérature une preuve présentant des arguments et un point de vue différents qui met également en évidence cette amélioration (en annexe de [Can+20]).

Cette démonstration et la borne qui en découle (dans 4.3.1) sont valides quel que soit l'affectation des tâches choisie ; tant que la phase de planification respecte un algorithme de liste.

Considérons un ordonnancement, nous allons définir les temps suivants :

- T_2 : l'ensemble d'instants où au moins un Cpu **et** un Gpu sont OFF
- T_{1C} : l'ensemble d'instants où aucun CPU n'est OFF
- T_{1G} : l'ensemble d'instants où aucun GPU n'est OFF
- T_1 : $T_{1C} \cup T_{1G}$: l'ensemble d'instants où soit tous les CPU, soit tous les GPU, exécutent simultanément une tâche.

Si on note W_{CPU} (resp. W_{GPU}) la somme des temps d'exécution des tâches affectées aux CPU (resp. aux GPU), on peut écrire directement une première borne au vu des définitions de T_1 et T_2 :

$$C_{max} = \|T_1\| + \|T_2\| \leq \|T_{1C}\| + \|T_{1G}\| + \|T_2\| = \frac{W_{CPU}}{m} + \frac{W_{GPU}}{k} + \|T_2\| \quad (4.6)$$

On note CC la valeur du Chemin Critique de l'ordonnancement définit par les relations de précédence du Graphe G tel que : $CC \leq max_{P \in \text{ Chemin de } G} \{ \text{P} \}$.
En appliquant le raisonnement de [Ama+17 ; KMT15], on a : $|T_2| \leq CC \leq C_{max}$.

Idée de la preuve

Si on s'intéresse à un intervalle de temps qui se trouve dans T_2. Au moins un CPU et un GPU sont inactifs. Toutes les tâches commençant leur exécution après cette inactivité ne peuvent donc pas être ordonnancées pendant celle ci ni sur CPU, ni sur GPU. Cela ne peut être qu'uniquement du aux relations de précédence qui contraignent ces tâches à devoir attendre, indépendamment de leur affectation.

On peut raisonner sur l'ordonnancement final et dire que si la dernière tâche à finir n'a pas pu être démarrée plus tôt peu importe son affectation, c'est qu'elle doit avoir une contrainte de précédence (directe ou indirecte) avec au moins une tâche s'exécutant pendant la dernière période d'inactivité de T_2. En appliquant récursivement ce raisonnement à l'instant précédent de T_2, on obtient que $\|T_2\|$ est une borne inférieure au plus long chemin critique du planning.

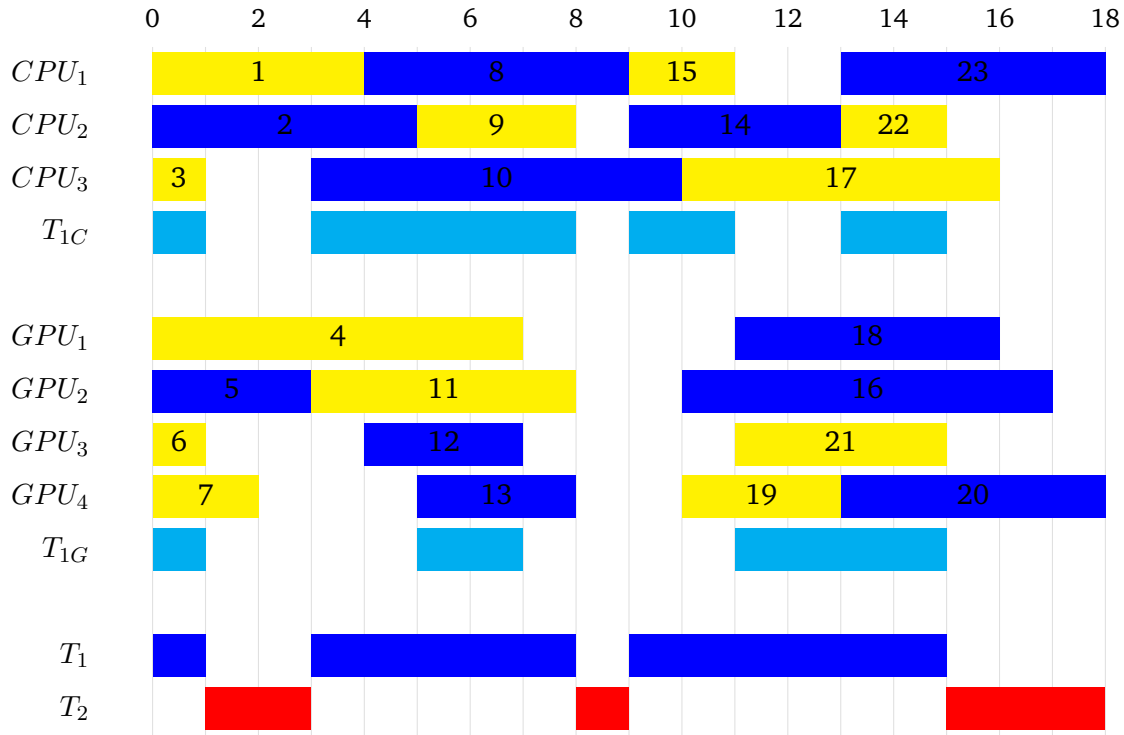


Figure 4.1. : Un exemple d'ordonnancement pour une affectation donnée d'une application composée de tâches numérotées de 1 à 23. La phase de planification est réalisée par un algorithme de List sur une plateforme comportant 3 CPUs et 4 GPUs identiques. Les "trous" dans l'ordonnancement révèlent certaines relations de précédence entre les tâches.
(Les couleurs attribuées aux tâches sont uniquement utilisées à des fins visuelles pour distinguer les différentes tâches.)

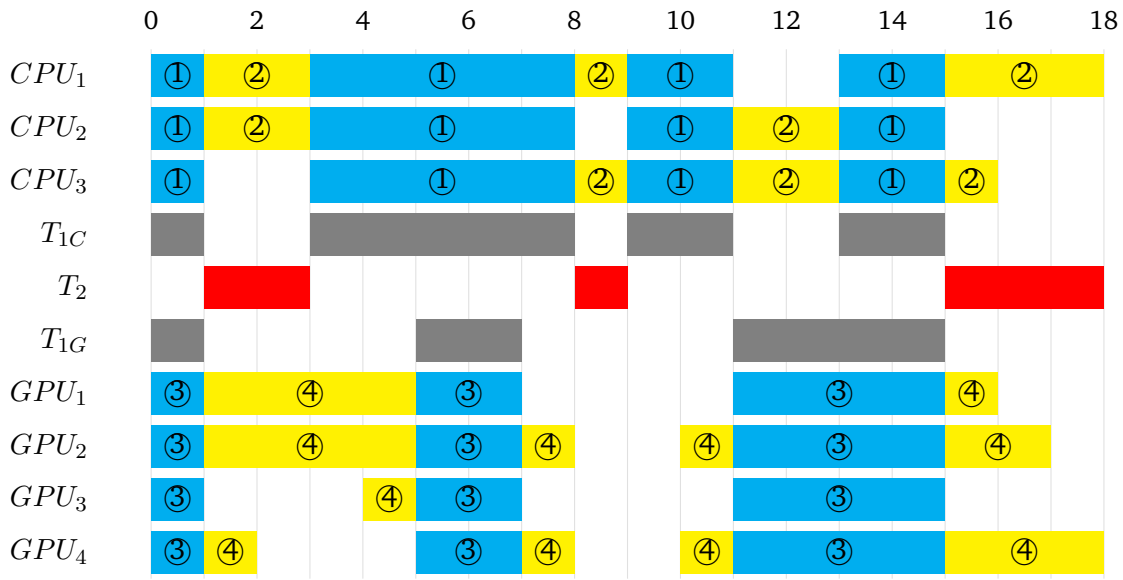


Figure 4.2. : L'ordonnancement présenté dans la figure 4.1 pour lequel l'exécution des tâches a été morcelée, selon leur couverture par T_{1C} ou T_{1G}, pour illustrer et donner une idée plus intuitive des notations définies dans cette Sous-Section.

Pour raffiner la borne précédente, nous allons devoir regarder plus en détail les temps d'exécution et définir les notations suivantes, imagées dans la figure 4.2 :

— $\sum ①$ la somme des temps d'exécution des morceaux de tâches exécutés sur les CPUs pendant les instants de T_{1C},

— $\sum ②$ la somme des temps d'exécution des morceaux de tâches exécutés sur les CPUs en dehors des instants de T_{1C},

— $\sum ③$ la somme des temps d'exécution des morceaux de tâches exécutés sur les GPUs pendant les instants de T_{1G},

— $\sum ④$ la somme des temps d'exécution des morceaux de tâches exécutés sur les GPUs en dehors des instants de T_{1G}.

$$\text{On a alors : } W_{CPU} = \sum ① + \sum ② = m\,\|T_{1C}\| + \sum ②$$
$$W_{GPU} = \sum ③ + \sum ④ = k\,\|T_{1G}\| + \sum ④$$

Comme les tâches sont planifiées suivant une politique de List, à chaque instant (en particulier à chaque instant de T_2), il y a au moins toujours une tâche en cours d'exécution soit sur un CPU, soit sur un GPU.
D'où $T_2 \leq \sum ② + \sum ④$.

On peut donc formuler directement l'inéquation suivante et en déduire une borne améliorée, de la même façon qu'avec la formule 4.6.

$$
\begin{aligned}
\|T_{1C}\| + \|T_{1G}\| &= \left(\frac{W_{CPU}}{m} - \frac{\sum \textcircled{2}}{m}\right) + \left(\frac{W_{GPU}}{k} - \frac{\sum \textcircled{4}}{k}\right) \\
&\leq \frac{W_{CPU}}{m} + \frac{W_{GPU}}{k} - \frac{\sum \textcircled{2} + \sum \textcircled{4}}{max(m,k)} \\
&\leq \frac{W_{CPU}}{m} + \frac{W_{GPU}}{k} - \frac{\|T_2\|}{m} \\
\text{d'où } C_{max} &\leq \|T_{1C}\| + \|T_{1G}\| + \|T_2\| \\
&\leq \frac{W_{CPU}}{m} + \frac{W_{GPU}}{k} + (1 - \frac{1}{m}) \|T_2\| \\
&\leq \frac{W_{CPU}}{m} + \frac{W_{GPU}}{k} + (1 - \frac{1}{m}) CC
\end{aligned}
$$

4.3.2 Analyse de la procédure d'arrondi

Nous commençons l'analyse d'HLP-b avec quelques lemmes basés sur la procédure d'arrondi. Rappelons les deux notations suivantes : pour une tâche j, x_j^R est la valeur de la variable de décision d'affectation dans la solution optimale du programme linéaire précédent relaxé, et x_j^A la valeur qui représente l'affectation de cette tâche pour notre algorithme.

Lemme 4.3.1. *Pour chaque tâche $j \in V$:* $(1 - x_j^A)\underline{p_j} \leq b \cdot (1 - x_j^R)\underline{p_j}$.

Preuve. Considérons n'importe quelle tâche $j \in V$. Si j est assignée aux CPU par l'algorithme, alors $x_j^A = 1$ et le lemme est vérifié car $x_j^R \leq 1$. Maintenant, si j est assignée aux GPU, alors $x_j^A = 0$. Par conséquent, $x_j^R \leq (1 - \frac{1}{b})$. Ainsi, nous pouvons conclure que $b \cdot (1 - x_j^R)\underline{p_j} \geq \underline{p_j} = (1 - x_j^A)\underline{p_j}$. □

Lemme 4.3.2. *Pour chaque tâche $j \in V$:*

$$x_j^A\overline{p_j} + (1 - x_j^A)\underline{p_j} \leq \frac{b}{b-1}(x_j^R\overline{p_j} + (1 - x_j^R)\underline{p_j}).$$

Preuve. Considérons n'importe quelle tâche $j \in V$. Nous nous trouvons dans un des trois cas suivants.

— Si $x_j^R \leq \frac{1}{b}$, alors $x_j^A = 0$ et nous avons :

$$(1 - x_j^R)\underline{p_j} \geq (1 - \frac{1}{b})(1 - x_j^A)\underline{p_j}$$
$$(1 - x_j^R)\underline{p_j} + x_j^R\overline{p_j} \geq (1 - \frac{1}{b})\left((1 - x_j^A)\underline{p_j} + x_j^A\overline{p_j}\right).$$

— Si $x_j^R \geq (1 - \frac{1}{b})$, alors $x_j^A = 1$ et nous avons :

$$x_j^R\overline{p_j} \geq (1 - \frac{1}{b})x_j^A\overline{p_j}$$
$$(1 - x_j^R)\underline{p_j} + x_j^R\overline{p_j} \geq (1 - \frac{1}{b})\left((1 - x_j^A)\underline{p_j} + x_j^A\overline{p_j}\right).$$

— Si $\frac{1}{b} < x_j^R < (1 - \frac{1}{b})$, alors nous avons :

$$x_j^R\overline{p_j} + (1 - x_j^R)\underline{p_j} \geq \min(\overline{p_j}, \underline{p_j}) = x_j^A\overline{p_j} + (1 - x_j^A)\underline{p_j}.$$

Par conséquent, en combinant ces trois cas, et comme $b/(b-1) \geq 1$, nous obtenons le lemme 4.3.2. □

En nous basant sur les lemmes précédents, nous pouvons en déduire le théorème suivant, qui donne un rapport d'approximation de l'algorithme HLP-b.

4.3.3 Rapport d'approximation

Théorème 4.3.3. HLP-b *atteint le rapport d'approximation suivant :*

$$3 + 2\sqrt{2 - \frac{k}{m} - \frac{k^2}{m^2} - \frac{1}{m}\left(1 - \frac{k}{m}\right)} - \frac{1}{m},$$

qui est majoré par $3 + 2\sqrt{2} \leq 5.83$.

Preuve. Tout d'abord, définissions quelques notations supplémentaires. Dans le planning décidé par l'algorithme, soient W^A_{CPU} (resp. W^A_{GPU}) la somme des temps d'exécution des tâches affectées aux CPUs (resp. GPUs), et CC^A la valeur du plus long chemin critique de G après la phase d'allocation de HLP-b. Notons $\mathcal{P}$ l'ensemble des chemins du graphe G.

$$W^A_{CPU} = \sum_{j \in V} \overline{p_j} x_j^A; \qquad W^A_{GPU} = \sum_{j \in V} \underline{p_j}(1 - x_j^A);$$
$$CC^A = \max_{p \in \mathcal{P}} \Big\{ \sum_{j \in p} (\overline{p_j} x_j^A + \underline{p_j}(1 - x_j^A)) \Big\}.$$

De façon similaire, nous définissons W^R_{CPU}, W^R_{GPU} et CC^R comme étant la charge totale des temps d'exécutions affectés aux CPUs, la charge totale affectée aux GPUs, et le Chemin Critique le plus long si l'on considère les durées des tâches après affectation par la solution optimale du programme linéaire relaxé.
De plus, nous notons $C^A_{\max}$, $C^R_{\max}$ et $C^*_{\max}$ (respectivement) : le makespan de l'ordonnancement crée par HLP-b, la valeur de la fonction objectif de la solution optimale du programme linéaire relaxé et le makespan de la solution optimale de notre problème. La borne 4.6 nous donne l'inégalité suivante :

$$\begin{aligned}
C^A_{max} &\leq \frac{W^A_{CPU}}{m} + \frac{W^A_{GPU}}{k} + \left(1 - \frac{1}{m}\right) CC^A \\
&= \frac{W^A_{CPU} + W^A_{GPU}}{m} + \frac{m-k}{mk} W^A_{GPU} + \left(1 - \frac{1}{m}\right) CC^A \\
&= \frac{1}{m} \sum_{j \in V} \left(x_j^A \overline{p_j} + (1 - x_j^A) \underline{p_j} \right) + \frac{m-k}{mk} \sum_{j \in V} (1 - x_j^A) \underline{p_j} \\
&\qquad + \left(1 - \frac{1}{m}\right) \max_{p \in \mathcal{P}} \Big\{ \sum_{j \in p} \left(x_j^A \overline{p_j} + (1 - x_j^A) \underline{p_j} \right) \Big\}
\end{aligned}$$

En utilisant les lemmes (4.3.1) et (4.3.2), nous obtenons :

$$\begin{aligned} C^A_{max} &\leq \frac{b}{b-1}\frac{1}{m}\sum_{j\in V}\left(x^R_j\overline{p_j} + (1-x^R_j)\underline{p_j}\right) + b\frac{m-k}{mk}\sum_{j\in V}(1-x^R_j)\underline{p_j} \\ &\quad + \left(1-\frac{1}{m}\right)\frac{b}{b-1}\max_{p\in\mathcal{P}}\left\{\sum_{j\in p}\left(x^R_j\overline{p_j} + (1-x^R_j)\underline{p_j}\right)\right\} \\ &= \frac{b}{b-1}\frac{W^R_{CPU}+W^R_{GPU}}{m} + b\frac{m-k}{mk}W^R_{GPU} + \left(1-\frac{1}{m}\right)\frac{b}{b-1}CC^R \end{aligned}$$

Les contraintes (4.1) à (4.4) du programme linéaire relaxé nous donnent :

$$C^A_{max} \leq \frac{b}{b-1}\frac{mC^R_{max}+kC^R_{max}}{m} + b\frac{m-k}{mk}kC^R_{max} + \left(1-\frac{1}{m}\right)\frac{b}{b-1}C^R_{max}.$$

Comme $C^R_{\max} \leq C^*_{\max}$ nous obtenons :

$$\begin{aligned} \frac{C^A_{max}}{C^*_{max}} &\leq \frac{b}{b-1}\cdot\frac{m+k}{m} + b\cdot\frac{m-k}{m} + \left(1-\frac{1}{m}\right)\frac{b}{b-1} \\ &= b + 2\cdot\frac{b}{b-1} - \frac{k}{m}(b-\frac{b}{b-1}) - \frac{1}{m}(\frac{b}{b-1}). \end{aligned} \tag{4.7}$$

Cette fonction atteint son minimum pour $b = 1+\sqrt{\frac{2+k/m-1/m}{1-k/m}} > 1+\sqrt{2}$, ce qui nous donne l'inégalité suivante :

$$\frac{C^A_{max}}{C^*_{max}} \leq 3 + 2\sqrt{2-\frac{k}{m}-\frac{k^2}{m^2}-\frac{1}{m}\left(1-\frac{k}{m}\right)} - \frac{1}{m} \leq 3+2\sqrt{2} \approx 5.83.$$

Les simplifications avec la valeur de b proposée qui mettent en évidence cette formulation de 4.7 qui a pour borne supérieure $3+2\sqrt{2}$ ne sont pas particulièrement triviales, sans être vraiment compliquées. Elles sont donc jointes dans l'annexe C.

□

4.4 Bornes inférieures conditionnelles sur le facteur d'approximation

Dans cette section, nous étendons les résultats de Bazzi et Norouzi-Fard [BN15] dans notre contexte. En supposant 1 (voir ci-dessous), ils montrent qu'il est NP-difficile d'approximer le problème $Q|prec|C_{\max}$ à un facteur constant. Si nous nous concentrons sur seulement deux types de processeurs, leur résultat implique une limite inférieure de 2 sur le rapport d'approximation, ce qui n'améliore pas le résultat de Svensson [Sve11]. Nous améliorons leur résultat pour obtenir une limite inférieure conditionnelle de 3 énoncée dans 4.4.1, qui est donc également valable dans notre cadre plus restreint $(Pm, Pk) \mid prec \mid C_{\max}$ dans lequel les temps de traitement des deux types de processeurs peuvent être arbitraires.

Hypothèse 1 (Problème q-parti)**.** *Pour tout petit $\varepsilon, \delta > 0$ et pour tous les entiers constants $q, Q > 1$: étant donné un graphe q-parti $G_q = (V_1, \ldots, V_q, E_1, \ldots, E_{q-1})$ avec $|V_i| = n$ pour tout $1 \leq i \leq q$ et E_i étant l'ensemble des arêtes entre V_i et V_{i+1} pour tout $1 \leq i < q$, il est NP-difficile de distinguer entre les deux cas suivants :*

- *Cas OUI : chaque V_i peut être partitionné en $V_{i,0}, \ldots V_{i,Q-1}$, de sorte que :*
  - *il n'y ait pas d'arête entre V_{i,j_1} et V_{i+1,j_2} pour tout $1 \leq i < q$, $j_1 > j_2$.*
  - *$|V_{i,j}| \geq \frac{1-\varepsilon}{Q} n$, pour tout $1 \leq i \leq q$, $0 \leq j \leq Q-1$.*
- *Cas NON : pour tout $1 \leq i < q$ et pour tous les deux ensembles $S \subseteq V_i$, $T \subseteq V_{i+1}$ tels que $|S| = |T| = \lfloor \delta n \rfloor$, il existe une arête entre S et T.*

Théorème 4.4.1. *En supposant l'hypothèse (1) vraie, et en supposant que $P \neq NP$: il n'existe pas de $(3-\alpha)$-approximation en temps polynomial pour le problème $(Pm, Pk) \mid prec \mid C_{\max}$, pour tout $\alpha > 0$, même si les processeurs sont liés.*

Nous commençons par fixer plusieurs valeurs : soient q un entier multiple de 3, Q un entier, $\delta \leq 1/(2Q)$ et $\varepsilon \leq 1/Q^2$. Nous considérons le problème q-parti paramétré par Q, ε, δ, qui est supposé être NP-difficile sous l'hypothèse (1).

Réduction. Nous définissons une réduction de $G_q = (V_1, \ldots, V_q, E_1, \ldots, E_{q-1})$, un graphe q-parti où pour chaque i, $|V_i| = n > Q$, vers une instance d'ordonnancement $\mathcal{I}$. L'instance consiste en $m = \lceil (1 + Q\varepsilon)n^4 \rceil$ CPU, $k = \lceil (1 + Q\varepsilon)n^2 \rceil$ GPU et deux types de tâches : des "tâches CPU" vérifiant $\overline{p_j} = n\underline{p_j} = 1$ et des "tâches GPU" vérifiant $\overline{p_j} = n\underline{p_j} = n$. Les tâches et les arcs (c'est-à-dire les contraintes de précédence) sont définies comme suit. Pour chaque $0 \leq z < q/3$, et pour chaque :

— sommet $v \in V_{3z+1}$, créer un ensemble $\mathcal{J}_{3z+1,v}$ de $Qn - Q$ tâches GPU (type a).

— sommet $v \in V_{3z+2}$, créer un ensemble $\mathcal{J}_{3z+2,v}$ de Qn^3 tâches CPU (type b).

— sommet $v \in V_{3z+3}$, créer un ensemble $\mathcal{J}_{3z+3,v}$ de $Q - 2$ tâches GPU (type c) indexées $J^1_{3z+3,v}, \ldots J^{Q-2}_{3z+3,v}$, et une arête de $J^{\ell}_{3z+3,v}$ à $J^{\ell+1}_{3z+3,v}$ pour ℓ allant de 1 à $Q - 3$.

— arcs $(v, w) \in E_i$, créer toutes les arcs de l'ensemble $\mathcal{J}_{i,v}$ vers l'ensemble $\mathcal{J}_{i+1,w}$.

Intuitivement, les tâches correspondant à chaque ensemble V_i de G_q peuvent être calculées en Q créneaux de temps. Pour y parvenir, chaque ensemble de type b nécessite presque tous les CPU, chaque ensemble de type a nécessite presque tous les GPU sauf n, et chaque ensemble de type c nécessite n GPU.

Dans une instance OUI, il est possible de progresser simultanément sur les tâches correspondant à trois ensembles consécutifs V_i, en pipelineant l'exécution, obtenant ainsi un makespan proche de $qQ/3$. Par exemple, il est possible d'exécuter $V_{i,1}$ à un certain temps, puis d'exécuter $V_{i+1,1}$ et $V_{i,2}$ en parallèle.

Dans une instance NON, les tâches correspondant à chaque V_i doivent être planifiées presque indépendamment, ce qui ne permet pas d'utiliser efficacement la puissance de calcul : il y a trop peu de GPU pour traiter une quantité importante de tâches CPU, et les CPU sont trop lents pour traiter les tâches GPU. Le makespan minimum possible est alors proche de qQ.

Les lemmes Completeness (4.4.2) et Soundness (4.4.3) formalisent ces résultats et sont prouvés dans l'Annexe (D).

Lemme 4.4.2 (Completness)**.** *Si G_q correspond au cas OUI du problème q-parti, alors l'instance $\mathcal{I}$ admet un planning avec un makespan de $(q + 3)Q/3$.*

Lemme 4.4.3 (Soundness)**.** *Si G_q correspond au cas NON du problème q-parti, alors l' ordonnancement issu de l'instance $\mathcal{I}$ a un makespan d'au moins $f(Q)qQ$, où f tend vers 1 lorsque Q croît.*

Nous sommes maintenant prêts à compléter la preuve.

Preuve du théorème 4.4.1.

Soit $\alpha > 0$ et choisissons q et Q de telle sorte que $f(Q)qQ > (3-\alpha)(q+3)Q/3$. Considérons une instance G_q du problème q-parti correspondant, avec $n > Q$. En raison de 4.4.2 et 4.4.3, si G_q est une instance OUI, alors son makespan optimal est au plus $(q+3)Q/3$, sinon, son makespan vaut au moins $f(Q)qQ > (3-\alpha)(q+3)Q/3$.

Par conséquent, un algorithme qui approche le problème d'ordonnancement à un facteur $3-\alpha$ résout également le problème q-parti en temps polynomial, ce qui contredit 1 ou $P \neq NP$. □

Nous pouvons en outre adapter cette preuve pour montrer le résultat suivant :

Corollaire 4.4.3.1. *En supposant vraie l'hypothèse 1 et $P \neq NP$, le problème $(Pm, Pk) \mid prec \mid C_{\max}$ ne peut pas avoir de $3-\alpha$-approximation, pour tout $\alpha > 0$ et toute valeur de m/k.*

Preuve (Esquisse de preuve). Nous définissons les tâches CPU comme $\overline{p_j} = 1$ et $\underline{p_j} = \infty$, et les tâches GPU comme $\overline{p_j} = \infty$ et $\underline{p_j} = 1$. La valeur de k est la même que précédemment, mais nous considérons maintenant toute valeur de $m \geq k$, et nous définissons les ensembles de type b comme contenant $n_b = \lfloor Qmn/k \rfloor$ tâches au lieu de Qn^3. Le lemme de Completness est toujours valide car $(1+Q\varepsilon)n \cdot n_b \leq m$ et le lemme de Soundness est respecté car les tâches ne peuvent pas être exécutées sur l'autre type de ressource. □

Ce résultat est intéressant car le rapport de compétitivité des algorithmes connus pour $(Pm, Pk) \mid prec \mid C_{\max}$, à la fois dans le cas hors ligne et dans le cas en ligne ($1 + 2\sqrt{m/k}$ [Can+18]), tendent vers 3 lorsque m/k est proche de 1, il n'y a donc pas d'écart entre la borne inférieure conditionnelle et la borne supérieure pour ce cas. Il est à noter que ce résultat de difficulté est également valable si un oracle fournit l'affectation (CPU ou GPU pour chaque tâche), dans ce cas la planification par List donne un rapport de compétitivité de 3 [Can+18, Théorème 7]. Par conséquent, l'écart entre la borne inférieure conditionnelle et l'algorithme HLP-b est principalement dû à la difficulté de l'allocation.

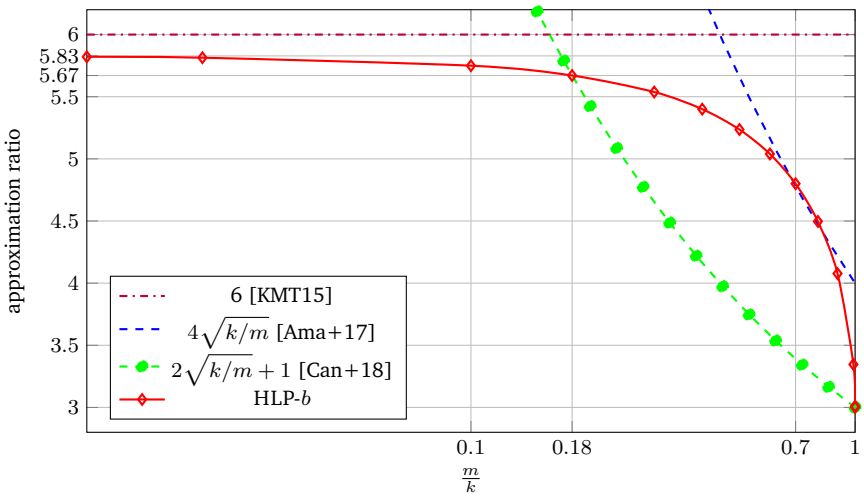


Figure 4.3. : Une comparaison des rapports d'approximation des quatre algorithmes connus pour ce problème d'ordonnancement sur plateforme hybride, avec contrainte de précédence entre les tâches, en fonction du rapport $\frac{k}{m}$, en faisant abstraction des composantes dépendant de $\frac{1}{m}$ pour nos résultats.

4.5 Conclusions et perspectives

Nous proposons un algorithme de $\left(3+2\sqrt{2}\right)$-approximation HLP-b pour le problème $(Pm, Pk) \mid prec \mid C_{\max}$. Notre algorithme améliore le rapport d'approximation par rapport au précèdent algorithme connu dans la littérature qui propose une approximation de 6. Nous utilisons une procédure d'arrondi différente, qui bien qu'elle ne soit pas optimale pour la phase d'allocation, conduit à un meilleur rapport de pire cas pour l'ensemble du problème. Nous montrons également une borne inférieure conditionnelle de 3 sur le rapport d'approximation pour ce problème, en supposant une variante généralisée de la conjecture des jeux uniques, améliorant le résultat précédent de 2.

La figure (4.3) représente la valeur des rapports obtenus par les différents algorithmes d'approximation, introduits dans la Sous-Section (4.1.1), connus pour ce problème. Le rapport d'approximation de HLP-b diminue en réalité vers 3 lorsque m et k sont proches, clôturant ainsi l'écart avec la borne inférieure pour $m = k$. L'objectif naturel serait de réduire cet écart pour toutes valeurs de m et k.

Les résultats publiés permettent d'identifier un travail antérieur effectué par Chudak et Shmoys (1997) qui ont étudiés en particulier les problèmes $Q \mid r_j, prec \mid C_{max}$ et $Q \mid prec, pmtn \mid C_{max}$ dans [CS97], qui sont étroitement liés à $(Pm, Pk) \mid prec \mid C_{\max}$. Cette publication présente un algorithme avec un rapport logarithmique par rapport au nombre de machines. Il est intéressant de noter qu'un théorème inter-

médiaire dans leur analyse prouve un rapport d'approximation de $K + 2\sqrt{K} + 1$ (avec K = le nombre de machines différentes dans Q), qui, si nous choisissons de ne pas tenir compte de la cardinalité de nos ensembles de machines, est équivalent au rapport d'approximation que nous présentons dans ce chapitre, la preuve étant également basée sur une technique de programmation linéaire. Bien que cela nous conforte dans nos résultats, il pourrait être conjecturé que les propriétés qui caractérisent les cas pathologiques pour nos algorithmes pourraient être similaires entre $Q \mid prec \mid C_{max}$ et $(Pm, Pk) \mid prec \mid C_{\max}$, en raison des techniques sur lesquelles nos algorithmes sont basés.

Cela pourrait également suggérer, si nous arrivons à généraliser nos résultats pour des systèmes hybrides avec d'autres groupes de machines (CPU / GPU / FPU / ...) que les résultats présentés dans [CS97] ne dépendent pas directement du rapport entre les vitesses des machines. Dans ce cas, il pourrait être conjecturé que suivre la même logique permettrait de borner le problème plus général $R \mid prec \mid C_{max}$.

5 Utilisation de l'oracle au pire cas

Ce chapitre présente plusieurs travaux en cours, en collaboration avec Giorgio Lucarelli, Yuqiang Ma, et Denis Trystram.

5.1 Introduction

Dans le domaine de l'ordonnancement, l'optimisation robuste joue un rôle crucial lorsqu'il existe des incertitudes dans le système. Ces incertitudes peuvent provenir de diverses sources telles que les variations de temps de traitement des tâches ou les perturbations externes. L'objectif est de concevoir des ordonnancements capables de résister à ces incertitudes et de garantir des performances satisfaisantes, même dans les pires scénarios.

Rappelons la définition initiale du problème (unique) d'oracle (online) tel que définit dans le Chapitre (2). Nous disposons d'une permutation composée de n_L tâches longues de durée p_L et de n_S tâches courtes de durées p_S. Une fois lancées, les tâches ne peuvent pas être interrompues. À chaque fois qu'une tâche arrive, nous pouvons décider d'appeler un oracle en faisant un test, et en fonction de la valeur de ce test, nous pouvons décider de retarder cette tâche. Interroger l'oracle prend un certain temps (p_T) et aucune autre tâche ne peut être traitée simultanément. Dès le début de l'ordonnancement, l'ordonnanceur connaît ces 5 valeurs (n_S,n_L,p_S,p_L,p_T) qui suffisent à définir une instance I. L'objectif de l'ordonnanceur est de minimiser la valeur totale des temps de complétion des tâches.

La formalisation classique d'incertitude sous forme d'intervalle ne semble pas adaptée car nous nous trouvons dans un cas binaire (une tâche est soit longue, soit courte). Par conséquent, nous nous tournons vers d'autres approches pour aborder cette problématique.

Dans ce chapitre, nous débutons en abordant le problème classique de la minimisation du pire cas. Cette approche constitue un point de départ pour aborder le problème de l'ordonnancement dans un contexte incertain.

Si l'on n'a pas l'aide d'un oracle ou d'un mécanisme externe, la version non clairvoyante du problème d'ordonnancement devient difficile. Voici un exemple pour illustrer cette difficulté.

Considérons l'instance $(n_L = 1, n_S = n-1, p_L \neq 0, p_S = 0)$. Dans ce cas, il n'est pas possible d'obtenir un rapport d'approximation inférieur à n. Le meilleur ordonnancement possible, illustré dans la Figure (5.1), donne une valeur de $\sum C_j = p_L$. Cependant, si la première tâche à être exécutée est la tâche longue, comme dans la Figure (5.2), $\sum C_j$ sera égal à $n \cdot p_L$.

Figure 5.1. : Meilleur ordonnancement possible pour l'instance $(n_L = 1, n_S = n-1)$ sans possibilité de test. La valeur de la fonction objectif vaut $p_L + \frac{(n-1)(n+2)}{2} \cdot p_S$

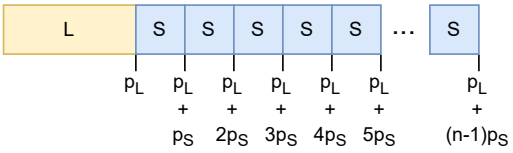


Figure 5.2. : Pire ordonnancement possible pour l'instance $(n_L = 1, n_S = n-1)$ sans possibilité de test. La valeur de la fonction objectif vaut $n \cdot p_L + \frac{n(n-1)}{2} \cdot p_S$

Pour surmonter cet exemple, nous introduisons un (unique) modèle d'oracle (online). Le fonctionnement de cet oracle est le suivant : à chaque fois qu'une tâche arrive, nous décidons de faire un test ou non, et en fonction de la valeur de ce test, nous décidons de retarder ou non cette tâche.

Une question qui se pose est de savoir quelle est la version *hors ligne* de ce problème d'oracle. Le terme *hors ligne* signifie que nous avons accès aux valeurs des données sans faire de test. Cependant, cela ne répond pas à la question de savoir comment nous accédons à ces données dans ce problème. Il est nécessaire de définir à quel moment nous avons accès à ces données et quel est le coût de leur utilisation.

Nous envisageons deux façons de modéliser ce problème :

— *Version clairvoyante, offline sur la liste :* Nous avons une vision globale de toutes les tâches dès le début, et nous effectuons les appels à l'oracle nécessaires. Intuitivement, nous pouvons choisir l'ordre d'arrivée des tâches en utilisant les tests.

— *Version (semi-)clairvoyante, online sur la liste :* Les tâches arrivent de façon online (nous ne choisissons pas leur ordre d'arrivée), mais nous connaissons la valeur du temps d'exécution d'une tâche dès son arrivée. Nous avons uniquement la possibilité de choisir la politique de test et l'action à prendre (exécuter ou retarder) en utilisant ces tests.

Remarque : Si il est possible d'accéder librement aux tâches (dont nous connaissons les temps d'exécution dans le cas offline), la politique optimale revient à utiliser l'algorithme SPT. On parlera alors du problème "omniscient". Comme nous avons déjà pu le justifier, nous nous intéressons à ces versions offline moins déloyales pour nous que SPT, qui pourra néanmoins toujours être considéré comme une borne inférieure de celles ci.

Il existe entre autres deux raisons pour lesquelles il est intéressant d'étudier la version hors-ligne de notre modèle. Tout d'abord, l'analyse du modèle hors ligne nous permettra d'obtenir des informations sur les propriétés structurelles ainsi que des bornes inférieures qui seront également valables dans le cas non-clairvoyant. Deuxièmement, dans la section 5.4, nous examinons comment étendre notre modèle en utilisant la métrique du min-max regret, ce qui nécessitera l'utilisation d'un résultat basé sur le modèle clairvoyant en ligne sur la liste.

Bien que nous utilisions le terme "modèle hors-ligne", il ne correspond pas à un véritable cas hors-ligne, car nous faisons toujours appel à un oracle non pas pour obtenir des informations, mais pour s'autoriser à utiliser certaines de ces informations. Cette distinction explique pourquoi nous continuons à examiner le rapport compétitif entre le modèle hors-ligne et SPT.

5.2 Un problème offline, avec oracle

5.2.1 Version clairvoyante, offline sur la liste

Dans cette version, nous connaissons les valeurs des données, et l'accès à ces données a un certain coût associé. Considérons que nous avons à disposition un ensemble identifié de tâches courtes, et un ensemble identifié de tâches longues. Payer le coût d'un test à un certain instant de l'ordonnancement permet de choisir dans quel ensemble *piocher* la prochaine tâche à traiter. Ainsi le coût du test est payé juste avant l'exécution de la tâche concernée.

> Remarque : Si l'algorithme optimal pour ce problème décide de tester (donc payer) pour une tâche longue, c'est bien pour utiliser cette information, donc il choisira de l'exécuter à la fin de l'ordonnancement. En décidant d'identifier une tâche longue à l'instant $t = 0$, le test ainsi que la tâches seront tout deux exécutés à la toute fin de l'ordonnancement.
> Ainsi, identifier toutes les tâches longues à $t = 0$ (en payant les tests à $t > 0$) contraint l'adversaire à ne fournir que des tâches courtes au début de l'ordonnancement.

Quelques règles de dominance :

— Si l'algorithme décide de tester une tâche longue, sa meilleure décision est de "la tester puis l'exécuter" le plus tard possible.
— Pour toutes les tâches non testées, l'adversaire a intérêt à fournir les tâches longues avant les tâches courtes (pour maximiser les surcoûts SC).

Prenons un algorithme $Néo$ qui décide de tester et d'exécuter immédiatement (au début du planning) X tâches courtes et Y tâches longues exactement.

> $Néo$ n'est pas optimal, dans la mesure où si on suppose que $X = 2$ et $Y = n_L$ testées, il serait préférable d'exécuter sans tester les $(n_s - 2)$ tâches courtes restantes, puis d'ordonnancer les 2 tâches courtes (avec leurs tests), puis finir par les Y tâches longues. Mais ce n'est pas grave, car ici, pour savoir si ce modèle offline est pertinent, nous souhaitons en trouver une borne supérieure. Le résultat (non optimal) trouvé par $Néo$ en est une.

La pire permutation que peut alors fournir l'adversaire entraîne la trace d'exécution présente dans la Figure (5.3).

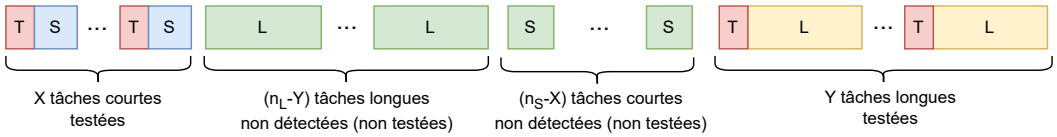


Figure 5.3. : Trace d'exécution de la pire permutation fournie par l'adversaire à $Néo$.

On note $C(X,Y)$ la valeur de $\sum C_j$ donnée par l'ordonnancement décrit par la Figure (5.3).

$$\begin{aligned} C(X,Y) = p_L & \left(\frac{n_L^2 + n_L}{2} + n_L n_S - n_L X - Y n_S + XY \right) \\ & + p_S \left(\frac{n_S^2 + n_s}{2} + n_L X + n_S Y - XY \right) \\ & + p_T \left(X(n_L + n_S + \frac{1}{2} - \frac{X}{2}) + Y \frac{Y+1}{2} \right) \end{aligned} \tag{5.1}$$

Nous cherchons alors la solution (X,Y) qui permette d'obtenir les meilleures performances pour $Néo$. On notera $C_{Néo}$ la valeur de la fonction objectif ainsi trouvée avec cette solution.

Lemme 5.2.1. *Pour toute solution (X,Y), on a :* $C(k-1,n_L) < C(k,n_L)\ \forall k \in (1,n_S)$

Preuve. Comme $(k \leq n_S)$, on utilisant directement l'équation (5.1), on obtient :

$$C(k-1,n_L) - C(k,n_L) = -p_T(n_L + (n_S - k) + 1) < 0$$

□

Lemme 5.2.2. *Pour toute solution (X,Y), on a :* $C(n_S,k-1) < C(n_S,k)\ \forall k \in (1,n_L)$

Preuve. Comme $(k \geq 1)$, en appliquant l'équation (5.1), on obtient :

$$C(n_S,k-1) - C(n_S,k) = p_T(-k) < 0$$

□

On considère maintenant deux domaines de D = $\{0 \leq Y \leq n_L,\ 0 \leq X \leq n_S\}$ tel que $D = D_1 \cup D_2$, en introduisant la coupe suivante :

$$\Delta((n_L - Y) - (n_S - X)) \lesseqgtr p_T((n_L - Y) + (n_S - X)) \qquad (5.2)$$

— $D_1 = D \cap \{$(X,Y) tel que $\Delta((n_L - Y) - (n_S - X)) \geq p_T((n_L - Y) + (n_S - X))\}$

— $D_2 = D \cap \{$(X,Y) tel que $\Delta((n_L - Y) - (n_S - X)) \leq p_T((n_L - Y) + (n_S - X))\}$

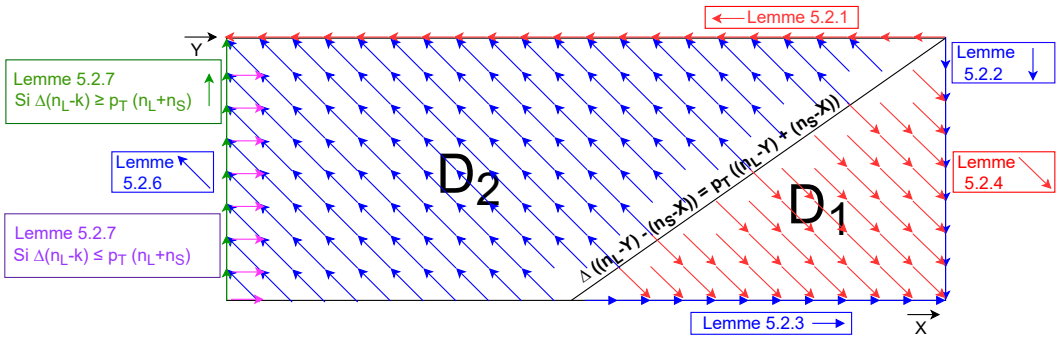


Figure 5.4. : Décomposition de l'espace de solutions et représentation des lemmes. Chaque point de cet espace représente une solution (X,Y). Une flèche allant d'une solution $A = (X_A, Y_A)$ vers une solution $B = (X_B, Y_B)$ symbolise que, au regard de l'équation (5.1), la solution B est meilleure que la solution A : c'est à dire que $C(X_B, Y_B) \leq C(X_A, Y_A)$.

Esquisse de la preuve : L'espace de toutes les solutions (X,Y) est représenté sur la Figure (5.4). Nous proposons une coupe pour séparer l'espace des solutions en deux, et nous prouvons que chacun de deux domaines converge vers l'un des deux points suivant $(0,n_L)$ ou $(n_S,0)$. Le Théorème (5.2.1) donne la meilleure valeur de la Formule (5.1).

Sous-domaine D_1 (inférieur)

Lemme 5.2.3. *Pour toutes les solutions (X,0) et (X + 1,0) dans D_1, on a :*

$$C(X,0) \geq C(X+1,0)$$

Preuve. En utilisant l'inéquation de la coupe (5.2) de D_1, pour $Y = 0$, on a :

$$\begin{aligned} p_T(n_L + n_S - X) &\leq \Delta(n_L - n_S + X) \\ &= \Delta n_L - \Delta(n_S - X) \\ &\leq \Delta n_L \text{ car } X \leq n_S \end{aligned}$$

En utilisant directement l'équation (5.1), on obtient :

$$C(X+1,0) - C(X,0) = p_T(n_L + n_S - X) - \Delta n_L \leq 0$$

□

Lemme 5.2.4. *Pour toutes les solutions (X,Y) et (X+1,Y-1) dans D_1, on a*

$$C(X,Y) \geq C(X+1,Y-1)$$

Preuve. En utilisant l'inéquation donnée par la coupe (5.2) de D_1, on a :

$$p_T((n_L - Y) + (n_S - X)) - \Delta((n_L - Y) - (n_S - X)) \leq 0$$
$$\text{Donc : } p_T(n_L - Y + n_S - X) + \Delta(-n_L + Y + n_S - X) - \Delta \leq -\Delta \leq 0$$

En utilisant directement l'équation (5.1), on obtient :

$$\begin{aligned} C(X+1,Y-1) - C(X,Y) &= p_T(n_L + n_S - X - Y) \\ &\quad + \Delta(-L + S - X + Y - 1) \leq 0 \end{aligned}$$

□

Lemme 5.2.5. *La solution (n_S,0) est la solution minimale du sous-domaine D_1.*

Preuve. CQFT à partir des lemmes (5.2.2), (5.2.3) et (5.2.4). □

Sous-domaine D_2 (supérieur)

Lemme 5.2.6. *Pour toutes les solutions (X,Y) et (X − 1,Y + 1) dans D_1, on a :*

$$C(X,Y) \geq C(X-1,Y+1)$$

Preuve. En utilisant directement l'équation (5.1), on obtient :

$$\begin{aligned} C(X-1,Y+1) - C(X,Y) &= p_T(-n_L - n_S + X + Y) + \Delta(L - S + X - Y - 1) \\ &= -\Delta - \left(p_T(n_L - Y + n_S - X) - \Delta(L - Y + X - S)\right) \\ &\leq -\Delta \qquad \text{(en utilisant la coupe 5.2)} \end{aligned}$$

□

Lemme 5.2.7. *Pour toute Solution de D_2 de la forme (X,Y) = (0,k) avec $k \leq n_L$:*

- *soit :* $C(0,k) \leq C(0,k-1)$
- *soit :* $C(0,k) \leq C(1,k)$

Preuve. Nous nous intéressons à la valeur de $\Delta(n_L - k) + p_T(-n_L - n_S)$:
Si $\Delta(n_L - k) \geq p_T(n_L + n_S)$:

$$C(0,k) - C(0,k-1) = -n_S\Delta + p_T \cdot k$$

et $p_T \cdot k \leq p_T(n_L + n_S) - \Delta(n_L - k - n_S)$ (à partir de (5.2))

$$\begin{aligned} \text{donc : } C(0,k) - C(0,k-1) &\leq -n_S\Delta + \left(p_T(n_L + n_S) - \Delta(n_L - k - n_S)\right) \\ &\leq p_T(n_L + n_S) - \Delta(n_L - k) \\ &\leq 0 \text{ (selon la condition du Si)} \end{aligned}$$

Si $\Delta(n_L - k) \leq p_T(n_L + n_S)$:

$$C(0,k) - C(1,k) = \Delta(n_L - k) - p_T(n_L + n_S) \leq 0 \text{ avec la condition du Si}$$

Dans ce cas, nous avons $C(0,k) \leq C(1,k)$. Et nous avons deux cas de figure possibles :

- la solution (1,k) est encore à l'intérieur du sous-domaine (D_2) et en appliquant le Lemme (5.2.6) sur la partie droite,
  nous avons $C(0,k) \leq C(1,k) \leq C(0,k+1)$
- Ou la solution (1,k) tombe dans le sous-domaine (D_1) donc nous pouvons appliquer le lemme (5.2.5) et on obtient (n_S, 0) est une meilleure solution.

□

Lemme 5.2.8. *Toute solution de D_2 est dominée soit par (0,n_S), soit par (0,n_L).*
Preuve. CQFD des lemmes précédents. □

Théorème 5.2.9. *La solution optimale pour ce problème est soit (X=0 et Y=n_L), soit ($X = n_S$ et $Y = 0$). Donc la solution optimale ($C_{Néo}$) vaut :*

$$C_{Néo} = \text{SPT} + p_T \cdot min\left\{(n_S\left(n_L + \frac{n_S}{2} + \frac{1}{2}\right), \frac{n_L(n_L+1)}{2}\right\}$$

Preuve. En utilisant les lemmes (5.2.5) et (5.2.1), on sait que la solution optimale est (0,n_L) ou (n_S,0).
En utilisant l'équation (5.1) sur ces deux solutions, on obtient :

— $C(n_S, 0) = \frac{p_L}{2}(n_L^2 + n_L) + p_S(\frac{n_S^2}{2} + \frac{n_s}{2} + n_S n_L) + p_T \cdot n_S\left(n_L + \frac{n_S}{2} + \frac{1}{2}\right)$
— $C(0, n_L) = \frac{p_L}{2}(n_L^2 + n_L) + p_S(\frac{n_S^2}{2} + \frac{n_s}{2} + n_S n_L) + p_T\left(\frac{n_L}{2}(n_L+1)\right)$

□

Puissance de ce modèle offline

Soit $\lambda = p_L + \left(\frac{p_L}{\Delta} + \frac{2n_S p_S}{\Delta(n_L+1)}\right)p_S$. Notons que $\lambda \geq p_S$ et que $\lambda \geq p_L$.

$$\begin{aligned}
C_{Néo} - \text{SPT} =& p_T \cdot min\left\{n_S n_L + \frac{n_S^2}{2} + \frac{n_S}{2}, \frac{n_L(n_L+1)}{2}\right\} \\
\leq& p_T \cdot \left(\frac{p_S}{\lambda}\left(n_S n_L + \frac{n_S^2}{2} + \frac{n_S}{2}\right) + \left(\frac{p_L}{\lambda} + \frac{\lambda - p_S - p_L}{\lambda}\right)\left(\frac{n_L(n_L+1)}{2}\right)\right) \\
=& \frac{p_T}{\lambda}\left(\text{SPT} + \left(\frac{p_L}{\Delta} + \frac{2n_S p_S}{\Delta(n_L+1)} - 1\right)p_S\left(\frac{n_L(n_L+1)}{2}\right)\right) \\
\leq& \frac{p_T}{\lambda}\left(\text{SPT} + \frac{1}{\Delta}p_S\left(p_L\frac{n_L(n_L+1)}{2} + (2n_S p_S)\frac{n_L}{2}\right)\right) \\
\leq& \frac{p_T}{\lambda}\left(\text{SPT} + \frac{p_S}{\Delta}SPT\right) = \frac{p_T p_L}{\Delta\lambda}SPT \leq \frac{p_L}{\lambda}SPT \leq SPT
\end{aligned}$$

D'où $C_{\text{Néo}} \leq \text{SPT} + \frac{p_L}{\lambda}\text{SPT} \leq 2\text{SPT}$

Nous constatons que ce modèle, dans lequel nous avons donné énormément de pouvoir à l'algorithme, ne nous éloignera pas particulièrement de SPT. Proposer un modèle d'oracle dont les coûts sont payés de la sorte revient (à une constante près) quasiment à révéler toutes les incertitudes pour $p_T < p_L - p_S$ (entre autres). Cela nous donne l'intuition qu'un modèle qui accorde de l'intérêt à une permutation donnée en entrée (par un adversaire) serait plus prometteur.

5.2.2 Version (semi-)clairvoyante, online sur la liste

Dans cette version offline du problème, on considère une permutation fixée et pour laquelle les tâches arrivent une par une. Nous ne connaissons le temps d'exécution d'une tâche au moment où elle arrive, mais pour utiliser cette information pour reporter l'exécution d'une tâche longue, il est nécessaire de payer immédiatement le coût d'un test. La permutation revêt alors une importance cruciale.

> Remarques :
> Cette présentation "Semi-Clairvoyante" (on ne voit que la valeur de la tâche qui arrive) diffère de la définition de ce modèle vu dans cette Section (5.2), la conclusion de cette sous-section en explique la raison.
> On peut se rendre compte que considérer ce problème initial, semi-clairvoyant, revient à considérer un modèle non-clairvoyant qui aurait accès à un oracle différent ; pour lequel tester une tâche qui s'avère être courte coûte 0.

Nous pouvons proposer un algorithme de programmation dynamique qui résout ce problème de manière optimale en $O(n^3)$ en appliquant le schéma récursif présenté dans la Figure (5.5). Cependant, nous ne sommes pas en mesure de proposer une formule de la valeur de sa fonction objectif, ni d'exposer de règles de dominance simples qui permette de résoudre ce problème.

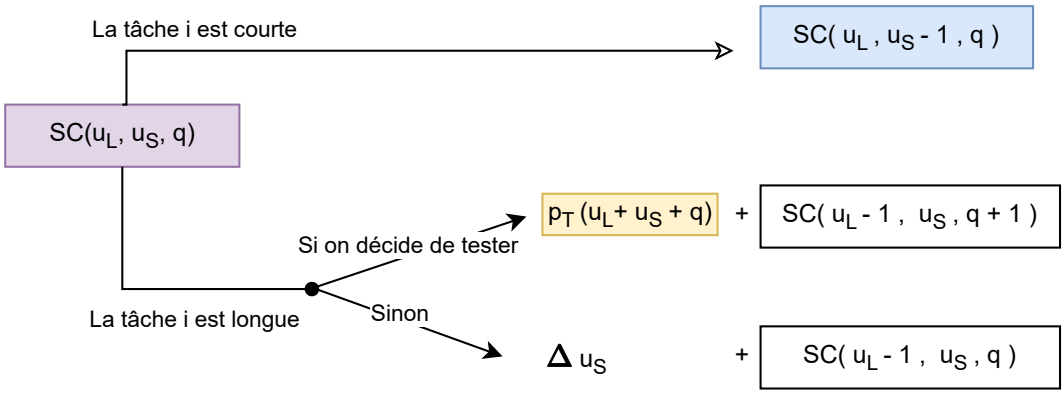


Figure 5.5. : Schéma récursif d'un algorithme optimal qui résout le problème présenté dans le modèle *Retarder une longue coûte*.

Bien que nous ne fournissons pas de formule de la valeur de l'optimal de ce modèle, nous pouvons construire la borne supérieure suivante : les décisions prises sont celles de l'arbre (5.5), mais les valeurs du surcoût que l'on considère dans chaque cas sont celles issues de la Figure (5.6).

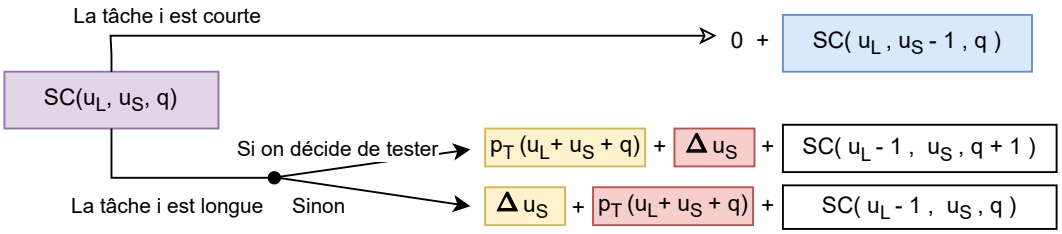


Figure 5.6. : Schéma récursif qui donne une borne supérieure pour l'algorithme optimal décrit par la Figure (5.5).

Ainsi, la valeur du surcoût induit par le fait de rencontrer une tâche courte est 0, alors que le surcoût de rencontrer (à l'instant où on la rencontre) une tâche longue est :

$$\begin{aligned} p_T(u_S + u_L + q) + \Delta u_S &\leq \Delta(u_S + u_L + q) + \Delta u_S \\ &= \Delta(u_L + q) \leq \Delta(n_L) \end{aligned}$$

$$\begin{aligned} \frac{C_{\text{Semi-clairvoyant, online sur la liste}}}{\text{SPT}} &\leq \frac{\text{SPT} + n_S \cdot 0 + n_L \cdot \Delta(n_L)}{p_S \frac{n_S(n_S+1)}{2} + p_S n_S n_L + p_L \frac{n_L(n_L+1)}{2}} \\ &< 1 + \frac{n_L \cdot p_L(n_L)}{p_L \frac{n_L(n_L)}{2}} = 3 \end{aligned}$$

On peut se rendre compte de la chose suivante : si on considère une modélisation complètement clairvoyante (où l'algorithme à la connaissance de la position des tâches dans la totalité de la permutation à l'instant 0 mais doit néanmoins payer p_T pour retarder immédiatement une tâche longue), il sera possible de construire un optimal directement à partir de la Figure (5.5). Nous pouvons donc en déduire qu'avoir la connaissance de la permutation entière n'apporte aucun *pouvoir* supplémentaire à l'algorithme par rapport à la connaissance uniquement de la tâche suivante (et de n_L et n_S).

Ce modèle semi-clairvoyant semble également assez proche de SPT, malgré le fait qu'il soit nécessaire pour l'algorithme de payer le coût d'un test pour avoir le droit d'utiliser les informations *supplémentaires*.

5.3 Problème Online

Rappelons la définition initiale du modèle (unique) d'oracle (online). A chaque fois qu'une tâche arrive, nous décidons de faire un test ou non, et en fonction de la valeur de ce test, nous décidons de retarder ou non cette tâche. Nous disposons de n_L tâches longues de durée p_L ainsi que de n_S tâches courtes de durées p_S. Interroger l'oracle prend un certain temps (p_T) et aucune autre tâche ne peut être traitée simultanément. De plus, une fois lancées, les tâches ne peuvent pas être interrompues. Dès le début de l'ordonnancement, l'ordonnanceur connaît ces 5 valeurs (n_S,n_L,p_S,p_L,p_T) qui suffisent à définir une instance I.

De même que dans la Section (2.4), un algorithme de type minmax se prête très bien pour résoudre ce type de problème. L'Algorithme 4 qui repose sur une programmation dynamique et une logique similaire à celle de l'Algorithme 2 ($OptimumPrime$) semble être tout adapté à la minimisation du pire cas.

Algorithme 4 : Programmation Dynamique Minmax$(n_L, n_S, p_T, p_L, p_S)$

Initialisation des valeurs de $SC[i][j][k]$ à 0 et de $Tester[i][j][k]$ à Faux
pour u_L de 1 à n_L :
  pour u_S de 1 à n_S :
    pour q de 0 à $n_L - u_L$:
      $a \leftarrow p_T \cdot (u_L + u_S + q) + \max\{SC[u_L][u_S - 1][q]\ ,\ SC[u_L - 1][u_S][q + 1]\}$
      $b \leftarrow \max\{(0 + SC[u_L][u_S - 1][q])\ ,\ (\Delta \cdot u_S + SC[u_L - 1][u_S][q])\}$
      si a<b :
        $SC[u_L][u_S][q] \leftarrow a$
        $Tester[u_L][u_S][q] \leftarrow Vrai$
      sinon :
        $SC[u_L][u_S][q] \leftarrow b$

Toutefois, l'analyse du pire cas de cet algorithme n'est pas une tâche facile, et il est difficile de montrer des bornes théoriques qui permettraient (à l'image d'un rapport de compétitivité) de savoir à quel point la solution optimale qui utiliserait l'oracle est proche ou non de SPT. Ça permettrait en quelque sorte de valider l'utilisation d'un modèle comme l'oracle.

Sur ce problème spécifique, les travaux en cours portent sur la recherche de règles de dominances,

— pour pouvoir proposer d'autres algorithmes également optimums (plus simples à analyser) (en utilisant des règles de dominance dans certains cas, ou proposer d'autres types de schéma récursif...)

— pour pouvoir fournir des heuristiques gloutones (moins lourdes) a faire tourner qu'une Programmation Dynamique (au moins partiellement),

— pour avoir une meilleure compréhension des subtilités du problème

Sur l'exemple présenté dans la Figure (5.7), on considère une instance $n_L = 2$, $n_S = 1$, et toutes les décisions possibles d'un algorithme non-adaptatif. La meilleure décision pour l'algorithme est de tester uniquement la deuxième tâche. La même analyse est effectuée dans le cadre adaptatif, et les résultats sont présentés dans la Figure (5.8) et la conclusion est la même que dans le cadre non-adaptatif.

Pourquoi la deuxième tâche ?
Il nous manque encore quelque chose au niveau de la compréhension du problème, du comportement de l'adversaire et des algorithmes optimums.

Adversaire	Trace (ne rien tester)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	SLL	3	3	0	30
LSL	LSL	4	2	0	38
LLS	LLS	5	1	0	46
Adversaire	Trace (tester J_1)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	TSLL	3	3	3	39
LSL	TSLL	3	3	3	39
LLS	TLSL	4	2	3	47
Adversaire	Trace (tester J_2)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	STLL	3	3	2	36
LSL	LTSL	4	2	2	44
LLS	LTSL	4	2	2	44
Adversaire	Trace (tester J_3)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	SLTL	3	3	1	33
LSL	LSTL	4	2	1	41
LLS	LLTS	5	1	1	49
Adversaire	Trace (tester J_1 et J_2)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	TSTLL	3	3	5	45
LSL	TTSLL	3	3	6	48
LLS	TTSLL	3	3	6	48
Adversaire	Trace (tester J_2 et J_3)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	STTLL	3	3	4	42
LSL	LTSTL	4	2	3	47
LLS	LTTSL	4	2	4	50
Adversaire	Trace (tester J_1 et J_3)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	TSLTL	3	3	4	42
LSL	TSTLL	3	3	5	45
LLS	TLTSL	4	2	5	53
Adversaire	Trace (tester toutes les tâches)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	TSTTLL	3	3	7	51
LSL	TTSTLL	3	3	8	54
LLS	TTTSLL	3	3	9	57

Figure 5.7. : Les différentes valeurs de $\sum C_j$ pour (p_L=9, p_S=1, p_T=3, n_L =2, n_S=1) dans le contexte non-adaptatif. Le pire cas pour l'algorithme (selon la politique de tests choisie) apparaît en couleur dans le tableau. On appelle J_i la $i^{\text{ème}}$ tâche de la permutation. Les différentes colonnes désignent :
1) la permutation donnée par l'adversaire
2) la trace d'exécution avec la politique de test envisagée
3, 4 et 5) le nombre d'occurrences de p_L, p_S et p_T dans la fonction objectif
6) la valeur de la fonction objectif au vu des valeurs de p_L, p_S et p_T

Adversaire	Trace (ne rien tester)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	SLL	3	3	0	30
LSL	LSL	4	2	0	38
LLS	LLS	5	1	0	46
Adversaire	Trace (tester J_1)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	TSLL	3	3	3	39
LSL	TSLL	3	3	3	39
LLS	TLSL	4	2	3	47
Adversaire	Trace (tester J_2)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	**SLL**	3	3	**0**	**30**
LSL	LTSL	4	2	2	44
LLS	LTSL	4	2	2	44
Adversaire	Trace (tester J_3)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	**SLL**	3	3	**0**	**30**
LSL	**LSL**	4	2	**0**	**38**
LLS	**LLS**	5	1	**0**	**46**
Adversaire	Trace (tester J_1 et J_2)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	**TSLL**	3	3	**3**	**39**
LSL	TTSLL	3	3	6	48
LLS	TTSLL	3	3	6	48
Adversaire	Trace (tester J_2 et J_3)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	**SLL**	3	3	**0**	**30**
LSL	**LTSL**	4	2	**2**	**44**
LLS	**LTSL**	4	2	**2**	**44**
Adversaire	Trace (tester J_1 et J_3)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	**TSLL**	3	3	**3**	**39**
LSL	**TSLL**	3	3	**3**	**39**
LLS	**TLSL**	4	2	**3**	**47**
Adversaire	Trace (tester toutes les tâches)	$\|p_L\|$	$\|p_S\|$	$\|p_T\|$	$\sum C_j$
SLL	**TSLL**	3	3	**3**	**42**
LSL	**TTSLL**	3	3	**6**	**48**
LLS	**TTSLL**	3	3	**6**	**48**

Figure 5.8. : Cette figure représente les mêmes résultats que dans la Figure (5.7) précédente, mais dans le cadre adaptatif.

5.4 Perspectives futures : vers un minmax regret

En parallèle de cela, nous nous concentrons sur la recherche d'une politique de test qui minimise deux critères de robustesse couramment utilisés dans la littérature : le minmax regret absolu et le minmax regret relatif.

On se positionne dans le contexte d'un jeu opposant Alice et Bob pour une instance donnée (n_S,n_L,p_S,p_L,p_t). Alice joue le rôle d'un décideur et sélectionne une politique de tests Π, tandis que Bob, en tant qu'adversaire, doit choisir une permutation (notée s). L'objectif principal d'Alice est de minimiser la performance de Bob.

Selon la variante étudiée, Bob a l'un des deux objectifs suivants :

— Regret minmax absolu : Bob vise à maximiser la différence entre la performance de la politique de tests Π choisie par Alice pour la permutation s, notée $\Pi(s)$, et la meilleure performance qu'il aurait pu obtenir en utilisant une solution optimale offline appliquée à cette permutation s qu'on note $Off^*(s)$.

— Regret minmax relatif : Bob veut maximiser le rapport entre $\Pi(s)$ et $Off^*(s)$.

Donc notre objectif est alors de trouver des algorithmes pour ces deux critères : $\min_{\Pi}\max_s (\Pi(s) - Off^*(s))$ ainsi que $\min_{\Pi}\max_s \frac{\Pi(s)}{Off^*(s)}$.

Bien que le critère du min max Regret absolu se prête (encore une fois) très bien à une programmation dynamique, le min max regret relatif est quant à lui plus complexe. Cependant, les questions à propos du modèle offline à considérer (tel que présenté dans la Sous-Section (5.2.2)), ainsi que la recherche de règles de dominances pour le problème présenté précédemment (plus "simple") prennent alors tout leur sens.

6 Conclusion

Cette thèse s'est attachée à développer des algorithmes d'ordonnancement pour les systèmes distribués en tirant parti de techniques issues du domaine de la recherche opérationnelle, ainsi que de mécanismes mathématiques récents tels que la prédiction et l'augmentation des ressources. Les problèmes étudiés ont été abordés en prenant en compte des contraintes spécifiques en fonction des cas d'application, et des algorithmes adaptés ont été proposés pour fournir des solutions avec des garanties de performances.

Dans le premier volet de cette thèse, nous avons examiné l'utilisation d'un oracle qui permet de révéler, à la demande, certaines incertitudes. Nous avons proposé des algorithmes d'ordonnancement qui garantissent des performances optimales en termes de temps de complétion total, et nous avons évalué leur robustesse à travers des simulations. Par ailleurs, nous avons entamé des recherches préliminaires sur les performances de tels modèles dans le pire des cas.

Ensuite, nous nous sommes penchés sur un problème bi-agent où chaque agent est soumis à des contraintes et des objectifs différents. Nous avons proposé un algorithme utilisant l'augmentation des ressources, à la fois en termes de vitesse et de rejet, et nous avons prouvé, grâce à la technique du dual fitting, un rapport de compétitivité dépendant de ces paramètres d'augmentation. Bien que la solution apportée peut être considérée comme multicritère (via une approche ϵ-contrainte (Sous-Section 3.6.3 de [TB06]) si on prend les paramètres d'augmentation de ressources comme objectifs à part entière, l'optimisation du *makespan* apparait comme complètement indépendante des autres objectifs.

Enfin, nous avons abordé le problème plus classique de l'ordonnancement d'une application de tâches séquentielles sur une plateforme hybride CPU/GPU. Nous avons proposé un algorithme basé sur l'étude de l'écart d'intégralité d'une programmation linéaire. Nous avons montré que, malgré des résultats négatifs forts, choisir une allocation des tâches qui semble non optimale permet d'améliorer des rapports d'approximation des algorithmes d'ordonnancement ; en interprétant les parties fractionnaires d'une solution donnée par la relaxation d'un PLNE comme *un manque de confiance du LP envers tout ou partie de sa solution appliquée à "son" PLNE*.

Perspectives de recherche à court terme :

— Approfondir l'étude de l'utilisation d'oracle dans les pires cas afin d'identifier des règles de dominance permettant de concevoir des heuristiques gloutonnes plus économes en termes d'énergie, de temps de calcul et de mémoire.

— Étendre les travaux sur la plateforme hybride à $R|prec|C_{max}$, en explorant les possibilités d'adapter les approches utilisées pour $(P, P)|prec|C_{max}$ dans [Fag+20] et pour $Q|prec|C_{max}$ dans [CS97].

Perspectives de recherche à moyen et long terme :

— Explorer les combinaisons entre systèmes experts et intelligence artificielle (comme ici dans l'utilisation raisonnée d'oracles) ; par exemple en évaluant l'impact écologique de l'entraînement des réseaux de neurones en retirant les instances "simples" pouvant être traitées efficacement par des heuristiques.

— Considérer des modèles hybrides plus fréquemment, permettant de combiner des ressources matérielles de générations différentes afin de réutiliser des ressources fonctionnelles qui ont été remplacées en raison de leur manque de performances, tout en tenant compte de leur efficacité énergétique.

— Intégrer les critères énergétiques en tant que facteur clé dans la conception des algorithmes, en accordant une importance plus grande à l'énergie par rapport aux mesures traditionnelles telles que le *flowtime* ou le *makespan*.

— Promouvoir l'utilisation généralisée de l'augmentation des ressources, en développant des méthodes et des modèles pour exploiter cette approche de manière raisonnée et réaliste ainsi que rechercher de nouvelles approches d'augmentation.

Bibliographie

[AA12] Víctor AVELAR et Dan AZEVEDO. « PUE™ : A COMPREHENSIVE EXAMINATION OF THE METRIC ». In : 2012. cf. p. 1

[AE20] Susanne ALBERS et Alexander ECKL. « Explorable Uncertainty in Scheduling with Non-Uniform Testing Times ». In : *arXiv preprint arXiv :2009.13316* (2020). cf. p. 16

[AGK12] S ANAND, Naveen GARG et Amit KUMAR. « Resource augmentation for weighted flow-time explained by dual fitting ». In : *Proceedings of the 23rd ACM-SIAM symposium on Discrete Algorithms*. SIAM. 2012, p. 1228-1241. cf. p. 56, 63, 66

[Agn+04] Alessandro AGNETIS, Pitu B. MIRCHANDANI, Dario PACCIARELLI et Andrea PACIFICI. « Scheduling Problems with Two Competing Agents ». In : *Operations Research* 52.2 (2004), p. 229-242. cf. p. 6

[Agn+14] Alessandro AGNETIS, Jean-Charles BILLAUT, Stanislaw GAWIEJNOWICZ, Dario PACCIARELLI et Ameur SOUKHAL. *Multiagent Scheduling - Models and Algorithms*. Springer, 2014. cf. p. 6

[Agn+15] Alessandro AGNETIS, Gaia NICOSIA, Andrea PACIFICI et Ulrich PFERSCHY. « Scheduling two agent task chains with a central selection mechanism ». In : *J. Scheduling* 18.3 (2015), p. 243-261. cf. p. 6

[Ama+17] Marcos AMARIS, Giorgio LUCARELLI, Clément MOMMESSIN et Denis TRYSTRAM. « Generic Algorithms for Scheduling Applications on Hybrid Multi-core Machines ». In : *Euro-Par 2017 : Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*. T. 10417. Lecture Notes in Computer Science. 2017, p. 220-231. cf. p. 14, 73, 74, 76, 78, 80, 90

[And04] David P ANDERSON. « Boinc : A system for public-resource computing and storage ». In : *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society. 2004, p. 4-10. cf. p. 6

[Bam+21] Evripidis BAMPIS, Konstantinos DOGEAS, Alexander V. KONONOV, Giorgio LUCARELLI et Fanny PASCUAL. « Speed Scaling with Explorable Uncertainty ». In : *SPAA '21 : 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*. Sous la dir. de Kunal AGRAWAL et Yossi AZAR. ACM, 2021, p. 83-93. DOI : 10.1145/3409964.3461812. URL : https://doi.org/10.1145/3409964.3461812. cf. p. 5

[Bam+22] Evripidis BAMPIS, Konstantinos DOGEAS, Alexander V. KONONOV, Giorgio LUCARELLI et Fanny PASCUAL. « Scheduling with Untrusted Predictions ». In : *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*. Sous la dir. de Luc De RAEDT. ijcai.org, 2022, p. 4581-4587. DOI : 10.24963/ijcai.2022/636. URL : https://doi.org/10.24963/ijcai.2022/636. cf. p. 11

[Bea+12] Olivier BEAUMONT, Nicolas BONICHON, Lionel EYRAUD-DUBOIS et Loris MARCHAL. « Minimizing Weighted Mean Completion Time for Malleable Tasks Scheduling ». In : *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*. IEEE Computer Society, 2012, p. 273-284. DOI : 10.1109/IPDPS.2012.34. URL : https://doi.org/10.1109/IPDPS.2012.34. cf. p. 11

[BK09] Nikhil BANSAL et Subhash KHOT. « Optimal long code test with one free bit ». In : *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. IEEE. 2009, p. 453-462. cf. p. 76

[Ble+15] Raphaël BLEUSE, Safia KEDAD-SIDHOUM, Florence MONNA, Grégory MOUNIÉ et Denis TRYSTRAM. « Scheduling independent tasks on multi-cores with GPU accelerators ». In : *Concurrency and Computation : Practice and Experience* 27.6 (2015), p. 1625-1638. cf. p. 76

[BN15] Abbas BAZZI et Ashkan NOROUZI-FARD. « Towards Tight Lower Bounds for Scheduling Problems ». In : *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*. T. 9294. Lecture Notes in Computer Science. 2015, p. 118-129. cf. p. 14, 75, 76, 87

[Bru+05] Richard BRUCE, Michael HOFFMANN, Danny KRIZANC et Rajeev RAMAN. « Efficient update strategies for geometric computing with uncertainty ». In : *Theory of Computing Systems* 38.4 (2005), p. 411-423. cf. p. 16

[BS03] Kenneth R. BAKER et J. Cole SMITH. « A Multiple-Criterion Model for Machine Scheduling ». In : *J. Scheduling* 6.1 (2003), p. 7-16. cf. p. 6

[Can+18] Louis-Claude CANON, Loris MARCHAL, Bertrand SIMON et Frédéric VIVIEN. « Online Scheduling of Task Graphs on Hybrid Platforms ». In : *Euro-Par 2018 : Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*. T. 11014. Lecture Notes in Computer Science. 2018, p. 192-204. cf. p. 73, 76, 89, 90

[Can+20] Louis-Claude CANON, Loris MARCHAL, Bertrand SIMON et Frédéric VIVIEN. « Online Scheduling of Task Graphs on Heterogeneous Platforms ». In : *IEEE Transactions on Parallel and Distributed Systems* 31.3 (2020), p. 721-732. DOI : 10.1109/TPDS.2019.2942909. cf. p. 80

[CB98] Chandra CHEKURI et Michael A. BENDER. « An Efficient Approximation Algorithm for Minimizing Makespan on Uniformly Related Machines ». In : *Integer Programming and Combinatorial Optimization, 6th International IPCO Conference, Houston, Texas, USA, June 22-24, 1998, Proceedings*. T. 1412. Lecture Notes in Computer Science. 1998, p. 383-393. cf. p. 76

[Cha+02] Moses CHARIKAR, Ronald FAGIN, Venkatesan GURUSWAMI et al. « Query strategies for priced information ». In : *Journal of Computer and System Sciences* 64.4 (2002), p. 785-819. cf. p. 16

[Cha+21] Steven CHAPLICK, Magnús M HALLDÓRSSON, Murilo S de LIMA et Tigran TONOYAN. « Query minimization under stochastic uncertainty ». In : *Theoretical Computer Science* 895 (2021), p. 75-95. cf. p. 16

[Che+19] C. CHENG, S. LI, K. YING et Y. LIU. « Scheduling Jobs of Two Competing Agents on a Single Machine ». In : *IEEE Access* 7 (2019). cf. p. 6

[Cho+18] Anamitra Roy CHOUDHURY, Syamantak DAS, Naveen GARG et Amit KUMAR. « Rejecting jobs to minimize load and maximum flow-time ». In : *J. of Computer and System Sciences* 91 (2018), p. 42-68. cf. p. 4, 56

[CKZ01] Chandra CHEKURI, Sanjeev KHANNA et An ZHU. « Algorithms for minimizing weighted flow time ». In : *Proceedings of the 33rd ACM symposium on Theory of Computing*. 2001, p. 84-93. cf. p. 55

[CMM03] Richard W CONWAY, Louis W MILLER et William L MAXWELL. *Theory of scheduling*. Dover, 2003. cf. p. 15

[CS97] Fabián A. CHUDAK et David B. SHMOYS. « Approximation Algorithms for Precedence-Constrained Scheduling Problems on Parallel Machines That Run at Fifferent Speeds (Extended Abstract) ». In : *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana, USA*. 1997, p. 581-590. cf. p. 76, 90, 91, 110

[CYZ14] Lin CHEN, Deshi YE et Guochuan ZHANG. « Online Scheduling of mixed CPU-GPU jobs ». In : *Int. J. Found. Comput. Sci.* 25.6 (2014), p. 745-762. cf. p. 73, 76

[DLY90] Jianzhong DU, Joseph Y-T LEUNG et Gilbert H YOUNG. « Minimizing mean flow time with release time constraint ». In : *Theoretical Computer Science* 75.3 (1990), p. 347-355. cf. p. 9

[Duf+22] Fanny DUFOSSÉ, Christoph DÜRR, Noël NADAL, Denis TRYSTRAM et Óscar C. VÁSQUEZ. « Scheduling with a processing time oracle ». In : *Applied Mathematical Modelling* 104 (2022), p. 701-720. DOI : https://doi.org/10.1016/j.apm.2021.12.020. URL : https://www.sciencedirect.com/science/article/pii/S0307904X21005953. cf. p. 5, 12, 16

[Dür+18] Christoph DÜRR, Thomas ERLEBACH, Nicole MEGOW et Julie MEISSNER. « Scheduling with explorable uncertainty ». In : *9th Innovations in Theoretical Computer Science Conference (ITCS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018. cf. p. 5, 16, 17

[EH15] Thomas ERLEBACH et Michael HOFFMANN. « Query-competitive algorithms for computing with uncertainty ». In : *Bulletin of EATCS* 2.116 (2015). cf. p. 16

[Erl+08] Thomas ERLEBACH, Michael HOFFMANN, Danny KRIZANC, Matús MIHAL'ÁK et Rajeev RAMAN. « Computing minimum spanning trees with uncertainty ». In : *arXiv preprint arXiv :0802.2855* (2008). cf. p. 16

[Fag+20] Vincent FAGNON, Imed KACEM, Giorgio LUCARELLI et Bertrand SIMON. « Scheduling on Hybrid Platforms : Improved Approximability Window ». In : *LATIN 2020 : Theoretical Informatics - 14th Latin American Symposium, São Paulo, Brazil, January 5-8, 2021, Proceedings*. Sous la dir. d'Yoshiharu KOHAYAKAWA et Flávio Keidi MIYAZAWA. T. 12118. Lecture Notes in Computer Science. Springer, 2020, p. 38-49. DOI : 10.1007/978-3-030-61792-9_4. URL : https://doi.org/10.1007/978-3-030-61792-9%5C_4. cf. p. 73, 110

[Fag+22] Vincent FAGNON, Giorgio LUCARELLI, Clément MOMMESSIN et Denis TRYSTRAM. « Two-Agent Scheduling with Resource Augmentation on Multiple Machines ». In : *Euro-Par 2022 : Parallel Processing - 28th International Conference on Parallel and Distributed Computing, Glasgow, UK, August 22-26, 2022, Proceedings*. Sous la dir. de José CANO et Phil TRINDER. T. 13440. Lecture Notes in Computer Science. Springer, 2022, p. 253-267. DOI : 10.1007/978-3-031-12597-3_16. URL : https://doi.org/10.1007/978-3-031-12597-3%5C_16. cf. p. 55, 71

[Fed+00] Tomas FEDER, Rajeev MOTWANI, Rina PANIGRAHY, Chris OLSTON et Jennifer WIDOM. « Computing the median with uncertainty ». In : *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. 2000, p. 602-607. cf. p. 16

[Fed+07] Tomás FEDER, Rajeev MOTWANI, Liadan O'CALLAGHAN, Chris OLSTON et Rina PANIGRAHY. « Computing shortest paths with uncertainty ». In : *Journal of Algorithms* 62.1 (2007), p. 1-18. cf. p. 16

[Fen+15] Qi FENG, Baoqiang FAN, Shisheng LI et Weiping SHANG. « Two-agent scheduling with rejection on a single machine ». In : *Applied Mathematical Modelling* 39.3 (2015), p. 1183-1193. cf. p. 9

[Goe+15] Marc GOERIGK, Manoj GUPTA, Jonas IDE, Anita SCHÖBEL et Sandeep SEN. « The robust knapsack problem with queries ». In : *Computers & Operations Research* 55 (2015), p. 12-22. cf. p. 16

[Gra69] Ronald L. GRAHAM. « Bounds on Multiprocessing Timing Anomalies ». In : *SIAM Journal of Applied Mathematics* 17.2 (1969), p. 416-429. cf. p. 13, 58, 73, 76

[Hoc97] Dorit S. HOCHBAUM. « Approximation Algorithms for NP-Hard Problems ». In : *SIGACT News* 28.2 (1997), p. 40-52. DOI : 10.1145/261342.571216. URL : https://doi.org/10.1145/261342.571216. cf. p. 3

[Im+21] Sungjin IM, Ravi KUMAR, Mahshid Montazer QAEM et Manish PUROHIT. « Non-Clairvoyant Scheduling with Predictions ». In : *SPAA '21 : 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*. Sous la dir. de Kunal AGRAWAL et Yossi AZAR. ACM, 2021, p. 285-294. DOI : 10.1145/3409964.3461790. URL : https://doi.org/10.1145/3409964.3461790. cf. p. 11

[Kah91] Simon KAHAN. « A model for data in motion ». In : *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. 1991, p. 265-277. cf. p. 16

[KC03] Jae-Hoon KIM et Kyung-Yong CHWA. « Non-clairvoyant scheduling for weighted flow time ». In : *Inf. Process. Lett.* 87.1 (2003), p. 31-37. DOI : 10.1016/S0020-0190(03)00231-X. URL : https://doi.org/10.1016/S0020-0190(03)00231-X. cf. p. 11

[KMT15] Safia KEDAD-SIDHOUM, Florence MONNA et Denis TRYSTRAM. « Scheduling Tasks with Precedence Constraints on Hybrid Multi-core Machines ». In : *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. 2015, p. 27-33. cf. p. 14, 73, 74, 76, 78, 80, 90

[KP00] Bala KALYANASUNDARAM et Kirk PRUHS. « Speed is as powerful as clairvoyance ». In : *Journal of the ACM* 47.4 (2000), p. 617-643. cf. p. 4, 56

[KT01] Sanjeev KHANNA et Wang-Chiew TAN. « On computing functions with uncertainty ». In : *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2001, p. 171-182. cf. p. 16

[KTW99] Hans KELLERER, Thomas TAUTENHAHN et Gerhard WOEGINGER. « Approximability and nonapproximability results for minimizing total flow time on a single machine ». In : *SIAM Journal on Computing* 28.4 (1999), p. 1155-1166. cf. p. 55

[Kum+05] V. S. Anil KUMAR, Madhav V. MARATHE, Srinivasan PARTHASARATHY et Aravind SRINIVASAN. « Scheduling on Unrelated Machines Under Tree-Like Precedence Constraints ». In : *Approximation, Randomization and Combinatorial Optimization, Algorithms and Techniques, 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2005 and 9th InternationalWorkshop on Randomization and Computation, RANDOM 2005, Berkeley, CA, USA, August 22-24, 2005, Proceedings*. T. 3624. Lecture Notes in Computer Science. 2005, p. 146-157. cf. p. 76

[LCF16] Shi-Sheng LI, Ren-Xia CHEN et Qi FENG. « Scheduling two job families on a single machine with two competitive agents ». In : *J. Comb. Optim.* 32.3 (2016), p. 784-799. cf. p. 6

[LGL19] Peihai LIU, Manzhan GU et Ganggang LI. « Two-agent scheduling on a single machine with release dates ». In : *Computers and Operations Research* 111 (2019), p. 35-42. cf. p. 7

[Li17] Shi LI. « Scheduling to Minimize Total Weighted Completion Time via Time-Indexed Linear Programming Relaxations ». In : *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*. 2017, p. 283-294. cf. p. 76

[Lip68] John S. LIPTAY. « Structural Aspects of the System/360 Model 85 II : The Cache ». In : *IBM Syst. J.* 7 (1968), p. 15-21. cf. p. 5

[LM22] Alexander LINDERMAYR et Nicole MEGOW. « Non-Clairvoyant Scheduling with Predictions Revisited ». In : *CoRR* abs/2202.10199 (2022). arXiv : 2202.10199. URL : https://arxiv.org/abs/2202.10199. cf. p. 11

[LMS19] Retsef LEVI, Thomas MAGNANTI et Yaron SHAPOSHNIK. « Scheduling with testing ». In : *Management Science* 65.2 (2019), p. 776-793. cf. p. 16

[LR07] Stefano LEONARDI et Danny RAZ. « Approximating total flow time on parallel machines ». In : *J. of Computer and System Sciences* 73.6 (2007), p. 875-891. cf. p. 9, 55

[LR78] Jan Karel LENSTRA et AHG RINNOOY KAN. « Complexity of scheduling under precedence constraints ». In : *Operations Research* 26.1 (1978), p. 22-35. cf. p. 75

[Luc+16] Giorgio LUCARELLI, Nguyen Kim THANG, Abhinav SRIVASTAV et Denis TRYSTRAM. « Online Non-Preemptive Scheduling in a Resource Augmentation Model Based on Duality ». In : *24th Annual European Symposium on Algorithms, ESA*. T. 57. LIPIcs. 2016, 63 :1-63 :17. cf. p. 10

[Luc+18a] Giorgio LUCARELLI, Benjamin MOSELEY, Nguyen Kim THANG, Abhinav SRIVASTAV et Denis TRYSTRAM. « Online Non-preemptive Scheduling on Unrelated Machines with Rejections ». In : *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA*. ACM, 2018, p. 291-300. cf. p. 10, 72

[Luc+18b] Giorgio LUCARELLI, Benjamin MOSELEY, Nguyen Kim THANG, Abhinav SRIVASTAV et Denis TRYSTRAM. « Online Non-Preemptive Scheduling to Minimize Weighted Flow-time on Unrelated Machines ». In : *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*. Sous la dir. d'Yossi AZAR, Hannah BAST et Grzegorz HERMAN. T. 112. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 59 :1-59 :12. DOI : 10.4230/LIPIcs.ESA.2018.59. URL : https://doi.org/10.4230/LIPIcs.ESA.2018.59. cf. p. 10

[Luc+19] Giorgio LUCARELLI, Benjamin MOSELEY, Nguyen Kim THANG, Abhinav SRIVASTAV et Denis TRYSTRAM. « Online Non-Preemptive Scheduling to Minimize Maximum Weighted Flow-Time on Related Machines ». In : *39th IARCS Annual Conference on FSTTCS*. T. 150. LIPIcs. 2019, 24 :1-24 :12. cf. p. 10

[LV18] Thodoris LYKOURIS et Sergei VASSILVITSKII. « Competitive Caching with Machine Learned Advice ». In : *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Sous la dir. de Jennifer G. DY et Andreas KRAUSE. T. 80. Proceedings of Machine Learning Research. PMLR, 2018, p. 3302-3311. URL : http://proceedings.mlr.press/v80/lykouris18a.html. cf. p. 11

[MMS17] Nicole MEGOW, Julie MEISSNER et Martin SKUTELLA. « Randomization helps computing a minimum spanning tree under uncertainty ». In : *SIAM Journal on Computing* 46.4 (2017), p. 1217-1240. cf. p. 16

[MNS07] Mohammad MAHDIAN, Hamid NAZERZADEH et Amin SABERI. « Allocating online advertisement space with unreliable estimates ». In : *Proceedings 8th ACM Conference on Electronic Commerce (EC-2007), San Diego, California, USA, June 11-15, 2007*. Sous la dir. de Jeffrey K. MACKIE-MASON, David C. PARKES et Paul RESNICK. ACM, 2007, p. 288-294. DOI : 10.1145/1250910.1250952. URL : https://doi.org/10.1145/1250910.1250952. cf. p. 11

[Mon14] Florence MONNA. « Scheduling for new computing platforms with GPUs. (Ordonnancement pour les nouvelles plateformes de calcul avec GPUs) ». Thèse de doct. Pierre et Marie Curie University, Paris, France, 2014. URL : https://tel.archives-ouvertes.fr/tel-01127919. cf. p. 80

[Ngu+21] Dinh C NGUYEN, Ming DING, Pubudu N PATHIRANA et al. « Federated learning for internet of things : A comprehensive survey ». In : *IEEE Communications Surveys & Tutorials* (2021). cf. p. 1

[Phi+97] Cynthia A PHILLIPS, Cliff STEIN, Eric TORNG et Joel WEIN. « Optimal time-critical scheduling via resource augmentation ». In : *Proceedings of the 29th ACM symposium on Theory of Computing*. 1997, p. 140-149. cf. p. 4, 56

[Pin12] Michael PINEDO. *Scheduling*. T. 29. Springer, 2012. cf. p. 15, 19, 20

[PSK18] Manish PUROHIT, Zoya SVITKINA et Ravi KUMAR. « Improving Online Algorithms via ML Predictions ». In : *Advances in Neural Information Processing Systems*. Sous la dir. de S. BENGIO, H. WALLACH, H. LAROCHELLE et al. T. 31. Curran Associates, Inc., 2018. URL : https://proceedings.neurips.cc/paper_files/paper/2018/file/73a427badebe0e32caa2e1fc7530b7f3-Paper.pdf. cf. p. 11

[PSR21] Lucas PEROTIN, Hongyang SUN et Padma RAGHAVAN. « Multi-Resource List Scheduling of Moldable Parallel Jobs under Precedence Constraints ». In : *ICPP 2021 : 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*. Sous la dir. de Xian-He SUN, Sameer SHENDE, Laxmikant V. KALÉ et Yong CHEN. ACM, 2021, 23 :1-23 :10. DOI : 10.1145/3472456.3472487. URL : https://doi.org/10.1145/3472456.3472487. cf. p. 77

[Rou20] Tim ROUGHGARDEN, éd. *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 2020. DOI : 10.1017/9781108637435. URL : https://doi.org/10.1017/9781108637435. cf. p. 4

[Smi56] Wayne E. SMITH. « Various optimizers for single-stage production ». In : *Naval Research Logistics Quarterly* 3 (1956), p. 59-66. cf. p. 9, 11

[ST09] Erik SAULE et Denis TRYSTRAM. « Multi-users scheduling in parallel systems ». In : *23rd IEEE International Symposium on Parallel and Distributed Processing, (IPDPS)*. 2009, p. 1-9. cf. p. 7

[Sve11] Ola SVENSSON. « Hardness of Precedence Constrained Scheduling on Identical Machines ». In : *SIAM J. Comput.* 40.5 (2011), p. 1258-1274. cf. p. 75, 76, 87

[TB06] Vincent T'KINDT et Jean-Charles BILLAUT. *Multicriteria Scheduling - Theory, Models and Algorithms (2. ed.)* Springer, 2006. DOI : 10.1007/b106275. URL : https://doi.org/10.1007/b106275. cf. p. 109

[YWC20] Yunqiang YIN, Dujuan WANG et T.C.Edwin CHENG. *Due Date-Related Scheduling with Two Agents - Models and Algorithms*. Springer, 2020. cf. p. 6

Détails du lemme 2.3.2

A

Lemme A.0.1. *Si l'algorithme optimal non-adaptatif décide de tester des tâches, alors ces tâches seront les premières de la permutation.*

Esquisse de la preuve Nous démontrons par un argument d'échange que : en supposant qu'il soit pertinent de tester une tâche J_i après avoir exécuté une tâche J_{i-1} sans la tester ; alors il est préférable de tester d'abord la tâche J_i, puis d'exécuter ensuite sans la tester la tâche J_{i-1}.
On pourra ensuite déduire directement de cette règle de dominance que dans une solution optimale, si une tâche J_k est testée, alors toutes les tâches J_i (avec $i \leq k$) le sont aussi.

Preuve.

Supposons l'existence d'un ordonnancement de n tâches, pour lequel n'est pas pas testés la $(i-1)^{\text{ème}}$ tâche. On rejoue cet ordonnancement post-mortem, et on se positionne au moment où on s'intéresse à la tâche J_{i-1}, qu'on notera a. A cet instant précis, on pourra noter k le nombre de tâches ayant déjà été exécutées, le nombre de tâches longues ayant été détectées vaut donc $i-2-k$.

Pour simplifier les démonstrations à venir, nous utilisons la notation suivante pour définir la valeur de la fonction objectif résultante de l'ordonnancement tel que décrit dans la Figure "x" :

$$\sum C_i$$

Figure 2.x

Tout d'abord, nous étudions l'impact sur l'ordonnancement d'avoir pris la décision de tester la tâche J_i (ordonnancement représenté avec nos notations Figure 2.4), ou d'avoir pris la décision de ne pas tester J_i (représenté Figure 2.3).

Ajoutons quelques notations supplémentaires. Soient :

— $F = \sum_{\forall j \in \text{ les k tâches executées}} C_j$

— $a = \Big(\text{nombre de tests faits sur les } (i-2)^{\text{ières}} \text{ tâches}\Big) \cdot p_T + \sum_{\forall j \in \text{ les tâches k executées}} p_j$

— d : le nombre de tests qui seront effectués sur les (n-i) tâches inconnues

— b : la date de début des tâches encore inconnues

— c : la date de fin des tâches détectées longues

— $G = \sum_{\forall j \in \text{ les n-i tâches inconnues}} (C_j - b) + \sum_{\forall j \in \text{ i-2-k tâches détéctées longues}} (C_j - b)$
intuitivement, G correspond au flowtime total des tâches exécutées entre c et b si on considère $b = 0$.

Avec ces définitions supplémentaires, on peut donc évaluer évaluer la valeur de l'objectif $\sum C_j$ sur les ordonnancements décrits par ces trois figures.

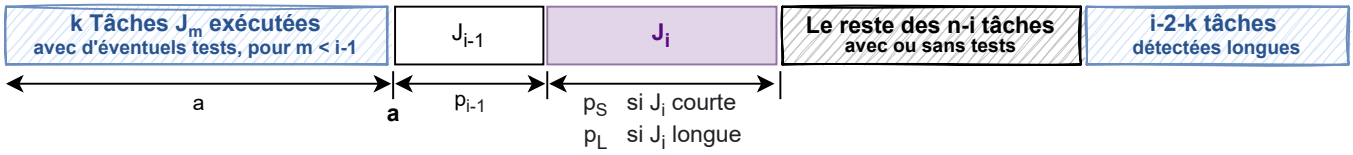


Figure A.1. : Un ordonnancement si on décide de ne tester ni la tâche J_{i-1}, ni la tâche J_i.

$$\sum \text{Figure A.1} \; C_j = F + a(n-k) + p_{i-1} + p_{i-1}(n-k-1) + p_i + p_i(n-k-2) + G$$

$$= F + a(n-k) + \left(p_S \frac{n_S}{n} + p_L \frac{n_L}{n}\right)(n-k) + \left(p_S \frac{n_S}{n} + p_L \frac{n_L}{n}\right)(n-k-1) + G$$

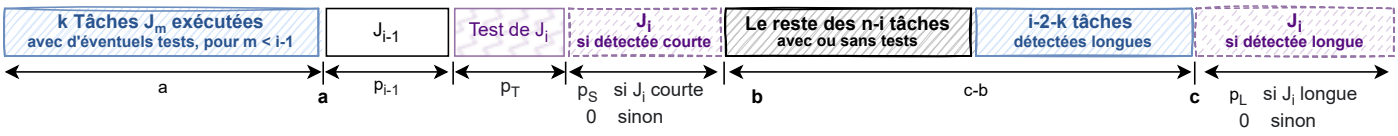


Figure A.2. : Le même ordonnancement mais on décide d'exécuter la tâche J_{i-1} sans la tester puis de tester la tâche J_i.

$$\sum \text{Figure A.2} \; C_j = F + a(n-k) + p_{i-1}(n-k) + p_T(n-k-1)$$

$$+ \mathbb{P}(\text{i} \in \mathcal{S})(p_S(n-k-1) + G) + \mathbb{P}(\text{i} \in \mathcal{L})(G + (c - b + p_L))$$

$$= F + a(n-k) + G + (\frac{n_S}{n} p_S + \frac{n_L}{n} p_L)(n-k) + p_T(n-k-1) + \frac{n_S}{n} p_S(n-k-1)$$

$$+ \frac{n_L}{n}\left((n-i)\left(p_S \frac{n_S}{n-1} + p_L \frac{n_L - 1}{n-1}\right) + d p_T + (i-2-k)p_L + p_L\right)$$

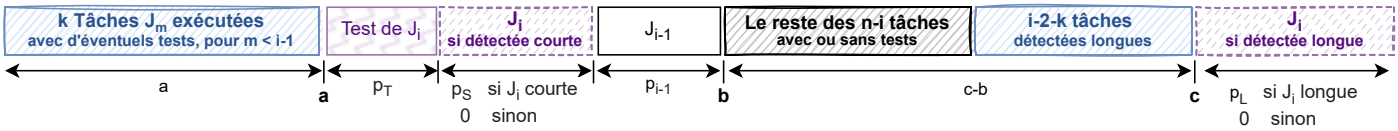


Figure A.3. : Le même ordonnancement mais cette fois, on décide d'inverser J_{i-1} et J_i, c'est à dire tout d'abord de tester la tâche J_i, puis ensuite d'exécuter sans la tester la tâche J_{i-1}.

$$\begin{aligned}\sum_{\text{Figure A.3}} C_j =& F + a(n-k) + p_T(n-k) + \mathbb{P}(i\in\mathcal{S})\,(p_S(n-k) + p_{i-1}(n-k-1) + G)\\ &+ \mathbb{P}(i\in\mathcal{L})\,(p_{i-1}(n-k) + G + (c-b) + p_L)\\ =& F + a(n-k) + p_T(n-k) + G\\ &+ \frac{n_S}{n}\left(p_S(n-k) + \left(\frac{n_S-1}{n-1}p_S + \frac{n_L}{n-1}p_L\right)(n-k-1)\right)\\ &+ \frac{n_L}{n}\left(\left(\frac{n_S}{n-1}p_S + \frac{n_L-1}{n-1}p_L\right)(n-k)\right.\\ &\qquad\left.+(n-i)(\frac{n_S}{n-1}p_S + \frac{n_L-1}{n-1}p_L) + dp_T + (i-2-k)p_L + p_L\right)\end{aligned}$$

Tester J_i sachant que J_{i-1} ne l'est pas réduit $\sum C_j \Leftrightarrow \sum_{\text{Figure A.2}} C_j - \sum_{\text{Figure A.1}} C_j \leq 0$

$$\Leftrightarrow p_T(n-k-1+d\frac{n_L}{n}) + p_S(n-i)\frac{n_S n_L}{n(n-1)} - p_L\frac{n_S n_L}{n(n-1)}(n-i) \leq 0 \qquad \text{(A.1)}$$

$$\Rightarrow p_T(n-k-1+d\frac{n_L}{n}) \leq \Delta\frac{n_S n_L}{n(n-1)}(n-i) \qquad \text{(A.2)}$$

Tester J_i avant d'executer J_{i-1} réduit $\sum C_j \Leftrightarrow \sum_{\text{Figure A.3}} C_j - \sum_{\text{Figure A.2}} C_j \leq 0$

$$\sum_{\text{Figure A.3}} C_j - \sum_{\text{Figure A.2}} C_j = p_T - \frac{n_S}{n}\frac{n_L}{n-1}\Delta \qquad \text{(A.3)}$$

Si on admet (A.1), alors on peut utiliser l'inégalité (A.2) sur (A.3) :

$$\begin{aligned}\sum_{\text{Figure A.3}} C_j - \sum_{\text{Figure A.2}} C_j &\leq \Delta\frac{n_S n_L}{n(n-1)}\left(\frac{n-i}{(n-k-1+d\frac{n_L}{n})} - 1\right)\\ &\leq \Delta\frac{n_S n_L}{n(n-1)}\frac{(-i+k+1-d\frac{n_L}{n})}{(n-k-1+d\frac{n_L}{n})} \leq 0 \qquad \text{(A.4)}\end{aligned}$$

(A.4) est correcte car $k \leq i-2$ et $n \geq k+2$. Donc si on veut tester J_i, il est préférable de l'échanger avec sa tâche précédente (J_{i-1}) si celle ci n'est pas testée. On peut en déduire récursivement que si on décide de tester m tâches (et d'exécuter sans tester les $n-m$ restantes), le meilleur cout moyen est obtenu en testant les m premières.

□

Détails Fractional Flowtime

B

On s'intéresse ici à une formulation de borne inférieure (à un facteur près) du flowtime. Comme cela concerne les tâches locales, il faut tenir compte du fait qu'elles puissent être préemptées.

En supposant que la tâche j ne soit pas préemptée on a :

$$\begin{aligned}
\int_{r_j}^{\infty} \left(\frac{(t-r_j)}{p_j} + \Gamma \right) x_j(t)dt &= 0 + \left[\frac{t^2}{2p_j} \right]_{S_j}^{S_j+p_j} + \left[\frac{\Gamma \cdot p_j - r_j}{p_j} \cdot t \right]_{S_j}^{S_j+p_j} + 0 \\
&= \frac{(S_j+p_j)^2 - S_j^2}{2p_j} + \frac{(\Gamma \cdot p_j - r_j)p_j}{p_j} \\
&= S_j + \frac{p_j}{2} - r_j + \Gamma \cdot p_j \\
&= \text{Flowtime}_j + (\Gamma - \frac{1}{2})p_j \\
&\leq (\Gamma + \frac{1}{2})\text{Flowtime}_j \qquad \text{(B.1)}
\end{aligned}$$

Si la tâche j peut être préemptée, alors $x_j(t) = 1$ "par morceaux". On s'intéresse à une seule tâche j dans nos formules, on va donc s'épargner la notation mathématique supplémentaire superflue $_j$. La durée de la tâche est notée p et sa date d'arrivée dans le système r.

Cependant on utilise la notation $_{(n)}$ pour indiquer qu'on parle d'une tâche est découpée en n morceaux, chacun de durée p_i et séparée du précédent par un idle time a_i (on note a_1 le temps d'attente de la tâche avant son démarrage). La durée p est donc égale à $\sum_{i=1}^{n} p_i$.

Comme $\Gamma \geq \frac{1}{2}$, il s'agit de montrer que $\int_{r_j}^{\infty} \left(\frac{(t-r_j)}{p_j} + \frac{1}{2} \right) x_j(t)dt$ est une borne inférieure du flowtime ; la transformation du $\frac{1}{2}$ en Γ pouvant être facilement faite via cette innappoximation B.1.

Lemme B.0.1. *Pour tout n ≥ 2 : la différence entre l'évaluation du terme* $\int_{r_j}^{\infty}\left(\frac{(t-r_j)}{p_j}+\frac{1}{2}\right)x_j(t)$ *pour une tâche séparée en n "morceaux" avec l'évaluation de cette même tâche pour laquelle on aurait réunit le* $n^{ième}$ *et le (n-1)*^{ième} *pour les executer sans interruption est* $\frac{a_n \cdot p_n}{p}$.
C'est à dire : $\int_r^{\infty}\left(\frac{(t-r)}{p}+\frac{1}{2}\right)x_j(t)_{(n)}dt = \int_r^{\infty}\left(\frac{(t-r)}{p}+\frac{1}{2}\right)x_j(t)_{(n-1)}dt + \frac{a_n \cdot p_n}{p}$

Preuve. Pour n = 1 :
$\int_r^{\infty}\left(\frac{(t-r)}{p}+\frac{1}{2}\right)x_j(t)_{(1)}dt = \left[\frac{t^2}{2p}\right]_{S_1}^{S_1+p} + \left[\frac{p-2r}{2p}t\right]_{S_1}^{S_1+p}$
$= \frac{2S_1 \cdot p+(p)^2}{2p} + \frac{p-2r}{2p}p = S_1 + p - r = Flowtime_{(1)}$

Pour n = 2 :
$\int_r^{\infty}\left(\frac{(t-r)}{p}+\frac{1}{2}\right)x_j(t)_{(2)}dt$
$= \left[\frac{t^2}{2p}\right]_{S_1}^{S_1+p_1} + \left[\frac{p-2r}{2p}t\right]_{S_1}^{S_1+p_1} + \left[\frac{t^2}{2p}\right]_{S_2}^{S_2+p_2} + \left[\frac{p-2r}{2p}t\right]_{S_2}^{S_2+p_2}$
$= \frac{2S_1p_1+p_1^2}{2p} + \frac{p-2r}{2p}p_1 + \frac{2S_2p_2+p_2^2}{2p} + \frac{p-2r}{2p}p_2$
$= \frac{2S_1p_1+p_1^2+2S_2p_2+p_2^2}{2p} + \frac{p-2r}{2p}(p_1+p_2)$ avec $S_2 = S_1 + p_1 + a_2$
$= \frac{2S_1p_1+p_1^2+2(S_1+p_1+a_2)p_2+p_2^2}{2p} + \frac{p-2r}{2p}(p_1+p_2)$
$= \frac{2S_1(p_1+p_2)+(p_1+p_2)^2+2a_2p_2}{2p} + \frac{p-2r}{2p}(p_1+p_2)$
$= \frac{2S_1(p)+(p)^2+2a_2p_2}{2p} + \frac{p-2r}{2p}p$
$= (\frac{2S_1 \cdot p+(p)^2}{2p} + \frac{p-2r}{2p}p) + \frac{a_2p_2}{p}$
$= \int_r^{\infty}\left(\frac{(t-r)}{p}+\frac{1}{2}\right)x_j(t)_{(1)} + \frac{a_2p_2}{p}$
and $Flowtime_{(2)} = Flowtime_{(1)} + a_2$
d'où $\int_r^{\infty}\left(\frac{(t-r)}{p}+\frac{1}{2}\right)x_j(t)_{(2)}dt$
$= Flowtime_{(1)} + \frac{a_2p_2}{p} = Flowtime_{(2)} - a_2 + \frac{a_2p_2}{p} = Flowtime_{(2)} - a_2\frac{p-p_2}{p}$

Au rang n-1, on note la durée du dernier processing time p_a

$\int_r^\infty \left(\frac{(t-r)}{p} + \frac{1}{2}\right) x_j(t)_{(n-1)} dt$

$= \sum_{i=0}^{n-1} \left(\left[\frac{t^2}{2p}\right]_{S_i}^{S_i+p_i} + \left[\frac{p-2r}{2p} t\right]_{S_i}^{S_i+p_i} \right)$

$= \sum_{i=0}^{n-2} \left(\left[\frac{t^2}{2p}\right]_{S_i}^{S_i+p_i} + \left[\frac{p-2r}{2p} t\right]_{S_i}^{S_i+p_i} \right) + \left[\frac{t^2}{2p}\right]_{S_{n-1}}^{S_{n-1}+p_a} + \left[\frac{p-2r}{2p} t\right]_{S_{n-1}}^{S_{n-1}+p_a}$

$= \sum_{i=0}^{n-2} \left(\left[\frac{t^2}{2p}\right]_{S_i}^{S_i+p_i} + \left[\frac{p-2r}{2p} t\right]_{S_i}^{S_i+p_i} \right) + \frac{2S_{n-1}p_a + p_a^2}{2p} + \frac{p-2r}{2p} p_a$

Au rang n, p_a devient p_{n-1} et p_n séparées d'un idle a_n avec ($p_a = p_{n-1} + p_n$)

$\int_r^\infty \left(\frac{(t-r)}{p} + \frac{1}{2}\right) x_j(t)_{(n)} dt$

$= \sum_{i=0}^{n} \left(\left[\frac{t^2}{2p}\right]_{S_i}^{S_i+p_i} + \left[\frac{p-2r}{2p} t\right]_{S_i}^{S_i+p_i} \right)$

$= \sum_{i=0}^{n-2} \left(\left[\frac{t^2}{2p}\right]_{S_i}^{S_i+p_i} + \left[\frac{p-2r}{2p} t\right]_{S_i}^{S_i+p_i} \right) + \left[\frac{t^2}{2p}\right]_{S_{n-1}}^{S_{n-1}+p_{n-1}} + \left[\frac{p-2r}{2p} t\right]_{S_{n-1}}^{S_{n-1}+p_{n-1}}$
$+ \left[\frac{t^2}{2p}\right]_{S_n}^{S_n+p_n} + \left[\frac{p-2r}{2p} t\right]_{S_n}^{S_n+p_n}$

$= \sum_{i=0}^{n-2} \left(\left[\frac{t^2}{2p}\right]_{S_i}^{S_i+p_i} + \left[\frac{p-2r}{2p} t\right]_{S_i}^{S_i+p_i} \right) + \frac{2S_{n-1}p_{n-1} + p_{n-1}^2}{2p} + \frac{p-2r}{2p} p_{n-1} + \frac{2S_n p_n + p_n^2}{2p} + \frac{p-2r}{2p} p_n$

D'où $\int_r^\infty \left(\frac{(t-r)}{p} + \frac{1}{2}\right) x_j(t)_{(n)} dt - \int_r^\infty \left(\frac{(t-r)}{p} + \frac{1}{2}\right) x_j(t)_{(n-1)} dt$

$= \frac{2S_{n-1}p_{n-1} + p_{n-1}^2}{2p} + \frac{p-2r}{2p} p_{n-1} + \frac{2S_n p_n + p_n^2}{2p} + \frac{p-2r}{2p} p_n - \left(\frac{2S_{n-1}p_a + p_a^2}{2p} + \frac{p-2r}{2p} p_a\right)$

$= \frac{2S_{n-1}p_{n-1} + p_{n-1}^2 + 2S_n p_n + p_n^2}{2p} + \frac{p-2r}{2p}(p_{n-1} + p_n) - \left(\frac{2S_{n-1}p_a + p_a^2}{2p} + \frac{p-2r}{2p} p_a\right)$

avec $S_n = S_{n-1} + p_{n-1} + a_n$

$= \frac{(p_{n-1}^2 + 2p_{n-1}p_n + p_n^2) + 2S_{n-1}(p_{n-1} + p_n) + 2a_n p_n - 2S_{n-1}p_a - p_a^2)}{2p} + \frac{p-2r}{2p}(p_{n-1} + p_n - p_a)$

$= \frac{a_n p_n}{p}$

L'hypothèse est vraie au rang 2, et reste vraie en passant du rang n au rang n+1.□

On en déduit que $\int_r^\infty \left(\frac{(t-r)}{p} + \frac{1}{2}\right) x_j(t)_{(n)} dt$

$= \int_r^\infty \left(\frac{(t-r)}{p} + \frac{1}{2}\right) x_j(t)_{(1)} dt + \frac{1}{p} \sum_{i=2}^{n} \left(a_i \sum_{k=i}^{n} p_k\right)$

$= \frac{1}{p} \sum_{i=2}^{n} \left(a_i \sum_{k=i}^{n} p_k\right) + Flowtime_{(1)}$

$= \frac{1}{p} \sum_{i=2}^{n} \left(a_i \sum_{k=i}^{n} p_k\right) + Flowtime_{(n)} - \sum_{i=2}^{n} a_i = Flowtime_{(n)} - \frac{1}{p} \sum_{i=2}^{n} \left(a_i \sum_{k=1}^{i} p_k\right)$

Pour tout ordonnancement fait avec la règle SRPT : $\sum_{k=i}^{n} p_k$, a_i, et p sont toujours supérieurs à la (aux) tâche(x) qu'on intercale dans l'idle. Cette quantification de l'écart entre nos formules et le flowtime pourrait être utilisée dans le choix de nos variables duales pour pouvoir proposer un meilleur rapport de compétitivité.

C

Détails de calculs du théorème 4.3.3

Lemme C.0.1. *Il est possible de définir b en fonction de m et k de façon de trouver le rapport d'approximation suivant :*

$$\frac{C^A_{max}}{C^*_{max}} \leq 3 + 2\sqrt{2 - \frac{k}{m} - \frac{k^2}{m^2} - \frac{1}{m}\left(1 - \frac{k}{m}\right)} - \frac{1}{m} \leq 3 + 2\sqrt{2} \approx 5.83$$

Preuve. Il s'agit de définir b (≥ 2) tel que la formule (4.7) soit minimale, c'est à dire trouver b qui minimise r(b), avec :

$$\frac{C^A_{max}}{C^*_{max}} \leq b + 2 \cdot \frac{b}{b-1} - \frac{k}{m}(b - \frac{b}{b-1}) - \frac{1}{m}(\frac{b}{b-1}) = r(b)$$

Dans le cas simple où $m = k$, pour b tend vers $+\infty$, on a

$$r(b) = \left(3 - \frac{1}{m}\right)\frac{b}{b-1}$$
$$\lim_{b\to+\infty} r(b) = \left(3 - \frac{1}{m}\right) = 3 + 2\sqrt{2 - \frac{k}{m} - \frac{k^2}{m^2} - \frac{1}{m}\left(1 - \frac{k}{m}\right)} - \frac{1}{m}$$

A présent, si on considère $m \neq k$ (donc $m > k$), on s'intéresse à la dérivée de r(b) :

$$\begin{aligned} r'(b) &= 1 - \frac{2}{(b-1)^2} - \frac{k}{m}\left(1 + \frac{1}{(b-1)^2}\right) + \frac{1}{m}\frac{1}{(b-1)^2} \\ &= \frac{b^2(1-\frac{k}{m}) - 2b(1-\frac{k}{m}) - 1 - 2\frac{k}{m} + \frac{1}{m}}{(b-1)^2} \\ &= \left(1 - \frac{k}{m}\right)\frac{(b - S_1)\,(b - S_2)}{(b-1)(b-1)} \end{aligned}$$

Avec $S_1 = 1 + \sqrt{\frac{(2 + \frac{k}{m} - \frac{1}{m})}{1 - \frac{k}{m}}}$; $\quad S_2 = 1 - \sqrt{\frac{(2 + \frac{k}{m} - \frac{1}{m})}{1 - \frac{k}{m}}}$; $\quad$ de plus $S_2 \leq 2 \leq S_1$

Nous ne nous intéressons qu'aux cas où $b \geq 2$: r'(b) est négative pour $2 \leq b \leq S_1$, puis positive pour $S_1 \leq b$, donc r(b) est décroissante puis croissante, et est donc minimale pour $b = S_1$.

d'où $\frac{C^A_{max}}{C^*_{max}}$

$$
\begin{aligned}
\leq & b + 2 \cdot \frac{b}{b-1} - \frac{k}{m}(b - \frac{b}{b-1}) - \frac{1}{m}(\frac{b}{b-1}) \\
= & \left(1 + \sqrt{\frac{(2 + \frac{k}{m} - \frac{1}{m})}{1 - \frac{k}{m}}}\right) + 2 \cdot \left(\sqrt{\frac{1 - \frac{k}{m}}{2 + \frac{k}{m} - \frac{1}{m}}} + 1\right) \\
& - \frac{k}{m}\left(\left(1 + \sqrt{\frac{2 + \frac{k}{m} - \frac{1}{m}}{1 - \frac{k}{m}}}\right) - \left(\sqrt{\frac{1 - \frac{k}{m}}{2 + \frac{k}{m} - \frac{1}{m}}} + 1\right)\right) - \frac{1}{m}\left(\sqrt{\frac{1 - \frac{k}{m}}{2 + \frac{k}{m} - \frac{1}{m}}} + 1\right) \\
= & 3 + \frac{\sqrt{2 + \frac{k}{m} - \frac{1}{m}}\sqrt{2 + \frac{k}{m} - \frac{1}{m}}}{\sqrt{2 + \frac{k}{m} - \frac{1}{m}}\sqrt{1 - \frac{k}{m}}} + \frac{2\sqrt{1 - \frac{k}{m}}\sqrt{1 - \frac{k}{m}}}{\sqrt{1 - \frac{k}{m}}\sqrt{2 + \frac{k}{m} - \frac{1}{m}}} \\
& - \frac{k}{m}\left(\frac{\sqrt{2 + \frac{k}{m} - \frac{1}{m}}\sqrt{2 + \frac{k}{m} - \frac{1}{m}}}{\sqrt{2 + \frac{k}{m} - \frac{1}{m}}\sqrt{1 - \frac{k}{m}}} - \frac{\sqrt{1 - \frac{k}{m}}\sqrt{1 - \frac{k}{m}}}{\sqrt{1 - \frac{k}{m}}\sqrt{2 + \frac{k}{m} - \frac{1}{m}}}\right) \\
& - \frac{1}{m}\left(\frac{\sqrt{1 - \frac{k}{m}}\sqrt{1 - \frac{k}{m}}}{\sqrt{1 - \frac{k}{m}}\sqrt{2 + \frac{k}{m} - \frac{1}{m}}} + 1\right) \\
= & 3 + \frac{\left(2 + \frac{k}{m} - \frac{1}{m} + 2(1 - \frac{k}{m})\right)}{\sqrt{2 + \frac{k}{m} - \frac{1}{m}}\sqrt{1 - \frac{k}{m}}} - \frac{k}{m}\left(\frac{\left(2 + \frac{k}{m} - \frac{1}{m}\right) - \left(1 - \frac{k}{m}\right)}{\sqrt{2 + \frac{k}{m} - \frac{1}{m}}\sqrt{1 - \frac{k}{m}}}\right) \\
& - \frac{1}{m}\left(\frac{1 - \frac{k}{m}}{\sqrt{1 - \frac{k}{m}}\sqrt{2 + \frac{k}{m} - \frac{1}{m}}} + 1\right) \\
= & 3 + \frac{\left(4 - \frac{k}{m} - \frac{1}{m}\right) - \frac{k}{m}\left(1 + 2\frac{k}{m} - \frac{1}{m}\right) - \frac{1}{m}\left(1 - \frac{k}{m}\right)}{\sqrt{2 + \frac{k}{m} - \frac{1}{m}}\sqrt{1 - \frac{k}{m}}} - \frac{1}{m} \\
= & 3 + \frac{4 - 2\frac{k}{m} - \frac{2}{m} - 2\frac{k^2}{m^2} + \frac{2}{m}\frac{k}{m}}{\sqrt{\left(2 + \frac{k}{m} - \frac{1}{m}\right) - \frac{k}{m}\left(2 + \frac{k}{m} - \frac{1}{m}\right)}} - \frac{1}{m} \\
= & 3 + \frac{4 - 2\frac{k}{m} - \frac{2}{m} - 2\frac{k^2}{m^2} + \frac{2}{m}\frac{k}{m}}{\sqrt{2 - \frac{k}{m} - \frac{1}{m} - \frac{k^2}{m^2} + \frac{k}{m}\frac{1}{m}}} - \frac{1}{m} \\
= & 3 + 2\sqrt{2 - \frac{k}{m} - \frac{k^2}{m^2} - \frac{1}{m}\left(1 - \frac{k}{m}\right)} - \frac{1}{m}
\end{aligned}
$$

□

D

Valeur de C_{max} selon l'instance du problème q-parti

Lemme D.0.1 (Completness). *Si G_q correspond au cas OUI du problème q-parti, alors l'instance $\mathcal{I}$ admet un planning avec un makespan de $(q+3)Q/3$.*

Preuve.

Supposons que G_q correspond à une instance OUI du problème q-parti et soient $V_{i,j}$ (pour $1 \leq i \leq q$ et $j < Q$) la partition associée des ensembles V_i. Notons que la taille de n'importe quel ensemble $V_{i,j}$ de la partition est au plus de $(1+Q\varepsilon)n/Q$, car $\sum_{j=0}^{Q-1} |V_{i,j}| = |V_i| = n$ et, par définition, dans un cas OUI il est vrai que $|V_{i,j}| \geq \frac{1-\varepsilon}{Q}n$. Nous partitionnons ensuite les tâches de $\mathcal{I}$ en ensembles $S_{i,j}$. Pour chaque z, $0 \leq z < q/3$, et j, $0 \leq j \leq Q-1$, nous définissons :

- type A : $S_{zQ+1,j} = \bigcup_{v \in V_{3z+1,j}} \mathcal{J}_{3z+1,v}$, donc $|S_{zQ+1,j}| \leq \frac{(Qn-Q)(1+Q\varepsilon)n}{Q} \leq \frac{kn-k}{n}$.
- type B : $S_{zQ+2,j} = \bigcup_{v \in V_{3z+2,j}} \mathcal{J}_{3z+2,v}$, donc $|S_{zQ+2,j}| \leq Qn^3(1+Q\varepsilon)n/Q \leq (1+Q\varepsilon)n^4 \leq m$.
- type C : pour $1 \leq \ell \leq Q-2$, $S_{zQ+2+\ell,j} = \bigcup_{v \in V_{3z+3,j}} J^{\ell}_{3z+3,v}$, donc $|S_{zQ+2+\ell,j}| = (1+Q\varepsilon)n/Q \leq k/(nQ)$.

Soit $\mathcal{T}_t$ l'union de tous les $S_{i,j}$ avec $t = i + j$, $1 \leq i \leq Qq/3$ et $0 \leq j \leq Q-1$. Nous créons un ordonnancement pour l'instance $\mathcal{I}$ de la manière suivante : au créneau horaire $[t-1, t)$, nous exécutons les tâches de l'ensemble $\mathcal{T}_t$. Une esquisse du début de ce planning est donnée dans le tableau D.1. Le type et le nombre de machines (CPU ou GPU) pour exécuter chaque ensemble de tâches $S_{i,j}$ est également donné dans ce tableau. Notez que les tâches du second triplet $\langle V_4, V_5, V_6 \rangle$ commencent leur exécution à partir du créneau $[Q, Q+1)$: plus précisément, $S_{Q+1,0}$ contient des tâches de V_4. De plus, l'exécution de certaines tâches du premier triplet $\langle V_1, V_2, V_3 \rangle$ a lieu après le temps $Q+1$: plus précisément, les dernières tâches de ce triplet appartiennent à l'ensemble $S_{Q,Q-1}$ et elles sont exécutées au créneau $[2Q-2, 2Q-1)$. Cependant, il n'y a pas aucun créneau dans lequel sont impliqués 3 triplets.

Dans la dernière plage horaire de l'ordonnancement ainsi créé, nous exécutons les tâches dans $\mathcal{T}_t$ avec $t = i + j$, $i = Qq/3$ et $j = Q-1$. Ainsi, le makespan vaut $Qq/3 + Q - 1 < Qq/3 + Q$. Il reste à prouver la faisabilité de l'ordonnancement créé :

	CPU	GPU							
	m	$k(1-1/n)$	$k/(nQ)$	$k/(nQ)$	$k/(nQ)$	$\dots$	$k/(nQ)$	$k/(nQ)$	$k/(nQ)$
$[0,1)$		$S_{1,0}$							
$[1,2)$	$S_{2,0}$	$S_{1,1}$							
$[2,3)$	$S_{2,1}$	$S_{1,2}$	$S_{3,0}$						
$[3,4)$	$S_{2,2}$	$S_{1,3}$	$S_{4,0}$	$S_{3,1}$					
$[4,5)$	$S_{2,3}$	$S_{1,4}$	$S_{5,0}$	$S_{4,1}$	$S_{3,2}$				
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$				
$[Q-1,Q)$	$S_{2,Q-2}$	$S_{1,Q-1}$	$S_{Q,0}$	$S_{Q-1,1}$	$S_{Q-2,2}$	$\dots$	$S_{3,Q-3}$		
$[Q,Q+1)$	$S_{2,Q-1}$	$S_{Q+1,0}$		$S_{Q,1}$	$S_{Q-1,2}$	$\dots$	$S_{4,Q-3}$	$S_{3,Q-2}$	
$[Q+1,Q+2)$	$S_{Q+2,0}$	$S_{Q+1,1}$			$S_{Q,2}$	$\dots$	$S_{5,Q-3}$	$S_{4,Q-2}$	$S_{3,Q-1}$
$[Q+2,Q+3)$	$S_{Q+2,1}$	$S_{Q+1,2}$				$\dots$	$S_{6,Q-3}$	$S_{5,Q-2}$	$S_{4,Q-1}$
$[Q+3,Q+4)$	$S_{Q+2,2}$	$S_{Q+1,3}$	$S_{Q+3,0}$			$\dots$	$S_{7,Q-3}$	$S_{6,Q-2}$	$S_{5,Q-1}$
$[Q+4,Q+5)$	$S_{Q+2,3}$	$S_{Q+1,4}$	$S_{Q+4,0}$	$S_{Q+3,1}$		$\dots$	$S_{8,Q-3}$	$S_{7,Q-2}$	$S_{6,Q-1}$
$\dots$									

Table D.1. : Une esquisse du début du planning pour les taches $\in \mathcal{I}$.

les contraintes de précédence sont satisfaites et il y a suffisamment de machines pour effectuer les tâches assignées à chaque plage horaire.

Considérons d'abord les contraintes de précédence à l'intérieur de chaque ensemble $\mathcal{J}_{3z+3,v}$, $0 \leq z < q/3$ et $v \in V_{3z+3}$, c'est-à-dire l'arc de la tâche $J^{\ell}_{3z+3,v}$ à la tâche $J^{\ell+1}_{3z+3,v}$, pour tout ℓ, $1 \leq \ell \leq Q-3$. Par construction, $J^{\ell}_{3z+3,v} \in S_{zQ+2+\ell,j}$ et $J^{\ell+1}_{3z+3,v} \in S_{zQ+2+\ell+1,j}$. Ainsi, $J^{\ell}_{3z+3,v}$ est exécuté dans la plage horaire $zQ+2+\ell+j$, tandis que $J^{\ell+1}_{3z+3,v}$ dans la plage horaire $zQ+2+\ell+1+j > zQ+2+\ell+j$. Par conséquent, ce type de contraintes de précédence sont satisfaites.

Considérons maintenant la contrainte de précédence d'une tâche $J \in \mathcal{J}_{i,v}$ correspondant à $v \in V_{i,j_1} \subset V_i$ à une tâche $J' \in \mathcal{J}_{i+1,w}$ correspondant à $w \in V_{i+1,j_2} \subset V_{i+1}$. Par construction et en raison du fait que G_q est une instance OUI, un arc de J à J' n'existe que si $j_1 \leq j_2$. Supposons que J appartienne à l'ensemble S_{i_1,j_1}, tandis que J' appartient à l'ensemble S_{i_2,j_2}. Par définition des ensembles $S_{i,j}$, nous avons que $i_1 < i_2$. Ainsi, $i_1 + j_1 < i_2 + j_2$, ce qui signifie que J est exécuté dans un créneau avant J', et par conséquent, ces contraintes de précédence sont également satisfaites.

Il reste à montrer que chaque ensemble $\mathcal{T}_t$ est composé d'au plus m tâches CPU et k tâches GPU, de sorte qu'il puisse être calculé sur un seul créneau. Dans un ensemble donné $\mathcal{T}_t$, il peut y avoir au plus un ensemble de type A, un ensemble de type B et $Q-2$ ensembles de type C. Comme expliqué dans la définition des ensembles $S_{i,j}$, chaque ensemble de type B est composé d'au plus m tâches CPU. De plus, chaque ensemble de type A est composé d'au plus $k(1-1/n)$ tâches GPU, tandis que chacun des $Q-2$ ensembles de type C est composé d'au plus k/nQ tâches GPU. En total, il y a $k(1-1/n)+(Q-2)k/nQ < k$ tâches GPU, ce qui conclut la preuve du lemme. □

Lemme D.0.2 (Soundness)**.** *Si G_q correspond au cas NON du problème q-partite, alors l'ordonnancement obtenu à partir de l'instance $\mathcal{I}$ a un makespan d'au moins $f(Q)qQ$, où f tend vers 1 lorsque Q croît.*

Preuve. Supposons que G_q correspond à une instance NON du problème q-partite, et considérons la partition suivante des tâches de l'instance associée $\mathcal{I}$, pour tout $0 \leq z < q/3$:

— type A : $S_{Qz+1} := \bigcup_{v \in V_{3z+1}} \mathcal{J}_{3z+1,v}$, donc $|S_{Qz+1}| = Qn^2 - Qn = n(n-1)Q$.
— type B : $S_{Qz+2} := \bigcup_{v \in V_{3z+2}} \mathcal{J}_{3z+2,v}$, donc $|S_{Qz+1}| = Qn^4$.
— type C : $S_{Qz+2+\ell} := \bigcup_{v \in V_{3z+3}} J^{\ell}_{3z+3,v}$, pour $1 \leq \ell \leq Q-2$, donc $|S_{Qz+2+\ell}| = n$.

Considérons un ordonnancement de $\mathcal{I}$ qui minimise le makespan et éliminons une fraction 2δ de chaque ensemble S_i de la partition où $(i \mod Q) \in 0, 1, 2, 3$: les premières $\lceil \delta |S_i| \rceil$ tâches à être exécutées sont S_i^s et les dernières $\lceil \delta |S_i| \rceil$ tâches à exécuter sont S_i^f. Soit $\mathcal{R}$ le pseudo-ordonnancement obtenu.

Supposons qu'il existe i tel qu'une tâche de S_{i+1} est commencée avant que toutes les tâches de S_i soient terminées, et au moins un ensemble parmi S_i, S_{i+1} est de type A ou B (*i.e.*, $(i \mod Q) \in 0, 1, 2$). Alors, cela signifie qu'il n'y a pas d'arc entre S_i^f et S_{i+1}^s. Soit i' tel que l'ensemble S_i correspond à des sommets de $V_{i'}$, et soit $V_{i'}^f$ l'ensemble de sommets $v \in V_{i'}$ qui vérifie $\mathcal{J}_{i',v} \cup S_i^f \neq \emptyset$. Définissons $V_{i'+1}^s$ de manière analogue. Par la définition des ensembles $\mathcal{J}$, il n'y a pas d'arc entre $V_{i'}^f$ et $V_{i'+1}^s$.

Comme tous les $\mathcal{J}_{i',v}$ ont la même taille, on a $|V_{i'}^f| \geq |V_{i'}| \cdot |S_i^f|/|S_i| \geq \lfloor \delta n \rfloor$, et de même, $|V_{i'+1}^s| \geq \lfloor \delta n \rfloor$. Cela contredit l'hypothèse selon laquelle G_q est une instance NO du problème q-partite. Par conséquent, dans le pseudo-ordonnancement $\mathcal{R}$, une tâche d'un ensemble S_i de type A ou B ne peut pas être exécutée simultanément avec une tâche des ensembles S_{i-1} ou S_{i+1}.

Par conséquent, pour tout z, l'ensemble S_{Qz+1} de type A doit être terminé avant le début de l'ensemble S_{Qz+2} de type B, qui doit lui-même être terminé avant le début de l'ensemble S_{Qz+3} de type C. Si $z < q/3 - 1$, l'ensemble $S_{Q(z+1)}$ de type C doit à son tour être terminé avant le début de l'ensemble $S_{Q(z+1)+1}$ de type A.

Fixons z et considérons les $Q-2$ ensembles de type C associés à V_{3z}. Parmi les $n(Q-2)$ tâches de ces ensembles, au plus $2(2\delta n+2) < n = |V_{3z}|$ ont été abandonnées, donc il existe un sommet $v \in V_{3z}$ pour lequel aucune des tâches $J^{\ell}_{3z,v}$, pour $1 \leq \ell \leq Q-2$, n'a été abandonnée. Parce que ces tâches forment une chaîne et que chaque tâche nécessite un temps 1 pour être terminée, le pseudo-ordonnancement

R a besoin d'au moins un temps $M_C = Q - 2$ pour planifier tous les ensembles de type C associés à V_{3z}.

Dans la pseudo-ordonnancement $\mathcal{R}$, si toutes les tâches d'un ensemble S_{Qz+1} de type A sont exécutées sur des GPU, cela prend un temps d'au moins (rappelons que $n > Q$, $\varepsilon \leq 1/Q^2$ et $\delta \leq 1/(2Q)$) :

$$\begin{aligned} M_A &= \frac{|S_{Qz+1}| - |S^s_{Qz+1}| - |S^f_{Qz+1}|}{k} \\ &\geq \frac{(1-2\delta)n(n-1)Q - 2}{(1+Q\varepsilon)n^2 + 1} \\ &\geq \frac{(1-\frac{1}{Q})(1-\frac{1}{n}) - \frac{2}{Qn^2}}{(1+\frac{1}{Q}) + \frac{1}{n^2}}\, Q \\ &\geq \frac{Q-2}{Q+2}\, Q. \end{aligned}$$

Si une tâche d'un ensemble A est exécutée sur un CPU, cela prend un temps $n > Q \geq M_A$.

Un ensemble S_{Qz+2} de type B doit être ordonnancé sur tous les CPU et GPU en un temps d'au moins :

$$\begin{aligned} M_B &= \frac{|S_{Qz+2}| - |S^s_{Qz+2}| - |S^f_{Qz+2}|}{m + n \cdot k} \\ &\geq \frac{(1-2\delta)Qn^4 - 2}{(1+Q\varepsilon)n^4 + 1 + n((1+Q\varepsilon)n^2 + 1)} \\ &\geq \frac{1 - \frac{1}{Q} - \frac{2}{Qn^4}}{1 + \frac{1}{Q} + \frac{1}{n^4} + \frac{1}{n} + \frac{1}{nQ} + \frac{1}{n^3}}\, Q \\ &\geq \frac{Q-2}{Q+3}\, Q. \end{aligned}$$

Par conséquent, le makespan de $\mathcal{R}$ est d'au moins :

$$\frac{q}{3}\left(M_A + M_B + M_C\right) \geq \frac{qQ}{3}\ \left(\frac{Q-2}{Q+2} + \frac{Q-2}{Q+3} + \frac{Q-2}{Q}\right).$$

Comme l'expression entre parenthèses tend vers 3 lorsque Q augmente, et que le makespan de $\mathcal{R}$ n'est pas supérieur au makespan minimum pour ordonnancer l'instance $\mathcal{I}$, le lemme est prouvé.

□