View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by OpenGrey Repository

UNIVERSITÉ d'ÉVRY-VAL d'ESSONNE

École Doctorale S&I (Sciences et Ingénierie)

THÈSE

présentée par :

Fadi Kacem

pour obtenir :

Spécialité : Informatique

Algorithmes Exacts et Approchés pour des problèmes d'Ordonnancement et de Placement

Thèse soutenue le 27 juin 2012 devant le jury composé de :

Rapporteurs :	M. Denis	Professeur à l'ENSIMAG, Grenoble
	Mme. Johanne	Chargée de Recherche CNRS au PRiSM, Versailles
Examinateurs :	M. Christoph	Directeur de Recherche CNRS au LIP6, Paris
	M. Jean-Marc	Professeur à l'UEVE, Evry
Directeur de thèse :	M. Evripidis	Professeur à l'UPMC, Paris
Co-encadrant :	M. Eric	Professeur à l'UEVE, Evry

Thèse préparée au sein de l'équipe OPAL du Laboratoire IBISC

EA 4526 - Université d'Évry-Val d'Essonne

Table des matières

Liste des tableaux **vi**

Table des figures **viii**

Introduction **1**

1 Ordonnancer pour économiser l'énergie : cas de plusieurs machines identiques **9**

1.1 Introduction 9

1.1.1 Contribution et organisation du chapitre 13

1.2 Définition du problème 13

1.3 Critères d'optimalité 14

1.3.1 Un programme convexe 14

1.3.2 Les conditions KKT 16

1.4 Un algorithme combinatoire basé sur le calcul de flots 23

1.4.1 Le problème de flot-max/coupe-min 23

1.4.2 Le problème d'affectation à vitesse constante "*AVC*" 25

1.4.3 L'algorithme *SSM* 32

1.5 Conclusion 40

2 Ordonnancer pour économiser l'énergie : cas d'une seule machine **43**

2.1 Introduction 43

2.2 Définition du problème 44

2.3 Travaux connexes 48

2.4 Structure de l'ordonnancement optimal 49

2.5 Suffixes et Préfixes 52

2.6 Le programme dynamique 56

2.7 Analyse de la complexité 58

2.8 Conclusion 62

3 Ordonnancer pour économiser l'énergie : cas de plusieurs machines hétérogènes 63

3.1 Introduction 63

3.2 Travaux connexes et contributions 64

3.3 Relaxation linéaire 67

3.4 Algorithme randomisé pour le problème $RS \mid r_{ij} | E + \sum w_j C_j$ 70

3.4.1 Dérandomisation 77

3.5 Extension au problème $RS | r_{ij}, E | \sum w_j C_j$ 82

3.6 Conclusion 85

4 Ordonnancement avec des contraintes de précédence et des temps de communication 87

4.1 Introduction 87

4.2 Définitions et Notations 90

4.2.1 Formulation du modèle 90

4.2.2 Temps de communication monotones 91

4.2.3 Algorithmes de liste 92

4.3 Travaux connexes et contributions 93

4.4 Complexité du problème général 95

4.5 Extension de l'algorithme *LPP* sur des machines dédiées 97

4.5.1 Une borne supérieure sur les temps de complétude des tâches 98

4.5.2 Modification des dates d'échéance 100

4.5.3 Optimalité de l'extension de l'algorithme *LPP* 104

4.6 Non optimalité des algorithmes de liste pour des machines identiques 106

4.7 Conclusion 107

5 Algorithmes optimaux pour le problème de placement de données 111

5.1 Introduction 111

5.2 Définition du problème 112

5.3 Travaux connexes et contributions 113

5.4 Algorithme de programmation dynamique 114

5.4.1 Placement de données sur une chaîne 114

5.4.2 Placement de données sur un anneau 116

5.4.3 Placement de données sur un graphe en étoile généralisée 120

5.5 Placement de données sur un graphe quelconque 121

5.6 Conclusion 125

Conclusion 127

Bibliographie 131

Liste des tableaux

3.1 Exemple d'application de l'algorithme 71

4.1 Temps de communication monotones pour le graphe d'ordre d'intervalles 92

4.2 Dates d'arrivée et d'échéance et types des tâches pour la figure 4.4 99

4.3 Dates d'arrivée, dates d'échéance et dates de début d'exécution d'une solution . . 99

4.4 Dates d'arrivée $r^\star$ améliorées et une nouvelle numérotation des tâches 101

4.5 Dates d'échéance améliorées $d_k^\star(i), (i, k) \in \{1, \ldots, 8\}^2$. 102

Table des figures

1.1 Influence du choix des vitesses sur l'énergie consommée. 11

1.2 Exemple d'ordonnancement sans préemption durant un intervalle I_j sur 3 machines. 19

1.3 Réduction de l'instance $(\mathcal{J}, \mathcal{I}, v)$ du problème *AVC* en un problème de flot max . 26

1.4 Exemple d'instance du problème de minimisation d'énergie avec migration. . . . 34

1.5 Exemple d'application de l'algorithme *SSM*. 36

2.1 Ordonnancement d'une instance composée de 5 tâches 46

2.2 Exemple d'ordonnancement sans augmentation d'énergie 47

2.3 Structure d'un ordonnancement optimal pour une instance composée de 11 tâches 54

2.4 Les différents événements durant la procédure de compression 61

3.1 L'ensemble des intervalles géométriques déduits de l'instance du problème . . . 71

3.2 Solution optimale du programme linéaire appliqué sur l'instance du problème . . 72

3.3 Solution retournée suite à l'exécution de l'algorithme sur l'instance . 72

3.4 Solution retournée suite à l'exécution de l'algorithme modifié 79

4.1 Ordonnancement d'un graphe de précédence avec 4 tâches unitaires 88

4.2 Ordonnancement d'un graphe de précédence avec des temps de communication . 88

4.3 Ordonnancement d'un graphe de précédence avec 3 tâches unitaires typées 89

4.4 Un graphe d'ordre d'intervalles et les intervalles correspondants 90

4.5 Transformation d'une instance du problème SCL en une instance du problème SBC 97

4.6 Extension du *LPP* pour l'exemple de la figure 4.4 avec des échéances modifiées . 104

4.7 Les algorithmes de liste ne sont pas optimaux pour 2 machines identiques 107

4.8 L'algorithme *LPP* risque de retourner un ordonnancement non réalisable 108

5.1 Problème de placement de données sur une chaîne. 115

5.2 Problème de placement de données sur une topologie d'anneau. 118

5.3 Problème de placement de données sur une topologie en étoile généralisée 120

Introduction

L'optimisation combinatoire est une branche de l'informatique dans laquelle on cherche à obtenir des solutions efficaces pour des problèmes discrets. Ces problèmes sont, généralement, caractérisés par un ensemble de contraintes que doit satisfaire toute solution réalisable et une fonction, dite fonction objectif, qui définit la notion de meilleure solution. En effet, pour chaque solution réalisable, la fonction objectif renvoie un entier ou un réel et la meilleure solution (ou solution optimale) est celle qui minimise ou maximise la fonction objectif. Il est, toutefois, possible qu'un problème d'optimisation combinatoire admette plusieurs solutions optimales.

Par ailleurs, un algorithme est dit efficace si son temps d'exécution est polynomial en la taille de l'instance, *i.e.* le nombre d'opérations élémentaires requises par l'algorithme est polynomial.

Il est ainsi possible de classifier les différents problèmes d'optimisation en problèmes simples et difficiles selon l'existence ou non d'algorithmes polynomiaux pouvant les résoudre. Stephen Cook [36] a formalisé cette classification en introduisant les classes de complexité *P* et *NP*, et la notion de *NP-difficulté*. La classe de complexité *P* contient tous les problèmes pour lesquels il existe des algorithmes polynomiaux pouvant les résoudre. Par ailleurs, un problème appartient à la classe de complexité *NP* s'il existe un algorithme polynomial capable de valider toute solution du problème de décision correspondant. Il est clair que tous les problèmes dans *P* sont aussi dans *NP*, et on a donc l'inclusion suivante : $P \subseteq NP$.

Maintenant, un problème d'optimisation est qualifié de *NP-difficile* s'il est au moins aussi difficile que tous les problèmes de la classe *NP*[1]. Dans ce cas où on ne peut pas espérer trouver un

[1] On utilise la notion de la *Réduction polynomiale* pour montrer qu'un problème Π est au moins aussi difficile que tous les problèmes d'une classe de complexité (la classe *NP* par exemple).

algorithme polynomial pour le résoudre, sauf si *P=NP*. Cook [36] et Karp [53] ont prouvé qu'un large éventail de problèmes d'optimisation naturels sont *NP-difficile* et depuis, la complexité de la plupart des problèmes d'optimisation combinatoire a été déterminée, *i.e.* soit dans *P*, ou *NP-difficile*. Cependant, il est à noter que la plupart des problèmes pratiques sont *NP-difficiles*.

Dans l'impossibilité d'une résolution optimale d'un problème en temps polynomial, on a souvent recours à des solutions approchées qui sont fournies par des algorithmes d'approximation. Une stratégie classique pour obtenir de tels algorithmes consiste à relacher quelques contraintes du problème de façon à obtenir une relaxation du problème d'origine pour laquelle on est capable de construire une solution optimale en un temps polynomial en la taille de l'instance. Il faut donc ensuite tranformer la solution obtenue de la relaxation en une solution réalisable du problème original. La solution optimale du problème original n'étant pas connue, toute la difficulté réside dans l'analyse visant à garantir que la solution obtenue à partir de l'approximation ne s'éloigne pas trop de l'optimum original.

Dans ce contexte, nous nous intéressons à la résolution de quelques problèmes d'optimisation combinatoires que nous avons choisi de traiter en deux volets. Dans un premier temps, nous étudions des problèmes d'optimisation issus de l'ordonnancement d'un ensemble de tâches sur des machines de calcul et où on cherche à minimiser l'énergie totale consommée par ces machines tout en préservant une qualité de service acceptable. Dans un deuxième temps, nous traitons deux problèmes d'optimisation classiques à savoir un problème d'ordonnancement dans une architecture de machines parallèles avec des temps de communication, et un problème de placement de données dans des graphes modélisant des réseaux pair-à-pair et visant à minimiser le coût total d'accès aux données.

Ordonnancement avec minimisation de l'énergie

Bien que l'informatique occupe aujourd'hui une place prépondérante dans notre vie dans le sens où elle facilite la gestion de nombreuses activités quotidiennes, il n'est cependant plus possible de négliger les conséquences indésirables qui en résultent. La consommation d'énergie

par les appareils informatiques fait partie des plus importants des effets secondaires. En effet, selon une étude menée en 2007 par l'agence de protection de l'environnement des États-Unis *"EPA"*, on a prédit qu'en 2010, environ 2% de l'énergie produite dans le monde sera consommée par les centres de données contre 1% en 2005 et 0.5% en 2000. Une grande partie de cette énergie est consommée alors que les serveurs sont inactifs et attendent sans rien faire. Si des grands efforts, aussi bien du point de vue matériel que logiciel, ont été effectués dans la conception des appareils portatifs (téléphones, ordinateurs portables, etc.) pour offrir une grande autonomie énergétique et donc un gaspillage d'énergie aussi faible que possible, peu de choses ont été faites pour la minimisation de l'énergie consommée par les serveurs informatiques et encore moins pour les centres de données. Dans ce contexte, on peut citer les propos d'Eric Schmidt, directeur exécutif de Google, qui dit : *" What matters most to the computer designers at Google is not speed but power - low power, because data centers can consume as much as a city "*.

Néanmoins, étant donné l'impact néfaste sur l'environnement ainsi que le coût important de l'énergie, de plus en plus d'acteurs prennent conscience du problème. Dans un rapport qui date d'Août 2011, Jonathan G. Koomey [56] a montré que contrairement aux prédictions de l'*EPA*, la consommation mondiale en énergie dans les centres de données n'a pas doublé en 2010 par rapport à 2005 mais elle a plutôt affiché un taux d'environ 1.3% de la consommation mondiale. Ce ralentissement dans le rythme de croissance de la consommation d'énergie est dû, en grande partie à des politiques et mécanismes de maîtrise de l'énergie dans les centres de données. D'un point de vue algorithmique, ceci inclut l'étude de problèmes d'ordonnancement ayant comme objectif la minimisation de l'énergie consommée par les processeurs. Cette consommation d'énergie est causée principalement par l'exécution des différentes tâches qui sont soumises aux machines. Cependant, une partie non négligeable de la dépense énergétique provient aussi de l'inactivité des machines puisque même en l'absence de programmes ou requêtes à exécuter, une machine dépense une quantité constante d'énergie que nous appelons énergie statique (ou énergie de base). Nous considérons, donc, deux modèles pour caractériser cette consommation d'énergie.

Le premier modèle connu sous le nom d'*Adaptation des vitesses* (*Speed Scaling* en anglais) considère des machines capables de modifier leurs vitesses d'exécution des tâches d'une façon dynamique et à tout moment. L'idée consiste à adapter la vitesse de chaque machine suivant le profil des tâches qui lui sont soumises de façon à préserver une certaine qualité de service requise. Cette adaptation des vitesses influe considérablement sur les niveaux de consommation d'énergie étant donnée que, dans le cas général, moins la vitesse d'une machine est importante moins d'énergie sera consommée.

Le deuxième modèle que nous considérons est connu sous le nom de *Gestion des états de veille* (*Power Down Management* en anglais). Dans ce modèle, les machines tournent à des vitesses constantes et donc les durées des tâches sont à leur tour constantes. Cependant, chaque machine est caractérisée par un ou plusieurs états de veille dans lesquels elle peut transiter dans le cas où l'ensemble des tâches à exécuter devient vide. Durant ces états de veille, les machines passent à des niveaux de consommation d'énergie bas, voire nuls. Par contre, une quantité d'énergie fixe est nécessaire si une machine passe dans un état actif pour reprendre l'exécution des tâches. Dans ce cas, la question est de déterminer à quels moments il serait intéressant de changer les états des machines et comment ordonnancer les tâches pour profiter au mieux de ces états de basse consommation. Une idée intuitive consiste à minimiser le nombre de passages entre états actifs et états de veille et maximiser la durée des intervalles où le système est au repos.

Il s'agit donc de concevoir des algorithmes qui offrent des compromis entre la qualité de services demandée par les utilisateurs et l'énergie consommée par les machines.

Des problèmes d'optimisation classiques

Comme nous l'avons mentionné précédemment, nous nous intéressons à deux problèmes classiques d'optimisation combinatoire.

Le premier problème est un problème d'ordonnancement de tâches sur des machines parallèles non identiques, et en présence de contraintes de précédence et de temps de communication. En effet, l'étude de ce type de problèmes présente un intêret dans la mesure où les architectures avec

plusieurs machines ou entités de calcul ne cessent de se développer.

Au cours de l'exécution d'un programme sur une telle architecture, dite parallèle, il est nécessaire de prévoir des délais supplémentaires entre les tâches dépendantes qui s'exécutent sur des machines différentes. Ces délais de communication sont nécessaires pour transférer les résultats fournis par une tâche à une autre tâche dépendante afin que cette dernière puisse commencer son exécution sur une machine différente. L'exécution de ces tâches sur la même machine implique l'absence des temps de communication du fait que ceux-ci deviennent négligeable comparés aux durées d'exécution des tâches.

Notons, par ailleurs, que dans les calculs parallèles, une grande partie de la complexité est due à la communication entre les machines. Il s'agit, donc, de traiter un problème d'ordonnancement avec des contraintes classiques de dépendance entre tâches, auxquelles s'ajoutent des délais supplémentaires appelé communément *temps de communication*. Dans le monde informatique, ce type de problème est souvent rencontré lorsqu'il est question de modéliser l'exécution d'applications sur un réseau de processeurs tel que les grilles de calcul.

Le deuxième problème est connu sous le nom du problème de *placement de données* et consiste à répliquer un ensemble de fichiers sur des machines connectées par un réseau de façon à minimiser le coût d'accès des différents utilisateurs aux différentes données demandées. Ce type de problème est posé essentiellement dans le contexte des réseaux de partage distribués à grande échelle tels que les réseaux pair-à-pair. L'intérêt majeur de la réplication des données dans ce type de réseau réside dans l'amélioration de la qualité de service caractérisée par un accès rapide aux données en plus de la fiabilité et de la tolérance aux fautes du service. Par conséquent, la consommation de la bande passante au niveau des serveurs contenant les données originales diminue d'une façon considérable.

Organisation de la thèse

Cette thèse est divisée en deux parties indépendantes. La première partie s'étend sur les trois premiers chapitres et traite des problèmes d'ordonnancement avec minimisation de l'énergie

consommée. La deuxième partie considère des problèmes d'optimisation classiques : un problème d'ordonnancement sur des machines non identiques avec des contraintes de précédence et des temps de communication, et un problème de placement de données dans un réseau pair-à-pair.

Dans le chapitre 1, nous considérons le modèle d'adaptation des vitesses (*Speed Scaling*) et nous traitons le problème d'ordonnancement d'un ensemble de tâches sur des machines parallèles et identiques. Nous supposons que la préemption et la migration des tâches sont autorisées, c'est à dire que l'exécution d'une tâche peut être interrompue et continuée plus tard sur la même ou une autre machine. Nous supposons aussi que chaque tâche possède une date d'arrivée et une date d'échéance. L'objectif consiste donc à minimiser l'énergie totale consommée par toutes les machines tout en respectant les contraintes d'échéances des tâches.

Nous proposons un algorithme polynomial capable de résoudre ce problème d'une façon optimale. Notre approche consiste à déduire la structure de toute solution optimale à l'aide d'une formulation convexe du problème, et de le réduire en un problème de recherche de flots maximums dans un graphe construit à partir d'une instance du problème original. Notons que d'une manière indépendante Albers et al. [5] ont également proposé un algorithme optimal pour ce problème.

Dans le chapitre 2, nous étudions un cas plus élaboré où le modèle d'adaptation des vitesses (*Speed Scaling*) et le modèle de gestion des états de veille (*Power Down Management*) sont considérés à la fois. Dans ce chapitre, nous considérons une seule machine pouvant changer dynamiquement sa vitesse d'exécution et pouvant aussi passer dans un état de veille où l'énergie consommée est nulle. Le problème consiste à ordonnancer un ensemble de tâches avec des dates d'arrivée et d'échéance *agréables*[2] dans le but de minimiser l'énergie consommée par la machine en question. Nous proposons ainsi une solution optimale basée sur la programmation dynamique. Notons que Albers et al. [4] ont prouvé récemment que le problème général est NP-difficile.

[2]Dans un problème d'ordonnancement, une instance est dite agréable si pour chaque paire de tâches (i, j) avec des dates d'arrivée (*resp.* d'échéance) r_i et r_j (*resp.* d_i et d_j), $r_i \leqslant r_j$ si et seulement si $d_i \leqslant d_j$.

Dans le chapitre 3, le problème étudié consiste à ordonnancer un ensemble de tâches sur des machines parallèles non uniformes sans préemption ni migration. Dans le modèle que l'on considère, chaque machine peut modifier sa vitesse qui sera sélectionnée parmi un ensemble fini de vitesses possibles et chaque tâche est caractérisée par une pondération qui indique son importance, une date d'arrivée qui dépend de la machine à laquelle elle est affectée et une durée d'exécution et une quantité d'énergie qui sont fonctions à la fois de la machine d'affectation et de la vitesse d'exécution de la tâche. Par ailleurs, nous supposons que chaque tâche est exécutée à une vitesse constante. Le but est, donc, la minimisation de la somme des temps de complétude pondérés plus l'énergie totale consommée par les machines. Ce problème étant *NP-difficile*, nous présentons une extension d'un algorithme d'approximation proposé par Andreas Schulz et Martin Skutella pour résoudre le problème $R|r_j|\sum w_jC_j$ [75]. La technique utilisée repose sur l'arrondi aléatoire d'une solution optimale d'une relaxation linéaire du problème et elle permet de construire une solution approchée. De plus, nous utilisons la même technique pour approcher le problème de minimisation de la somme des temps de complétude pondérés avec un budget d'énergie fixe à ne pas dépasser.

Dans le chapitre 4, nous étudions un problème classique qui consiste à construire un ordonnancement réalisable sur un ensemble de machines *dédiées*. Les tâches considérées sont de durées unitaires et elles sont reliées par un graphe de précédence ayant la structure *d'un ordre d'intervalles*. Nous supposons que chaque tâche possède une date d'arrivée et une date d'échéance, et nous considérons des délais de communication entre les tâches reliées par des contraintes de précédence et s'exécutant sur des machines différentes. De plus, nous considérons des tâches *typées* dans le sens où chaque tâche ne peut être exécutée que par un sous ensemble de machines spécialisées. Notons par ailleurs qu'une machine est dite *dédiée* lorsqu'elle est la seule à être spécialisée dans l'exécution d'un type donné de tâches.

Nous montrons, tout d'abord, que le problème est *NP*-complet dans le cas où les délais de

communication sont quelconques. Ensuite, nous faisons l'hypothèse que ces délais de communication sont monotones et nous construisons une extension d'un algorithme de liste présenté par Leung et al. [62]. Nous prouvons que, sous cette hypothèse, l'extension de l'algorithme de Leung et al. résout notre problème de décision en un temps polynomial.
Finalement, nous montrons par un simple contre exemple que dans le cas de grands temps de communication, il n'est pas possible de construire un algorithme de liste pour résoudre la variante du problème où il existe plusieurs machines identiques spécialisées pour l'exécution d'un type donné de tâches. Néanmoins, la question reste ouverte lorsque les temps de communication sont unitaires.

Dans le chapitre 5, nous nous consacrons à l'étude du problème de placement de données dans un réseau de partage de type pair-à-pair. Ce problème consiste à répliquer au niveau de chaque noeud du réseau un ensemble de fichiers de façon à minimiser le coût d'accès total de tous les noeuds à l'ensemble des données requises par chacun des noeuds. Par ailleurs, nous supposons que la capacité de stockage de chaque noeud du réseau est limitée. Dans sa forme la plus générale, Baev et al. ont montré dans [17] qu'il n'existe pas de schéma d'approximation polynomial pour ce problème à moins que que $P = NP$. Nous nous limitons alors à des topologies particulières du réseau (les chaînes, les anneaux et les étoiles généralisées) pour lesquelles nous décrivons des algorithmes optimaux basés sur la programmation dynamique. Nous montrons ensuite qu'il n'est pas possible d'étendre cette approche dans le cas d'une topologie quelconque avec un nombre constant de noeuds de degré supérieur ou égal à 3.

Enfin dans la conclusion, nous résumons l'ensemble des travaux effectués et nous présentons quelques perspectives qui nous semblent intéressantes.

C 1

O ´ ’´ :

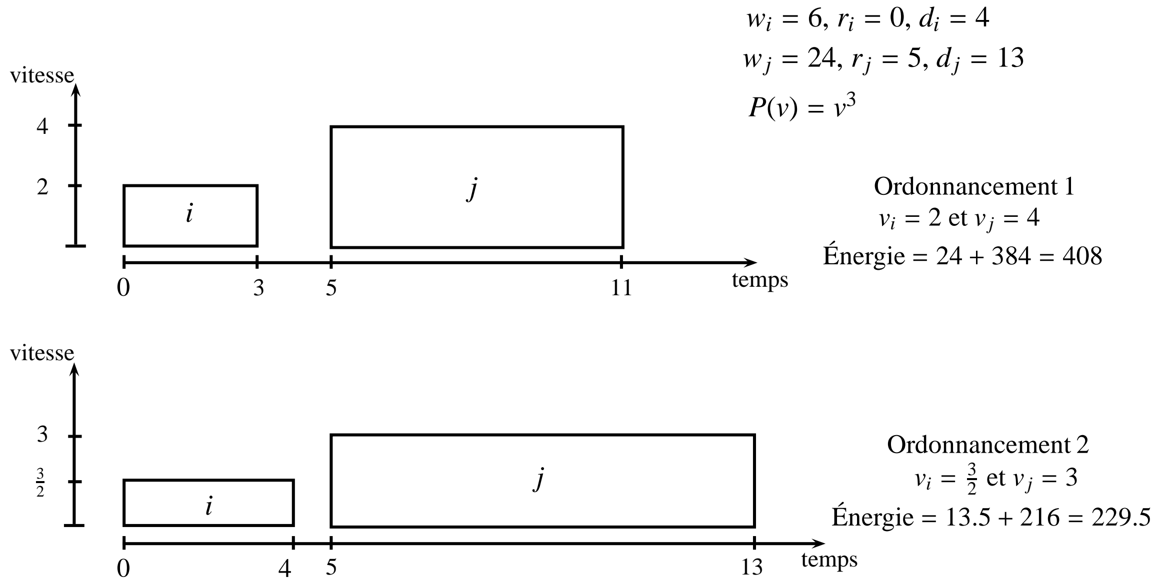
1.1 I

L’efficacité énergétique des systèmes informatiques est devenue un axe de recherche important durant ces dernières années. Des mécanismes prometteurs ont été mis en place dans le but de réduire efficacement l’énergie consommée par les unités de calcul allant des petits appareils portatifs jusqu’aux grands centres de données. D’un point de vue algorithmique, de nouveaux problèmes d’optimisation apparaissent où la consommation d’énergie est prise en compte en tant que contrainte du problème ou en tant que fonction objectif [3]. Cette dernière approche a été adoptée par Yao et al. dans un papier précurseur [83] où il s’agit d’ordonnancer un ensemble de tâches indépendantes caractérisées par leurs dates d’arrivée et leurs dates d’échéance sur une machine unique qui peut faire varier sa vitesse de manière dynamique dans le but de minimiser l’énergie totale consommée. Ce modèle est connu sous le nom de *Speed Scaling* ou adaptation de la vitesse. Dans ce modèle, si la machine fonctionne à une petite vitesse alors l’énergie consommée est faible, mais la qualité de l’ordonnancement retourné risque d’être mauvaise. Il s’agit donc d’adapter d’une façon dynamique la vitesse afin de trouver un compromis permettant de réduire l’énergie dépensée tout en préservant une bonne qualité de l’ordonnancement.

Le modèle de calcul de l'énergie le plus étudié dans la littérature est défini comme suit : si la vitesse d'une machine est égale à une valeur v pendant une durée donnée Δ, alors la puissance de consommation d'énergie est donnée par v^α pour une constante $\alpha > 1$, et l'énergie totale consommée est $v^\alpha \times \Delta$ [39]. Si nous supposons, maintenant, que la vitesse de la machine est donnée par une fonction du temps, $v(t)$, alors l'énergie totale consommée pendant la durée Δ est calculée par intégration de la puissance sur la durée Δ : $\int_\Delta v(t)^\alpha dt$. Il faut noter que la forme de la fonction de la puissance de consommation d'énergie implique qu'une petite augmentation de la vitesse peut causer une croissance importante dans la dépense de l'énergie par le système.

Dans un problème d'ordonnancement, chaque tâche est caractérisée par une quantité de travail qui modélise le nombre de cycles processeur requis pour l'exécuter en entier. La durée totale de cette tâche dépend, donc, de la vitesse du système. Soit une tâche i et notons par w_i sa quantité de travail. Si cette tâche est exécutée sur une machine qui tourne à une vitesse v, alors elle aura besoin de w_i/v unités de temps pour finir et l'énergie consommée par la machine est $E_i = v^\alpha.\frac{w_i}{v}$. La figure 1.1 montre l'influence du choix de la vitesse sur l'énergie consommée. Dans cet exemple nous considérons deux tâches i et j avec des quantités de travail $w_i = 6$ et $w_j = 24$, des dates d'arrivée $r_i = 0$ et $r_j = 5$, et des dates d'échéance $d_i = 4$ et $d_j = 13$. Nous considérons aussi une fonction de puissance $P(v) = v^3$. Par ailleurs, nous supposons que chaque tâche est exécutée à une vitesse constante et nous présentons deux ordonnancements réalisables dans lesquels nous modifions les vitesses d'exécution des tâches et nous montrons que le système économise d'avantage d'énergie en diminuant les vitesses d'exécution. Notons que dans cet exemple, la quantité de travail de chaque tâche est égale à la surface du rectangle correspondant.

Cas d'une seule machine. Dans [83], Yao et al. ont proposé un algorithme fondamental connu sous le nom de *"YDS"* selon les initiales des auteurs. Cet algorithme résout en un temps polynomial et d'une façon optimale le problème off-line de minimisation d'énergie sur une seule machine avec préemption, *i.e.* l'exécution d'une tâche quelconque peut être interrompue et reprise plus tard. Ils ont également considéré dans le même papier le cas on-line pour lequel ils ont proposé



F . 1.1 – Influence du choix des vitesses sur l'énergie consommée.

deux algorithmes : le premier noté *"AVR"* (Average Rate) est $2^{\alpha-1}\alpha^{\alpha}$-compétitif par rapport à l'énergie, et le deuxième, noté *"OA"* (Optimal Available). Dans [18], Bansal et al. ont prouvé à l'aide d'une fonction de potentiel que l'algorithme *OA* est α^{α}-compétitif, et ils ont décrit un nouvel algorithme on-line *"BKP"* avec un rapport de compétitivité égal à $2(\frac{\alpha}{\alpha-1})^{\alpha}e^{\alpha}$ et améliorant ainsi la qualité de *OA* pour des grandes valeurs de α.

Cas de plusieurs machines. Dans ce cas, il est possible de distinguer deux variantes différentes du modèle d'adaptation de la vitesse (*Speed Scaling*). La première variante notée *variante sans migration*, autorise la préemption des tâches mais n'autorise pas la migration. Ceci implique que l'exécution de toute tâche peut être interrompue et reprise plus tard sur la même machine et il n'est pas autorisé que cette reprise ait lieu sur une autre machine. Dans la seconde variante *avec migration*, la préemption et la migration des tâches sont toutes les deux autorisées. Il faut noter cependant que, dans tous les cas, l'exécution parallèle des tâches n'est permise, *i.e.* à chaque instant aucune tâche ne peut être exécutée simultanèment sur deux machines différentes.

Dans [7], Albers et al. ont considéré la variante *sans migration* du problème de minimisation de l'énergie. Dans le cas où les tâches possèdent des quantités de travail unitaires et des

dates d'arrivée et d'échéance agréables, les auteurs ont proposé un algorithme polynomial basé sur la politique *Round Robin* pour l'affectation des tâches aux machines. Par ailleurs, si les dates d'arrivée et d'échéance sont quelconques, alors le problème devient NP-difficile même pour des quantités de travail unitaires. Dans ce dernier cas, Albers et al. ont décrit l'algorithme approché *"CRR"* (Classified Round Robin) qui a un rapport d'approximation égal à $\alpha^{\alpha}2^{4\alpha}$. Dans le cas où toutes les tâches partagent la même date d'arrivée ou d'échéance, les auteurs ont proposé un autre algorithme d'approximation *"EDL"* (Earliest Deadline and List scheduling) pour des quantités de travail quelconques où le rapport d'approximation est $2(2-\frac{1}{m})^{\alpha}$, m étant le nombre de machines de l'instance. Dans [43], Greiner et al. présentent une réduction générique transformant tout algorithme β-approché pour le problème sur une seule machine en un algorithme βB_{α}-approché pour le problème sur plusieurs machines sans migration, où B_{α} est le $\alpha^{ème}$ nombre de Bell[1]. Ils ont montré également qu'une β-approximation pour le problème à plusieurs machines avec migration conduit à une βB_{α}-approximation pour le problème à plusieurs machines sans migration.

Maintenant, dans le contexte de plusieurs machines avec migration, Chen et al. [34] ont initié l'étude du problème de minimisation de l'énergie consommée et ils ont décrit un algorithmme optimal simple dans le cas où toutes les tâches sont disponibles au même instant et partagent la même date d'échéance. Dans [21], Bingham et al. ont montré que la version off-line du problème peut être résolu d'une façon polynomiale. Cependant, l'algorithme qu'ils ont proposé est basé sur la programmation linéaire et selon les auteurs sa complexité peut être assez élevée. Au moment de l'achèvement de cette étude, un papier de Albers et al. [5] est apparu où les auteurs ont considéré le même problème à savoir l'ordonnancement de tâches avec des dates d'arrivées et des dates d'échéances quelconques sur un ensemble de machines identiques avec migration et dans le but de minimiser l'énergie totale consommée. Albers et al. ont utilisé une approche combinatoire similaire à la notre basée sur le calcul de flots maximums et ils ont proposé un algorithme polynomial dont la complexité est en $O(n^2C(n,n^2))$, avec $C(x,y)$ est la complexité nécessaire pour calculer un

[1] Si on considère un ensemble E de cardinal n, alors le $n^{ème}$ nombre de Bell, noté B_n, est le nombre de partitions de E. Les nombres de Bell satisfont la formule de récurrence : $B_{n+1}=\sum_{k=0}^{n}C_n^kB_k$. Le premier nombre de Bell correspond à l'ensemble vide et il est égal à $B_0=1$.

flot maximum dans un graphe en couche avec n sommets et y arcs. Par ailleurs, les auteurs ont étendu l'analyse des algorithmes on-line sur une seule machine *AVR* et *OA* [83] dans le cas de plusieurs machines avec migration.

1.1.1 Contribution et organisation du chapitre

Dans ce chapitre, nous considérons le modèle d'adaptation de la vitesse (*speed scaling*) et nous étudions le problème d'ordonnancement sur plusieurs machines identiques avec comme objectif la minimisation d'énergie lorsque la migration des tâches est autorisée. Dans la section 1.3, nous formulons le problème en programme convexe et nous appliquons, ensuite, les conditions de Karush-Kuhn-Tucker (*conditions KKT*) [54, 57] afin d'obtenir des propriétés nécessaires à l'optimalité de toute solution du problème. Ensuite, nous décrivons dans la section 1.4 un algorithme combinatoire optimal basé sur le calcul de flots maximums et nous détaillons sa compléxité.

1.2 D´ `

Nous considérons l'ensemble des tâches $\mathcal{J} = \{J_1, \ldots, J_n\}$, tel que chaque tâche J_i soit caractérisée par une quantité de travail w_i, une date d'arrivée r_i et une date d'échéance d_i. Notons que toutes ces valeurs sont positives et non nulles. Ainsi, une tâche J_i est dite *disponible* à l'instant t si $t \in [r_i, d_i)$ et nous définissons sa densité $\delta_i = \frac{w_i}{d_i - r_i}$. Nous considérons, aussi, m machines homogènes telles que chaque machine soit capable de varier dynamiquement sa vitesse d'exécution. Nous supposons que sur chaque machine le spectre des vitesses autorisées est continu et non borné et que la consommation d'énergie est donnée par une fonction de puissance convexe $P(v) = v^\alpha$, v étant la vitesse d'exécution et α une constante strictement supérieure à 1. Plus précisément, si une tâche $J_i \in \mathcal{J}$ est exécutée sur une machine qui tourne à une vitesse constante v_i durant un intervalle de temps de longueur ℓ, alors la quantité de travail efféctuée est $v_i.\ell$ et $P(s).\ell$ unités d'énergie sont consommées. D'une façon plus générale, il est clair que l'énergie est la puissance intégrée sur le temps. Ainsi si nous considérons qu'à chaque instant t correspond une vitesse $v(t)$ sur une machine quelconque k, alors le travail total effectué est égal à $\int_{|\sigma|} v(t)dt$ et l'énergie totale consommée est $\int_{|\sigma|} v(t)^\alpha dt$, où $|\sigma|$ représente la durée d'un ordonnancement

quelconque σ sur la machine k.

Dans notre modèle, nous supposons que la préemption et la migration sont autorisées, *i.e.* l'exécution d'une tâche peut être suspendue et reprise ultérieurement sur la même machine ou sur une autre machine. Cependant, l'exécution parralèle n'est pas autorisée dans le sens où aucune tâche ne peut être exécutée simultanément sur deux ou plusieurs machines différentes. Notre objectif est de construire un ordonnancement réalisable qui minimise l'énergie totale consommée par toutes les machines.

Nous définissons $\mathcal{R}= \{t_0, \dots, t_L\}$ comme étant l'ensemble des dates d'arrivée et d'échéance des tâches dans $\mathcal{J}$ considérées dans l'ordre croissant et sans doublons. Il est clair que $t_0 = min_{J_i \in \mathcal{J}}\{r_i\}$ et que $t_L = \max_{J_i \in \mathcal{J}}\{d_i\}$. Pour tout $1 \leqslant j \leqslant L$, nous considérons l'intervalle $I_j = [t_{j-1}, t_j)$, et soit l'ensemble $\mathcal{I}= \{I_1, \dots, I_L\}$. Nous notons par $|I_j|$ la longueur de l'intervalle I_j. D'autre part, pour tout $1 \leqslant j \leqslant L$, soit D_j l'ensemble de toutes les tâches disponibles durant l'intervalle I_j, *i.e.* toutes les tâches i telles que $I_j \subseteq [r_i, d_i)$, et soit $|D_j|$ sa cardinalité. Étant donné un ordonnancement σ, on définit $t_{i,j}(\sigma)$ la durée totale d'exécution de la tâche J_i durant l'intervalle I_j dans l'ordonnancement σ.

1.3 C ` ’ ´

Le but de cette partie est de déduire les propriétés d'un ordonnancement optimal pour le problème de minimisation de l'énergie sur des machines parallèles avec migration.

1.3.1 Un programme convexe

Nous formulons notre problème par un programme convexe ce qui nous permettra, ensuite, d'appliquer les conditions de Karush−Kuhn−Tucker (KKT) afin d'obtenir des conditions nécessaires d'optimalité. Nous montrons, par la suite, que ces conditions sont suffisantes pour qu'un ordonnancement réalisable soit optimal.

Pour chaque tâche $J_i \in \mathcal{J}$ et pour chaque intervalle I_j tel que $J_i \in D_j$ nous introduisons les variables v_i et $t_{i,j}$ et qui définissent respectivement la vitesse de la tâche J_i et la durée d'exécution totale de la tâche J_i pendant l'intervalle I_j. Le programme convexe est décrit comme suit :

$$\min \sum_{J_i \in \mathcal{J}} w_i v_i^{\alpha-1} \tag{1.1}$$

$$\frac{w_i}{v_i} - \sum_{I_j :\ J_i \in D_j} t_{i,j} \leqslant 0 \qquad J_i \in \mathcal{J} \tag{1.2}$$

$$\sum_{J_i \in D_j} t_{i,j} - m \cdot |I_j| \leqslant 0 \qquad 1 \leqslant j \leqslant L \tag{1.3}$$

$$\sum_{J_i \in D_j} t_{i,j} - |D_j| \cdot |I_j| \leqslant 0 \qquad 1 \leqslant j \leqslant L \tag{1.4}$$

$$t_{i,j} - |I_j| \leqslant 0 \qquad 1 \leqslant j \leqslant L,\ J_i \in D_j \tag{1.5}$$

$$-t_{i,j} \leqslant 0 \qquad 1 \leqslant j \leqslant L,\ J_i \in D_j \tag{1.6}$$

$$-v_i \leqslant 0 \qquad J_i \in \mathcal{J} \tag{1.7}$$

Il faut noter que le temps d'exécution total et la consommation de l'énergie totale de chaque tâche J_i est donné, respectivement par $\frac{w_i}{v_i}$ et $w_i v_i^{\alpha-1}$. Il s'agit donc de minimiser la fonction objectif : l'énergie totale consommée par toutes les tâches, et qui est exprimée dans la ligne 1.1 du programme convexe. Les contraintes 1.2 expriment le fait que chaque tâche doit être exécutée entièrement, *i.e.* une quantité de travail w_i doit être exécutée pour chaque tâche $J_i \in \mathcal{J}$ à une vitesse constante v_i. Les contraintes 1.3 et 1.4 garantissent que durant chaque intervalle $I_j \in \mathcal{I}$ au plus $min\{m, |D_j|\}$ machines peuvent être utilisées. D'autre part, l'ensemble des contraintes 1.5 empêchent toute tâche J_i d'être exécutée pendant plus que $|I_j|$ unités de temps durant tous les intervalles $I_j \subseteq [r_i, d_i)$ afin d'éviter le risque d'exécution parallèle. Finalement, les contraintes 1.6 et 1.7 impliquent que les variables du programme convexe sont positives.

Précisons qu'il s'agit bien d'un programme convexe étant donnée que la fonction objectif dans la ligne 1.1, ainsi que le premier ensemble de contraintes dans la ligne 1.2 sont convexes. Toutes les autres contraintes sont linéaires. De ce fait, il est possible de résoudre ce programme convexe en utilisant la méthode de l'Ellipsoïde [68]. Cependant, nous nous proposons de construire un algorithme combinatoire plus simple et surtout plus efficace.

1.3.2 Les conditions KKT

Nous appliquons les conditions KKT au programme convexe décrit ci-dessus afin d'obtenir des conditions nécessaires d'optimalité de toute solution réalisable. Nous montrons, par la suite, que ces conditions suffisent pour qu'un ordonnancement donné soit optimal. Nous donnons, tout d'abord, une définition concise des conditions KKT.

La formulation générale d'un programme convexe est donnée par :

$$\begin{aligned} &\min f(x) && \\ &g_i(x) \leqslant 0 && 1 \leqslant i \leqslant q \\ &h_j(x) = 0 && 1 \leqslant j \leqslant r \\ &x \in \mathbb{R}^n && \end{aligned}$$

Supposons que ce programme soit strictement réalisable dans le sens où il existe une solution $x^* \in \mathbb{R}^n$ telle que $g_i(x^*) < 0$ et $h_j(x^*) = 0$ pour tout $1 \leqslant i \leqslant q$ et $1 \leqslant j \leqslant r$, où g_i et h_j sont différentiables en x^*. Soit λ_i et μ_j les variables duales associées, respectivement, aux contraintes $g_i(x) \leqslant 0$ et $h_j(x) = 0$. Dans ce cas, les conditions de Karush $-$Kuhn$-$Tucker (KKT) sont données par :

$$g_i(x) \leqslant 0 \qquad 1 \leqslant i \leqslant q \tag{1.8}$$

$$h_j(x) = 0 \qquad 1 \leqslant j \leqslant r \tag{1.9}$$

$$\lambda_i \geqslant 0 \qquad 1 \leqslant i \leqslant q \tag{1.10}$$

$$\lambda_i g_i(x) = 0 \qquad 1 \leqslant i \leqslant q \tag{1.11}$$

$$\nabla f(x) + \sum_{i=1}^{q} \lambda_i \nabla g_i(x) + \sum_{j=1}^{m} \mu_j \nabla h_j(x) = 0 \tag{1.12}$$

Ces conditions sont nécessaires et suffisantes pour que des solutions $x^* \in \mathbb{R}^n$, $\lambda^* = (\lambda_1^*, \lambda_2^*, \ldots, \lambda_q^*) \in \mathbb{R}^q$ et $\mu^* = (\mu_1^*, \mu_2^*, \ldots, \mu_r^*) \in \mathbb{R}^r$ soient primales et duales optimales. Les conditions 1.8 et 1.9 garantissent l'admissibilité de la solution du problème primal et l'ensemble des conditions 1.10 garantissent l'admissibilité de la solution du problème dual. Les conditions 1.11 sont dites conditions de complémentarité et les conditions 1.12 représentent les conditions stationnaires.

Le lemme suivant résulte de l'application des conditions KKT à la formulation convexe de notre problème.

Lemme 1 *Chaque ordonnancement optimal du problème de minimisation d'énergie sur des machines parallèles avec migration satisfait les propriétés suivantes :*

1. *Chaque tâche J_i est exécutée à une vitesse constante v_i.*
2. *Pour tout intervalle I_j, on a $\sum_{J_i \in D_j} t_{i,j} = \min\{|D_j|, m\}.|I_j|$.*
3. *Si $|D_j| \leqslant m$ durant un intervalle I_j, alors $t_{i,j} = |I_j|$, pour chaque tâche J_i telle que $I_j \subseteq [r_i, d_i)$.*
4. *Si $|D_j| > m$ alors*
   - i *Toutes les tâches J_i disponibles durant I_j, avec $0 < t_{i,j} < |I_j|$, sont exécutées à la même vitesse.*
   - ii *Si une tâche J_i n'est pas exécutée durant un intervalle $I_j \subset [r_i, d_i)$, i.e. $t_{i,j} = 0$, alors $v_i \leqslant v_k$ pour chaque tâche J_k telle que $I_j \subseteq [r_k, d_k)$ et $t_{k,j} > 0$.*
   - iii *Si une tâche J_i vérifie $t_{i,j} = |I_j|$ pour un intervalle I_j donné, alors $v_i \geqslant v_k$ pour chaque tâche J_k disponible durant I_j avec $t_{k,j} < |I_j|$.*

Preuve Soit une instance du problème avec migration et supposons, par contradiction, qu'il existe un ordonnancemment optimal σ tel que au moins une tâche est exécutée avec plusieurs vitesses différentes. Soit J_k une telle tâche et supposons qu'il existe un ensemble de vitesses $\{v_1, \ldots, v_L\}$ et un ensemble de durées $\{\ell_1, \ldots, \ell_L\}$ tel que la tâche J_k soit exécutée à la vitesse v_i pendant ℓ_i unités de temps pour tout $1 \leqslant i \leqslant L$. Ainsi, l'énergie totale dépensée par l'exécution de la tâche J_k est donnée par $E_\sigma(J_k) = \sum_{i=1}^{L} v_i^\alpha.\ell_i$.

Soit σ' un ordonnancement identique à σ avec la seule différence que la tâche J_k est exécutée à la vitesse constante $v_k = \frac{\sum_{i=1}^{L} v_i.\ell_i}{\sum_{i=1}^{L} \ell_i}$ pendant les mêmes intervalles de temps. Il est clair que σ' reste un ordonnancement réalisable étant donné que $\sum_{i=1}^{L} v_i.\ell_i$ constitue la quantité de travail de la tâche J_k. Par ailleurs, la nouvelle énergie consommée par J_k est donnée par $E_{\sigma'}(J_k) = v_k^\alpha. \sum_{i=1}^{L} \ell_i$.

Étant donnée que la fonction de puissance $P(v) = v^\alpha$, pour $\alpha > 1$, est strictement convexe, il en découle que pour toute ensemble $\{\theta_1, \ldots, \theta_L\}$, tel que $\theta_i \in [0, 1] \forall 1 \leqslant i \leqslant L$ et $\sum_{i=1}^{L} \theta_i = 1$, on a :

$$P(\theta_1 v_1 + \ldots + \theta_L v_L) < \theta_1 P(v_1) + \ldots + \theta_L P(v_L)$$

Ceci implique que :

$$P(\frac{\ell_1}{\sum_{j=1}^{L} \ell_j} v_1 + \ldots + \frac{\ell_L}{\sum_{j=1}^{L} \ell_j} v_L) < \frac{\ell_1}{\sum_{j=1}^{L} \ell_j} P(v_1) + \ldots + \frac{\ell_L}{\sum_{j=1}^{L} \ell_j} P(v_L) \Rightarrow$$

$$\sum_{j=1}^{L} \ell_j P(v_k) < \ell_1 P(v_1) + \ldots + \ell_L P(v_L) \Rightarrow$$

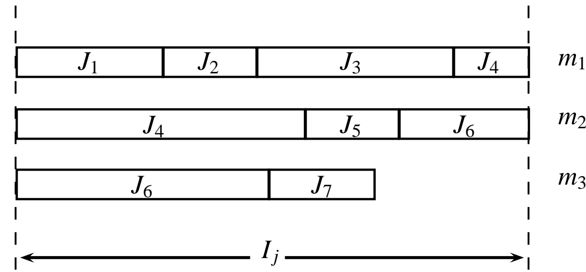
$$E_{\sigma'}(J_k) < E_\sigma(J_k)$$

D'où une contradiction vis à vis de l'optimalité de σ, et la première propriété est donc vérifiée.

Considérons, maintenant un intervalle quelconque $I_j \in \mathcal{I}$. Étant donné que pour toute tâche $J_i \in D_j$, $t_{i,j} \leqslant |I_j|$, il est clair que $\sum_{J_i \in D_j} t_{i,j} \leqslant \min\{|D_j|, m\}.|I_j|$.

Supposons par contradiction qu'il existe un ordonnancement optimal σ tel que $\sum_{J_i \in D_j} t_{i,j} < \min\{|D_j|, m\}.|I_j|$. Il est possible de transformer la partie de l'ordonnancement σ qui estréstreinte à l'intervalle I_j de façon à ce que toutes les tâches soient exécutées dans un ordre quelconque sans

préemption et sans période d'inactivité entre la fin d'une tâche et le début de la tâche suivantes. Au niveau de chaque machine, s'il existe des tâches à exécuter, alors l'exécution commence toujours au début de l'intervalle I_j et on a recours à la migration des tâches à chaque fois que c'est necéssaire, *i.e.* à chaque fois que la taille de la partie encore disponible de l'intervalle I_j sur une machine donnée m_p ne permet pas d'exécuter une tâche J_k en sa totalité. Dans ce cas, J_k débute sur la machine m_p et se termine sur la machine m_{p+1}. La forme de cet ordonnancement est schématisée dans l'exemple de la figure 1.2.



F . 1.2 – Exemple d'ordonnancement sans préemption durant un intervalle I_j sur 3 machines.

Étant donné que $\sum_{J_i \in D_j} t_{i,j} < \min\{|D_j|, m\}.|I_j|$, alors il existe au moins une tâche J_k dans l'ordonnancement obtenu qui commence son exécution au début de l'intervalle I_j et telle que $t_{k,j} < |I_j|$. Pour le même argument, il existe aussi au moins une machine où aucune tâche n'est exécutée durant un intervalle $\delta_\epsilon = [t_j - \epsilon, t_j)$ pour une valeur $\epsilon > 0$ arbitrairement petite (t_j est la borne supérieure de I_j). Il est donc possible de réduire la vitesse de la tâche J_k en faisant étendre son exécution sur l'intervalle δ_ϵ pour une valeur adéquate de ϵ. On obtient ainsi un nouvel ordonnancement avec une consommation d'énergie moins importante. D'où une contradiction de l'optimalité de l'ordonnancement σ et ceci valide la deuxième propriété.

La troisième propriété est une conséquence de la deuxième propriété. En effet, si $|D_j| \leqslant m$, alors $\sum_{J_i \in D_j} t_{i,j} = |D_j||I_j|$. Supposons donc par contradiction qu'il existe une tâche $J_k \in D_j$ telle que $t_{k,j} < |I_j|$. Dans ce cas, on a $\sum_{J_i \in D_j} t_{i,j} < |D_j||I_j|$, ce qui contredit la deuxième propriété.

Nous nous proposons maintenant d'appliquer les conditions KKT au programme convexe dé-

fini précédemment pour prouver les propriétés $4(i)$, $4(ii)$ et $4(iii)$. Ainsi, nous associons les variables duales β_i, γ_j, δ_j, $\epsilon_{i,j}$, $\zeta_{i,j}$ et η_i respectivement aux ensembles des contraintes allant de 1.2 à 1.7. Par les conditions stationnaires 1.12 on a :

$$\begin{aligned}
\nabla \sum_{J_i \in \mathcal{J}} w_i v_i^{\alpha-1} + \sum_{J_i \in \mathcal{J}} \beta_i \cdot \nabla\Big(\frac{w_i}{v_i} - \sum_{I_j:\, J_i \in D_j} t_{i,j}\Big) \\
+ \sum_{j=1}^{L} \gamma_j \nabla\Big(\sum_{J_i \in D_j} t_{i,j} - m \cdot |I_j|\Big) + \sum_{j=1}^{L} \delta_j \nabla\Big(\sum_{J_i \in D_j} t_{i,j} - |D_j| \cdot |I_j|\Big) \\
+ \sum_{j=1}^{L} \sum_{J_i \in D_j} \epsilon_{ij} \nabla(t_{i,j} - |I_j|) + \sum_{j=1}^{L} \sum_{J_i \in D_j} \zeta_{ij} \nabla(-t_{i,j}) + \sum_{J_i \in \mathcal{J}} \eta_i \nabla(-v_i) &= 0
\end{aligned}$$

Cette equation peut être reformulée comme suit :

$$\begin{aligned}
\sum_{j=1}^{L} \sum_{J_i \in D_j} \Big(-\beta_i + \gamma_j + \delta_j + \epsilon_{i,j} - \zeta_{i,j}\Big) \nabla t_{i,j} \\
+ \sum_{J_i \in \mathcal{J}} \Big((\alpha - 1) w_i s_i^{\alpha-2} - \frac{\beta_i w_i}{v_i^2} - \eta_i\Big) \nabla s_i &= 0 \qquad (1.13)
\end{aligned}$$

De plus, les conditions de complémentarité 1.11 impliquent que :

$$\beta_i \cdot \Big(\frac{w_i}{v_i} - \sum_{I_j:\, J_i \in D_j} t_{i,j}\Big) = 0 \qquad J_i \in \mathcal{J} \qquad (1.14)$$

$$\gamma_j \cdot \Big(\sum_{J_i \in D_j} t_{i,j} - m \cdot |I_j|\Big) = 0 \qquad 1 \leqslant j \leqslant L \qquad (1.15)$$

$$\delta_j \cdot \Big(\sum_{J_i \in D_j} t_{i,j} - |D_j| \cdot |I_j|\Big) = 0 \qquad 1 \leqslant j \leqslant L \qquad (1.16)$$

$$\epsilon_{ij} \cdot (t_{i,j} - |I_j|) = 0 \qquad 1 \leqslant j \leqslant L,\ J_i \in D_j \qquad (1.17)$$

$$\zeta_{ij} \cdot (-t_{i,j}) = 0 \qquad 1 \leqslant j \leqslant L,\ J_i \in D_j \qquad (1.18)$$

$$\eta_i \cdot (-v_i) = 0 \qquad J_i \in \mathcal{J} \qquad (1.19)$$

Il est possible de supposer qu'il n'existe aucune tâche avec une quantité de travail nulle. Ainsi, pour toute tâche J_i, il est vrai que $v_i > 0$ et que $\sum_{I_j \subseteq [r_i, d_i)} t_{i,j} > 0$. De ce fait, l'équation 1.19 implique que $\eta_i = 0, \forall 1 \leqslant i \leqslant n$. En annulant les dérivés partielles ∇v_i et $\nabla t_{i,j}$ dans 1.13, nous

obtenons $\beta_i = (\alpha - 1)v_i^\alpha$ pour chaque tâche $J_i \in \mathcal{J}$ et

$$(\alpha - 1)v_i^\alpha = \gamma_j + \delta_j + \epsilon_{i,j} - \zeta_{i,j} \quad (1.20)$$

pour chaque tâche $J_i \in \mathcal{J}$ et $I_j \subseteq [r_i, d_i)$. Pour chaque intervalle I_j, $D_j > m$. A partir de l'équation 1.16 et de la deuxième propriété, on déduit que $\delta_j = 0$. Considérons maintenant les cas suivants obtenus en fonction de la durée d'exécution de chaque tâche $J_i \in D_j$:

1. Cas 1 : $0 < t_{i,j} < |I_j|$

   Les conditions complémentaires 1.17 et 1.18 impliquent que $\epsilon_{i,j} = \zeta_{i,j} = 0$. Ainsi, 1.20 devient :

   $$(\alpha - 1)v_i^\alpha = \gamma_j$$

   La variable γ_j est specifique pour chaque intervalle I_j et donc, toutes les tâches dans ce cas possèdent la même vitesse durant tout l'ordonnancement et la propriété 4(i) est vérifiée. Nous allons noté cette vitesse par $v(I_j)$ pour chaque intervalle I_j.

2. Cas 2 : $t_{i,j} = 0$

   Ceci implique, par 1.17, que $\epsilon_{i,j} = 0$ et 1.20 peut être exprimée par :

   $$(\alpha - 1)v_i^\alpha = \gamma_j - \zeta_{i,j}$$

   de ce fait, comme $\zeta_{i,j} \geqslant 0$, nous avons $v_i \leqslant v(I_j)$ pour toute tâche J_i de ce type. D'où la propriété 4(ii) est vérifiée.

3. Cas 3 : $t_{i,j} = |I_j|$

   Dans ce cas, par 1.18, nous obtenons $\zeta_{i,j} = 0$. Ainsi, 1.20 devient :

   $$(\alpha - 1)v_i^\alpha = \gamma_j + \epsilon_{i,j}$$

   À cause des conditions d'admissibilité duales 1.10, $\epsilon_{i,j} \geqslant 0$. D'où, toutes les tâches de ce type ont des vitesses vérifiant $v_i \geqslant v(I_j)$, et la propriété 4(iii) est donc valide.

□

Lemme 2 *Les propriétés énumérées dans le lemme 1 sont suffisantes pour l'optimalité d'une solution du problème.*

Preuve Supposons, par contradiction, qu'il existe un ordonnancement A non optimal et qui satisfait les propriétés du lemme 1 et considérons, aussi, un autre ordonnancement optimal B satisfaisant les mêmes propriétés. Notons par E^x, v_i^x et $t_{i,j}^x$, respectivement, l'énergie consommée, la vitesse de la tâche J_i et le temps d'exécution total de J_i durant l'intervalle I_j dans un ordonnancement donné X. Ainsi, on a $E^A > E^B$. Soit S l'ensemble de toutes les tâches J_i telles que $v_i^A > v_i^B$. Il est clair qu'il existe au moins une tâche J_k telle que $v_k^A > v_k^B$ car sinon l'ordonnancement A ne consommerait pas plus d'énergie que l'ordonnancement B. Ainsi, $S \neq \emptyset$ et par définition de S nous avons :

$$\sum_{J_i \in S} \sum_{I_j : J_i \in D_j} t_{i,j}^A < \sum_{J_i \in S} \sum_{I_j : J_i \in D_j} t_{i,j}^B$$

Il existe, donc, au moins un intervalle I_p tel que :

$$\sum_{J_i \in S} t_{i,p}^A < \sum_{J_i \in S} t_{i,p}^B$$

Ceci implique que $t_{k,p}^A < t_{k,p}^B$ pour $J_k \in S$. Ainsi, $t_{k,p}^A < |I_p|$ et $t_{k,p}^B > 0$. Soit un intervalle quelconque I_j, la somme des temps d'exécution de toutes les tâches dans I_j est constante pour tout ordonnancement vérifiant les propriétés du lemme 1. Ainsi, il existe forcément une tâche $J_\ell \notin S$ telle que $t_{\ell,p}^A > t_{\ell,p}^B$. Donc, $t_{\ell,p}^A > 0$ et $t_{\ell,p}^B < |I_p|$. Par la propriété 4(iii) du lemme 1, nous concluons que $v_\ell^A \geqslant v_k^A > v_k^B \geqslant v_\ell^B$, or ceci contredit le fait que $J_\ell \notin S$. □

Il reste à remarquer que les propriétés des solutions optimaux citées dans le lemme 1 ne permettent pas d'aboutir directement à la solution optimale mais peuvent, en revanche, nous guider pour construire un ordonnancement optimal. Il s'agit, en effet, d'attribuer à chaque tâche une vitesse constante et de spécifier l'ensemble des tâches qui seront exécutées dans chaque intervalle de temps et sur chaque machine.

1.4 U ´

Dans cette partie, nous proposons un algorihme combinatoire pour résoudre le problème de minimisation d'énergie sur des machines identiques avec migration. Cet algorithme construit toujours un ordonnancement qui satisfait les propriétés du lemme 1 pour toute instance du problème. Comme nous l'avons déjà mentionné précédement, ces propriétés sont nécessaires et suffisantes pour l'optimalité, ce qui impliquerait que notre algorithme retourne toujours une solution optimale.

Cet algorithme, basé sur la notion de "*tâches critiques*" que nous allons définir par la suite, procède par étapes. Il s'agit, à chacune des étapes, de diminuer la taille de l'ensemble des tâches à ordonnancer en détectant un sous ensemble de tâches critiques auxquelles nous attribuons une unique vitesse constante dite "*vitesse critique*". Au terme de cette boucle, une vitesse constante s_i est attribuée à chaque tâche J_i. La recherche des vitesses (et des tâches) critiques repose sur le calcul de flot maximum dans un graphe orienté correspondant à l'instance en cours de résolution. Cette procédure correspond à la résolution d'une instance du problème noté "*AVC*" et qui sera décrit en détail dans cette partie. À ce stade de l'algorithme, la durée d'exécution de chaque tâche J_i est donnée par w_i/s_i et finalement nous obtenons une instance du problème d'ordonnancement classique $P|r_i, d_i, pmtn|-$ dans lequel on doit décider s'il existe un ordonnancement réalisable avec préemption et migration d'un ensemble de tâches caractérisées par leurs durées, leurs dates d'arrivée et leurs dates d'échéance sur un ensemble de m machines identiques. Ce problème peut être résolu en temps polynomial en le reformulant comme un problème de flot maximum [69].

1.4.1 Le problème de flot-max/coupe-min

Dans ce qui suit, nous présentons, brièvement, quelques notations et définitions concernant les problèmes de flot maximum et de coupe minimale. Soit un graphe $G = (V, A)$ orienté dans lequel chaque arc $(u, v) \in A$ possède une capacité $c(u, v)$, et soit deux noeuds $s, p \in V$. Une (s, p)-coupe

de G est une partition des noeuds du graphe en 2 sous-ensembles disjoints X et Y tels que $s \in X$ et $p \in Y$ et tels que la suppression de tous les arcs (u, v) avec $u \in X$ et $v \in Y$ entraîne la suppression de tous les chemins entre les noeuds s et p. Une (s, p)-coupe (X, Y) est dite minimale si la somme des capacités de tous les arcs (u, v) avec $u \in X$ et $v \in Y$ est minimale. Dans la suite, si nous parlons d'une coupe alors il sera fait référence à l'ensemble des arcs qui la forme.

Un (s, p)-flot $\mathcal{F}$ dans le graphe G est tout simplement une quantité de même unité que la capacité des arcs à faire passer du noeud s au noeud p à travers le graphe G. Nous notons $|\mathcal{F}|$ le coût du flot $\mathcal{F}$ et qui n'est autre que le flot sortant de s (égal au flot entrant en p). Nous utilisons, aussi, le terme $f(u, v)$ pour représenter la quantité du flot passant par l'arc $(u, v) \in A$. Par ailleurs, chaque flot $\mathcal{F}$ doit satisfaire les propriétés suivantes :

i $f(u, v) \leqslant c(u, v)$, pour tout arc $(u, v) \in A$,

ii $f(u, v) = -f(v, u)$, pour tout arc $(u, v) \in A$,

iii $\sum_{\substack{v \in V \\ (u,v) \in A}} f(u, v) = \sum_{\substack{w \in V \\ (w,u) \in A}} f(w, u)$, pour tout noeud $u \in V$,

iv $\sum_{\substack{v \in V \\ (s,v) \in A}} f(s, v) = \sum_{\substack{w \in V \\ (w,p) \in A}} f(w, p) = |\mathcal{F}|$.

Etant donné un graphe G et un flot $\mathcal{F}$, nous définissons également, le graphe résiduel correspondant G_f comme suit :

i G_f possède le même ensemble de noeuds que le graphe G,

ii pour chaque arc $(u, v) \in A$, tel que $f(u, v) < c(u, v)$, on rajoute à G_f l'arc (u, v) avec une capacité égale à $c(u, v) - f(u, v)$,

iii Pour chaque arc (u, v) avec $f(u, v) > 0$, nous incluons à G_f l'arc (v, u) auquel on attribue la capacité $f(u, v)$.

Soit un flot $\mathcal{F}$. On dit que ce flot sature un arc (u, v) si $f(u, v) = c(u, v)$. De plus, on considère qu'un chemin π est saturé par $\mathcal{F}$ s'il existe au moins un arc (u, v) dans π qui soit saturé.

1.4.2 Le problème d'affectation à vitesse constante "*AVC*"

Nous présentons dans ce paragraphe le problème d'affectation à vitesse constante "*AVC*", ainsi que la notion de *tâches crititques* et d'*instance critique*.

Le problème *AVC* peut être vu comme une variante du problème $P|r_i, d_i, pmtn|-$ et il peut être décrit comme suit : soit une instance du problème initial caractérisée par l'ensemble des tâches $\mathcal{J}$, l'ensemble de m machines et l'ensemble d'intervalles $\mathcal{I}$. Nous rappelons que chaque tâche $J_i \in \mathcal{J}$ peut être disponible dans un ou plusieurs intervalles de $\mathcal{I}$, et nous supposons que durant chaque intervalle $I_j \in \mathcal{I}$, $m_j \leqslant m$ machines peuvent être utilisées. Par ailleurs, soit une constante $v > 0$. Le problème *AVC* consiste donc à répondre à la question suivante : existe-il un ordonnancement réalisable qui exécute toutes les tâches dans $\mathcal{J}$ à la vitesse v ? Rappelons qu'un ordonnancement est dit réalisable si et seulement si chaque tâche est exécutée durant ses intervalles de disponibilité et par au plus une machine à chaque instant t. Nous précisons, également, que la préemption et la migration des tâches sont autorisées.

Notons que le problème *AVC* est similaire au problème de décision $P|r_i, d_i, pmtn|-$ avec la seule différence que, dans le problème *AVC*, le nombre de machines m_j pouvant être utilisées durant chaque intervalle $I_j \in \mathcal{I}$ n'est pas constant.

Réduction de *AVC* en un problème de flot. Chaque instance du problème *AVC* peut être réduite en une instance du problème de flot maximum comme suit : soit une instance $(\mathcal{J}, \mathcal{I}, v)$ du problème *AVC*. Nous définissons le graphe correspondant $G = (V, E)$ qui contient un noeud (x_i)

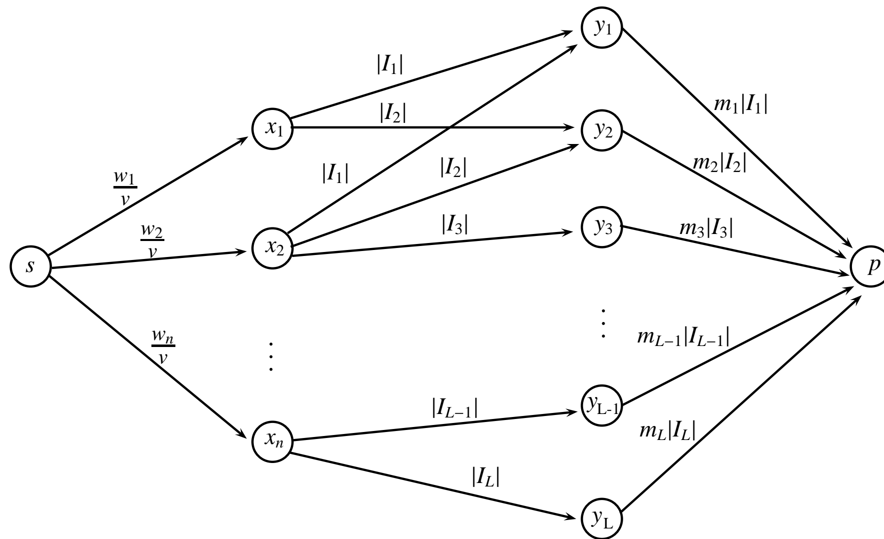
associé à chaque tâche $J_i \in \mathcal{J}$, un noeud (y_j) pour chaque intervalle $I_j \in \mathcal{I}$, un noeud source (s) et un noeud puits (p). Ces noeuds sont reliés comme suit :

- Un arc (s, x_i) de capacité w_i/v relie la source (s) à chaque noeud (x_i), $1 \leqslant i \leqslant |\mathcal{J}|$.

- Un arc (x_i, y_j) de capacité $|I_j|$ relie le noeud (x_i) au noeud (y_j) si la tâche J_i est disponible durant l'intervalle I_j, *i.e.* $J_i \in D_j$.

- Un arc (y_j, p) de capacité $m_j|I_j|$ relie chaque noeud (y_j) au noeud puits (p).

Ce graphe est schématisé dans la figure 1.3.



F . 1.3 – Réduction de l'instance $(\mathcal{J}, \mathcal{I}, v)$ du problème *AVC* en un problème de flot maximum. Dans cet exemple, les ensembles $\mathcal{J} = \{J_1, \ldots, J_n\}$ et $\mathcal{I} = \{I_1, \ldots, I_L\}$ représentent les ensembles de tâches et d'intervalles correspondant au problème original de minimisation d'énergie avec migration. Initialement, toutes les machines sont disponible durant chaque intervalle I_j, *i.e.* $m_1 = \ldots = m_L = m$.

Nous définissons, à ce stade, les notions de *tâches critiques* et d'*instance critique*.

Définition 1 *Soit* $(\mathcal{J}, \mathcal{I}, v)$ *une instance réalisable du problème AVC. Une tâche* $J_c \in \mathcal{J}$ *est dite critique, si et seulement si, pour chaque solution réalisable* σ *et pour chaque intervalle* $I_j \subseteq [r_c, d_c)$*, on a soit* $t_{c,j}(\sigma) = |I_j|$*, soit* $\sum_{J_i \in D_j} t_{i,j}(\sigma) = m_j|I_j|$.

Définition 2 *Une instance* $(\mathcal{J}, \mathcal{I}, v)$ *du problème AVC est critique, si et seulement si,* v *est la vitesse minimale qui garantit que cette instance est réalisable. Dans ce cas, la vitesse* v *est appelée vitesse critique pour l'instance en question.*

Dans la suite, nous allons présenter une suite de propriétés relatives au problème *AVC*, et le but final sera de trouver les conditions d'existence de tâches critiques pour une instance quelconque de *AVC*, et de concevoir un moyen efficace pour les repérer.

Théorème 1 *[69] Soit une instance* $(\mathcal{J}, \mathcal{I}, v)$ *du problème AVC. Il existe un ordonnancement réalisable de* $(\mathcal{J}, \mathcal{I}, v)$*, si et seulement si, il existe un* $(s, p)-$*flot maximum dans le graphe correspondant de valeur égale à* $\sum_{i=1}^{|\mathcal{J}|} \frac{w_i}{v}$.

En se basant sur le théorème 1, il est possible d'énoncer le corollaire suivant,

Corollaire 1 *Soit* $(\mathcal{J}, \mathcal{I}, v)$ *une instance réalisable du problème AVC et soit* $G = (V, E)$ *le graphe correspondant. Une tâche* J_c *est critique, si et seulement si, sur chaque chemin* $\pi = (x_c, y_j, p)$ *avec* $I_j \in \mathcal{I}$*, un des deux arcs* (x_c, y_j) *ou* (y_j, p) *est saturé dans tout flot maximum dans* G.

Les lemmes suivants impliquent les notions de tâche et d'instance critique et ils sont très utiles dans la recherche d'un outil pratique de détection des tâches critiques.

Lemme 3 *Si* $(\mathcal{J}, \mathcal{I}, v)$ *est une instance critique du problème AVC, alors il existe au moins une tâche critique* $J_i \in \mathcal{J}$.

Preuve Soit $G = (V, E)$ le graphe correspondant à l'instance $(\mathcal{J}, \mathcal{I}, v)$. Étant donné qu'il s'agit d'une instance critique, il existe une (s, p)-coupe minimale C du graphe G qui contient soit un arc (x_i, y_j) ou un arc (y_j, p), pour une certaine tâche $J_i \in \mathcal{J}$ et un certain intervalle $I_j \in \mathcal{I}$. En effet,

dans le cas contraire la seule (s, p)-coupe minimale serait celle constituée de tous les arcs de type (s, x_i), et toutes les (s, p)-coupes contenant des arcs de type (x_i, y_j) ou de type (y_j, p) auront des capacités strictement supérieures à la coupe minimale, *i.e.* strictement supérieures à $\sum_{J_i \in \mathcal{J}} \frac{w_i}{v}$. Or, ceci implique la possibilité de réduire la vitesse v à une vitesse $v - \epsilon$, pour une certaine valeur $\epsilon > 0$ bien choisie, et telle que le graphe correspondant à l'instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$ admet un (s, p)-flot maximum égal à $\sum_{J_i \in \mathcal{J}} \frac{w_i}{v-\epsilon}$, ce qui contredit le fait que l'instance $(\mathcal{J}, \mathcal{I}, v)$ est critique.

Considérons maintenant la coupe C définie précédemment. Cette coupe contient au moins un arc de type (x_i, y_j) ou de type (y_j, p) et elle ne peut donc pas contenir tous les arcs de type (s, x_i) car dans ce cas elle aura une capacité strictement supérieure à la (s, p)-coupe qui contient uniquement les arcs (s, x_i) ce qui contredit la minimalité de C. Il existe donc au moins un arc (s, x_c) ne faisant pas partie de la coupe C.

En se basant donc sur la définition d'une (s, p)-coupe, on conclut que la coupe C contient un arc appartenant à chaque chemin de type (x_c, y_j, p), pour $I_j \in \mathcal{I}$. Ainsi, puisque tout (s, p)-flot maximum sature la coupe C et par définition, la tâche J_c représentée par le noeud x_c est critique.

□

Remarquons que si $(\mathcal{J}, \mathcal{I}, v)$ est une instance critique, alors tout instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$ pour $\epsilon > 0$ quelconque n'est pas réalisable dans le sens où la vitesse $v - \epsilon$ n'est pas suffisante pour exécuter toutes les tâches entièrement. Bien que les tâches critiques ont été définies dans le cas d'instances réalisables, nous nous proposons d'étendre cette notion dans le cas d'instances non réalisables et pour lesquelles nous définissons les tâches critiques exactement de la même manière que pour les instances réalisables.

Soit une instance critique $(\mathcal{J}, \mathcal{I}, v)$ du problème *AVC* et soit G le graphe correspondant. Nous allons proposer, par la suite, un moyen simple et efficace pour identifier les tâches critiques de cette instance en utilisant le graphe G' qui correspond à l'instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$ pour une valeur $\epsilon > 0$ bien choisie. En effet, cette valeur ϵ est définie de façon à ce que les deux instances $(\mathcal{J}, \mathcal{I}, v)$ et $(\mathcal{J}, \mathcal{I}, v - \epsilon)$ possèdent exactement les mêmes tâches critiques. Dans le lemme suivant, nous

prouvons l'existence d'une telle valeur.

Lemme 4 *Soit une instance critique* $(\mathcal{J}, \mathcal{I}, v)$ *du problème AVC. Il existe une constante* $\epsilon > 0$ *telle que l'instance non réalisable* $(\mathcal{J}, \mathcal{I}, v - \epsilon)$ *et l'instance critique* $(\mathcal{J}, \mathcal{I}, v)$ *possèdent exactement les mêmes tâches critiques.*

Preuve Dans le but de prouver ce lemme, nous allons construire un ensemble de valeurs possibles de ϵ tel qu'une tâche J_i soit critique dans l'instance $(\mathcal{J}, \mathcal{I}, v)$ si et seulement si elle est critique dans l'instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$.

L'instance $(\mathcal{J}, \mathcal{I}, v)$ étant critique, d'après le lemme 3, elle contient au moins une tâche critique. Dans le cas où toutes les tâches sont critiques, il existe dans le graphe G correspondant à l'instance $(\mathcal{J}, \mathcal{I}, v)$ une (s, p)-coupe minimale C contenant exactement un arc pour chaque chemin de type (x_i, y_j, p). Pour tout $\epsilon > 0$, la coupe C est aussi minimale dans le graphe G' qui correspond à l'instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$. En effet, la seule différence entre les graphes G' et G réside dans l'augmentation des capacités des arcs de type (s, x_i) dans le graphe G', *i.e.* la capacité passe de w_i/v à $w_i/v - \epsilon$ pour chaque tâche $J_i \in \mathcal{J}$. Ainsi, toutes les tâches dans $\mathcal{J}$ sont aussi critiques dans l'instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$, et le lemme est prouvé dans ce premier cas.

Supposons maintenant qu'il existe au moins une tâche non critique J_i dans l'instance $(\mathcal{J}, \mathcal{I}, v)$. Ainsi, il existe au moins un (s, p)-flot maximum dans le graphe G correspondant à la tâche J_i et que l'on note par $\mathcal{F}_i$ tel qu'il existe au moins un chemin $\pi = (x_i, y_j, p)$ non saturé par $\mathcal{F}_i$ pour un intervalle $I_j \subseteq [r_i, d_i)$ (dans le cas contraire tout (s, p)-flot maximum sature tous les chemins de type (x_i, y_j, p), ce qui implique que la tâche J_i est critique). Pour chaque tâche non critique J_k, nous allons considérer un (s, p)-flot maximum correspondant $\mathcal{F}_k$ tel qu'il existe au moins un chemin non saturé (x_k, y_j, p) que l'on appelle π_k-*chemin*. Il faut remarquer que pour deux tâches non critiques J_k et $J_{k'}$ il est parfois possible que les flots correspondants $\mathcal{F}_k$ et $\mathcal{F}_{k'}$ soient identiques. Ensuite, nous définissons $\eta_k = \min\{c(x_k, y_j) - f_k(x_k, y_j), c(y_j, p) - f_k(y_j, p)\}$ pour chaque tâche non critique J_k, où $f_k(e)$ est la quantité de flot qui passe par l'arc e dans $\mathcal{F}_k$. En effet, η_k est la quantité

supplémentaire qui peut saturer le π_k-*chemin*.

Considérons, maintenant, l'instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$ où la valeur de $\epsilon > 0$ vérifie $\sum_{J_i \in \mathcal{J}} \frac{w_i}{v-\epsilon} - \sum_{J_i \in \mathcal{J}} \frac{w_i}{v} = \sum_{J_i \in \mathcal{J}} \frac{\epsilon w_i}{v(v-\epsilon)} < \min_k\{\eta_k\}$, et soit G' le graphe correspondant.

Nous allons, tout d'abord, montrer que le passage de l'instance $(\mathcal{J}, \mathcal{I}, v)$ à l'instance $(\mathcal{J}, \mathcal{I}, v-\epsilon)$ conserve l'ensemble des tâches non critiques. Pour toute tâche J_k non critique dans l'instance $(\mathcal{J}, \mathcal{I}, v)$, nous définissons le (s, p)-flot $\mathcal{F}'_k$ dans le graphe G' tel que $f'_k(s, x_i) = \frac{w_i}{v}$ pour chaque tâche J_i dont tous les chemins de type (x_i, y_j, p) sont saturés dans le flot $\mathcal{F}_k$, et $f'_k(s, x_i) = \frac{w_i}{v-\epsilon}$ pour les tâches restantes, où $f'_k(e)$ est la quantité de flot traversant l'arc e dans $\mathcal{F}'_k$. Remarquons que la seule différence entre les flots $\mathcal{F}_k$ et $\mathcal{F}'_k$ consiste dans le fait que pour chaque π_i-*chemin*$= (x_i, y_j, p)$ non saturé dans le flot $\mathcal{F}_k$ on a : $f'_k(x_i, y_j) = f_k(x_i, y_j) + \frac{\epsilon w_i}{v(v-\epsilon)}$ et $f'_k(y_j, p) = f_k(y_j, p) + \frac{\epsilon w_i}{v(v-\epsilon)}$. Il est clair que le flot $\mathcal{F}'_k$ est réalisable et maximum. Pour la tâche non critique J_k et le π_k-*chemin*$= (x_k, y_j, p)$, et par définition de la valeur ϵ, on a $f'_k(x_k, y_j) < c(x_k, y_j)$ et $f'_k(y_j, p) < c(y_j, p)$. Donc pour chaque tâche non critique J_i, le π_i-*chemin* reste non saturé dans le flot $\mathcal{F}'_k$ dans G'. Rappelons que pour prouver qu'une tâche J_i est non critique, il suffit de trouver un flot maximum qui ne sature pas un chemin de type (x_i, y_j, p). D'où le fait que le passage de l'instance $(\mathcal{J}, \mathcal{I}, v)$ à l'instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$ conserve les tâches non critiques.

Considérons, maintenant, une tâche critique J_i dans l'instance $(\mathcal{J}, \mathcal{I}, v)$ et montrons que cette tâche est également critique dans l'instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$. Par définition d'une tâche critique, il existe une (s, p)-coupe minimale C dans le graphe G contenant soit un arc (x_i, y_j), soit un arc (y_j, p) pour tout chemin (x_i, y_j, p). D'autre part, puisque l'instance $(\mathcal{J}, \mathcal{I}, v)$ est réalisable, il existe une (s, p)-coupe minimale C' dans le graphe G constituée de tous les arcs (s, x_i). Notons que les coupes C et C' ont la même valeur. Étant donné que le graphe G' correspondant à l'instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$ diffère du graphe G uniquement par des capacités plus grandes sur les arcs de type (s, x_i), il est possible de conclure que la coupe C reste une (s, p)-coupe minimale pour le graphe G' mais pas la coupe C'. Ainsi, pour chaque tâche J_i critique dans G, chaque chemin (x_i, y_j, p) est saturé par tout (s, p)-flot dans le graphe G'. Finalement, si une tâche est critique dans l'instance $(\mathcal{J}, \mathcal{I}, v)$ alors elle reste critique dans l'instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$. $\square$

Lemme 5 *Soit* $(\mathcal{J}, \mathcal{I}, v)$ *une instance critique du problème AVC et soit* G' *le graphe correspondant à l'instance* $(\mathcal{J}, \mathcal{I}, v - \epsilon)$, *pour une valeur* $\epsilon > 0$ *suffisamment petite (conformément au lemme 4). Dans ce cas, toute* (s, p)*-coupe minimale* C' *dans le graphe* G' *est composée de :*

(i) un arc de chaque chemin (x_i, y_j, p) *pour chaque tâche critique* $J_i \in \mathcal{J}$,

(ii) l'arc (s, x_i) *pour chaque tâche non critique* J_i.

Preuve Rappelons tout d'abord que d'après le lemme 4 les instances $(\mathcal{J}, \mathcal{I}, v)$ et $(\mathcal{J}, \mathcal{I}, v - \epsilon)$ possèdent le même ensemble de tâches critiques. Considérons une tâche critique J_i. Chaque chemin (x_i, y_j, p) est saturé par tout $(s, p)-$flot maximum dans le graphe G ou G'. Soit donc $\mathcal{F}'$ un (s, p)-flot maximum dans le graphe G'. Pour la tâche critique J_i, le flot $\mathcal{F}'$ ne sature pas l'arc (s, x_i) car sinon il existe au moins un chemin (x_i, y_j, p) qui n'est pas saturé par au moins un flot maximum dans le graphe G. Ainsi, l'arc (s, x_i) n'appartient à aucune (s, p)-coupe minimale dans G' et donc chaque coupe minimale contient exactement un seul arc sur chaque chemin (x_i, y_j, p).

Soit maintenant une tâche J_i non critique dans les deux instances du problème *AVC*. Il existe donc un (s, p)-flot maximum $\mathcal{F}'$ dans le graphe G' et un chemin (x_i, y_j, p), $I_j \subseteq [r_i, d_i)$, qui n'est pas saturé par $\mathcal{F}'$. Ainsi, les arcs (x_i, y_j) et (y_j, p) n'appartiennent à aucune (s, p)-coupe minimale dans G' et c'est donc l'arc (s, x_i) qui appartient à toute coupe minimale dans G'. □

À partir du lemme 5, il est possible de déduire dans le corollaire suivant un moyen simple pour déterminer l'ensemble des tâches critiques étant donnée une instance critique du problème *AVC*.

Corollaire 2 *Soit une instance critique* $(\mathcal{J}, \mathcal{I}, v)$ *du problème AVC. L'ensemble des tâches critiques est entièrement défini en suivant les deux étapes suivantes :*

i Calculer une (s, p)*-coupe minimale* C *dans le graphe correspondant à l'instance* $(\mathcal{J}, \mathcal{I}, v - \epsilon)$ *pour une valeur* $\epsilon > 0$ *infiniment petite.*

ii Retourner l'ensemble des tâches J_i *tel que la coupe* C *contienne un arc sur chaque chemin partant du noeud* x_i *correspondant à* J_i *jusqu'au noeud puits* p.

1.4.3 L’algorithme *SSM*

Nous nous proposons à présent de donner une description de l’algorithme combinatoire qu’on note "*SSM*" (Speed Scaling avec Migration) pour résoudre d’une façon optimale le problème de minimisation de l’énergie consommée sur des machines identiques avec migration.

Initialement, nous définissons la vitesse v_u telle qu’il existe un ordonnancement réalisable où toutes les tâches d’une instance du problème initial s’exécutent à cette vitesse. L’idée consiste à décroître cette vitesse jusqu’à ne plus être capable de concevoir un ordonnancement réalisable où toutes les tâches sont exécutées à une vitesse commune. Rappelons que pour une vitesse donnée, un ordonnancement réalisable est retourné en calculant un flot maximum pour l’instance du problème *AVC* correspondante. La dernière vitesse garantissant un ordonnancement réalisable est une vitesse critique et il existe au moins une tâche critique qui ne peut être exécutée qu’à une vitesse au moins égale à cette vitesse critique dans tout ordonnancement réalisable. À chaque étape de l’algorithme *SSM*, l’ensemble des tâches critiques est détécté en utilisant le corollaire 2. Ensuite, l’algorithme attribue la vitesse critique à ces tâches et met à jour le nombre de machines disponibles pour chaque intervalle $I_j \in \mathcal{I}$. Cette mise à jour est réalisée à l’aide la coupe minimale décrite dans le corollaire 2 : si cette coupe contient un arc (x_i, y_j) alors nous considérons qu’une machine est utilisée pendant l’intervalle I_j pour l’exécution de la tâche critique correspondante J_i. Par contre, si la coupe minimale contient un arc (y_j, p) alors, durant l’intervalle I_j, toutes les machines disponibles seront résérvées à l’exécution des tâches critiques. Finalement en supprimant les tâches critiques de l’ensemble initial de tâches, nous obtenons un sous-problème et nous définissons pour chaque vitesse l’instance du problème *AVC* correspondante.

Le pseudo-code de l’algorithme *SSM* peut être décrit comme suit,

Algorithme 1: SSM

Données : Une instance du problème de minimisation d'énergie sur des machines identiques avec migration.

Résultat : Un ordonnancement optimal.

début

$v_\ell = \max_{J_i \in \mathcal{J}}\{\delta_i\}$, $v_u = \max_{I_j \in \mathcal{I}}\{\frac{\sum_{J_i \in D_j} w_i}{|I_j|}\}$;

tant que $\mathcal{J} \neq \emptyset$ **faire**

Rechercher par dichotomie sur l'intervalle $[v_\ell, v_u]$ la vitesse minimale $v^\star$ telle que l'instance $(\mathcal{J}, \mathcal{I}, v^\star)$ du problème *AVC* soit réalisable. Calculer un flot maximum dans le graphe correspondant pour vérifier la faisabilité de l'instance ;

Déterminer l'ensemble de tâches critiques $\mathcal{J}^\star$ en utilisant l'instance $(\mathcal{J}, \mathcal{I}, v^\star - \epsilon)$ pour un réel $\epsilon > 0$ infiniment petit (corollaire 2) ;

$v_i := v^\star$ pour toute tâche critique $J_i \in \mathcal{J}^\star$;

Mettre à jour l'ensemble des tâches : $\mathcal{J} = \mathcal{J} \backslash \mathcal{J}^\star$;

Mettre à jour le nombre m_j de machines disponibles pour chaque intervalle I_j à l'aide de la coupe minimale dans le graphe de l'instance $(\mathcal{J}, \mathcal{I}, v^\star - \epsilon)$;

$v_u = v^\star$, $v_\ell = \max_{J_i \in \mathcal{J}}\{\delta_i\}$;

fin

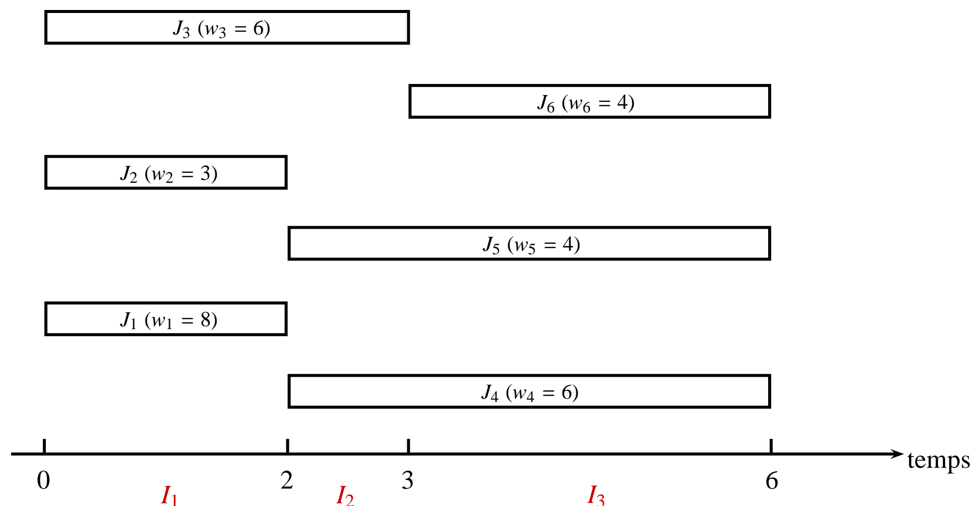
Résoudre d'une façon optimale l'instance obtenue du problème $P|r_i, d_i, pmtn|-$ avec une durée d'exécution égale à $\frac{w_i}{v_i}$ pour toute tâche $J_i \in \mathcal{J}$;

fin

À chaque itération de l'algorithme *SSM*, la vitesse critique et l'ensemble des tâches cri-

tiques sont représentées, respectivement, par les variables $v^\star$ et $\mathcal{J}^\star$. Afin de déterminer la vitesse critique à chaque itération, l'algorithme effectue une recherche binaire en supposant que toutes les tâches non encore traitées seront exécutées à la même vitesse. Il est possible de déterminer les bornes inférieure et supérieure de l'intervalle de recherche d'une façon simple. En effet, la vitesse qui sera attribuée à chaque tâche ne peut pas être inférieure à sa densité si elle doit être exécutée en sa totalité. Ainsi, étant donné un ensemble de tâches $\mathcal{J}$ il n'existe aucun ordonnancement réalisable pouvant exécuter toutes les tâches à une vitesse $v < \max_{J_i \in \mathcal{J}}\{\delta_i\}$. D'autre part, observons que pour le même ensemble $\mathcal{J}$, il est toujours possible de construire un ordonnancement réalisable qui exécute toutes les tâches à une vitesse unique égale à $v = \max_{I_j \in \mathcal{I}}\{\frac{\sum_{J_i \in D_j} w_i}{|I_j|}\}$ et qui constitue donc une borne supérieure dans l'intervalle de recherche. Dans l'étape suivante de l'algorithme *SSM*, ces bornes seront mis à jour et la valeur de la borne supérieure de l'intervalle sera substituée par la vitesse critique calculée à l'étape courante.

Soit l'exemple suivant : nous considérons l'ensemble de tâches $\mathcal{J} = \{J_1, J_2, J_3, J_4, J_5, J_6\}$ et l'ensemble d'intervalles $\mathcal{I} = \{I_1, I_2, I_3\}$. La figure 1.4 détaille les paramètres de cette instance (dates d'arrivée, dates d'échéance et quantités de travail). Aussi nous supposons que deux machines M_1 et M_2 sont à disposition pour l'exécution des tâches.



F . 1.4 – Exemple d'instance du problème de minimisation d'énergie avec migration.

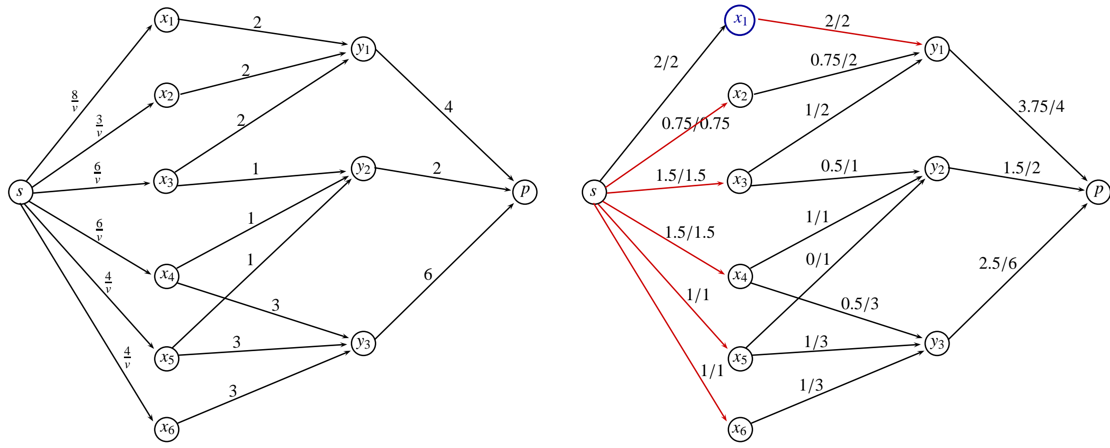
En appliquant l'algorithme *SSM*, celui-ci résout le problème en trois itérations et nous obtenons l'ensemble de vitesses critiques suivantes $\{v_1 = 4, v_2 = 3, v_3 = 2\}$, où chaque valeur v_i représente la vitesse critique retournée à l'issue de la $i^{ème}$ itération. Dans la figure 1.5, nous résumons le déroulement de l'algorithme sur l'instance en question : la figure 1.5(a) modélise l'instance $(\mathcal{J}, \mathcal{I}, v)$ du problème *AVC* en terme de flots pour $v > 0$. Les figures 1.5(b), 1.5(c) et 1.5(d) présentent les différentes itérations de l'algorithme. Dans chacun des flots, l'ensemble des arcs colorés en rouge représentent la coupe minimale dans le graphe correspondant à l'instance $(\mathcal{J}, \mathcal{I}, v - \epsilon)$ pour $\epsilon > 0$ infiniment petit, et l'ensemble des noeuds x_i colorés en bleu représentent l'ensemble des tâches J_i critiques trouvées à la fin de l'itération courante. Il faut noter aussi que suite à la mise à jour du nombre de machines disponibles durant chaque intervalle lorsqu'on passe d'une itération à la suivante, la capacité des arcs de type (y_j, p) est susceptible de changer. Par exemple lors du passage de la première itération (figure 1.5(b)) à la deuxième itération (figure 1.5(c)), la machine M_1 ou M_2 est utilisée durant l'intervalle I_1 par la tâche critique dévoilée J_1, et ainsi une seule machine est désormais disponible pendant I_1, ce qui fait passer la capacité de l'arc (y_1, p) de la valeur 4 à la valeur 2.
À la fin, l'algorithme attribue la vitesse v_1 à la tâche J_1, la vitesse v_2 aux tâches J_2 et J_3 et la vitesse v_3 aux tâches J_4, J_5 et J_6. Ensuite, il suffit de résoudre l'instance du problème $P|r_i, d_i, pmtn|-$ correspondante.

Dans le théorème suivant, nous allons prouver que l'algorithme *SSM* retourne un ordonnancement vérifiant les propriétés énumérées dans le lemme 1 et il est donc optimal d'après le lemme 2.

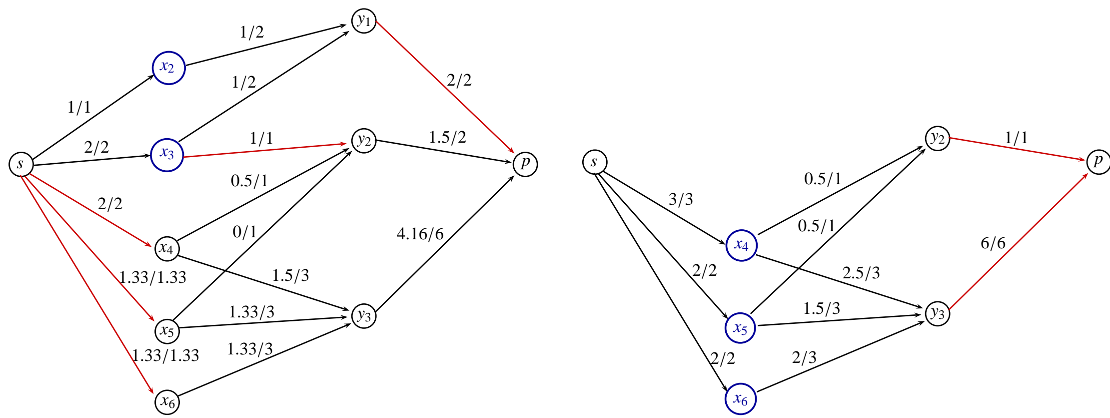
Théorème 2 *L'algorithme SSM retourne un ordonnancement optimal pour toute instance du problème.*

Preuve Tout d'abord, il est clair que l'algorithme *SSM* attribue une vitesse constante à chaque tâche en une seule itération avant que celle-ci soit supprimée de l'instance. Ainsi, la première propriété du lemme 1 est vérifié.

À chaque itération de l'algorithme, une vitesse constante est attribuée à un sous ensemble



(a) (b)



(c) (d)

F . 1.5 – Exemple d'application de l'algorithme *SSM*.

de tâches. En plus, durant un certain nombre d'intervalles, le nombre de machines disponibles sera réduit puisqu'elles seront consacrées à l'exécution de ces tâches. Soit la $k^{ème}$ itération de l'algorithme, nous allons noter par $\mathcal{J}^{(k)}$ et $\mathcal{I}^{(k)}$ respectivement l'ensemble des tâches auquelles l'algorithme n'a pas encore attribué de vitesse, et l'ensemble des intervalles pendant lesquels au moins une machine est encore disponible, *i.e.* non utilisée par d'autres tâches critiques pendant les itérations précédentes. On note, également, par $s_l^{(k)}$ et $s_u^{(k)}$ les bornes inférieure et supérieure de l'intervalle de recherche de la vitesse critique au cours de la $k^{ème}$ itération. Par ailleurs, pour une vitesse donnée $v \in [s_l^{(k)}, s_u^{(k)}]$, on note par $G^{(k)}$ le graphe correspondant à l'instance $(\mathcal{J}^{(k)}, \mathcal{I}^{(k)}, v)$ du problème *AVC*.

Considérons maintenant la propriété $4(i)$ du lemme 1. Soit un ordonnancement retourné par l'algorithme *SSM* et supposons qu'il existe deux tâches J_i et J_ℓ disponibles durant un intervalle I_j et telles que $0 < t_{i,j} < |I_j|$ et $0 < t_{\ell,j} < |I_j|$. Il s'agit donc de prouver que ces deux tâches sont exécutées à la même vitesse ce qui revient à prouver que cette vitesse leur a été attribuée à la fin d'une unique itération. Supposons donc que la tâche J_i devient critique à la fin de l'itération k, ce qui implique qu'il existe une (s, p)-coupe C dans le graphe $G^{(k)}$ contenant l'arc (x_i, y_j) ou l'arc (y_j, p). Du fait que $0 < t_{i,j} < |I_j|$, il existe un (s, p)-flot maximum dans le graphe $G^{(k)}$ tel que $0 < f(x_i, y_j) < |I_j|$. L'arc (x_i, y_j) n'est donc pas saturé, et on déduit que l'arc (y_j, p) fait partie de la coupe C. Ainsi, tout flot maximum sature l'arc (y_j, p), ce qui implique qu'à la fin de l'itération k, aucune machine ne sera disponible durant l'intervalle I_j au cours des itérations suivantes, *i.e.* $k+1, k+2, etc$. Puisque $0 < t_{\ell,j} < |I_j|$, l'attribution d'une vitesse à la tâche J_ℓ ne peut pas avoir lieu pendant une itération postérieure à l'itération k. De la même façon, nous montrons que l'attribution d'une vitesse à la tâche J_i ne peut pas se faire à une itération postérieure à l'itération k, ce qui fait que l'algorithme attribue la même vitesse aux tâches J_i et J_ℓ à la fin de l'itération k.

Considérons maintenant la propriété $4(ii)$ du lemme 1 et soit une tâche J_i et un intervalle $I_j \subset [r_i, d_i)$ tels que $t_{i,j} = 0$. Supposons que la tâche J_i devient critique durant la $k^{ème}$ itération dans l'algorithme *SSM*. Dans le graphe $G^{(k)}$ correspondant, il y a deux cas possibles : soit que le noeud y_j ne fait pas partie des noeuds du graphe, ce qui implique qu'aucune machine n'est dispo-

nibles durant l'intervalle I_j au début de l'itération k (la totalité des machines utilisées pendant les itérations précédentes), ou alors que l'arc (y_j, p) appartient à une (s, p)-coupe minimale dans $G^{(k)}$. Dans l'hypothèse qu'aucun de ces deux cas n'était vrai, l'arc (y_j, p) aurait fait partie de l'ensemble des arcs du graphe $G^{(k)}$ sans qu'il appartienne à une (s, p)-coupe minimale. Dans ce cas, tous les arcs (x_ℓ, y_j) tels que $I_j \subseteq [r_i, d_j)$ feront partie d'une (s, p)-coupe minimale, et en particulier l'arc (x_i, y_j) sera saturé par tout (s, p)-flot maximum. Or ceci est une contradiction étant donné qu'il existe au moins un (s, p)-flot maximum tel que $f(x_i, y_j) = 0$ puisque $t_{i,j} = 0$. Ainsi, dans les deux cas (y_j n'appartient pas à $G^{(k)}$ ou (y_j, p) appartient à une coupe minimale), aucune machine ne sera diponible durant l'intervalle I_j après la $k^{ème}$ itération de l'algorithme, ce qui fait que toutes les tâches J_ℓ avec $t_{\ell,j} > 0$ possèdent des vitesses d'exécution supérieures ou égales à la vitesse de la tâche J_i.

Nous considérons à présent la propriété 4(iii) du lemme 1 et soit une tâche J_i telle que $t_{i,j} = |I_j|$ et telle que l'algorithme lui attribue une vitesse à la $k^{ème}$ itération. Il est clair que d'après la propriété 4(i), l'itération k ne peut pas avoir lieu après une itération k' à la suite de laquelle l'algorithme attribue une vitesse à une tâche J_ℓ qui vérifie $0 < t_{\ell,j} < |I_j|$. En effet, il a été démontré qu'à la fin d'une telle itération k', aucune machine n'est encore disponible durant l'intervalle I_j. Ainsi la vitesse s_i attribuée à la tâche J_i est au moins égale à la vitesse s_ℓ attribuée à toute tâche J_ℓ vérifiant $0 < t_{\ell,j} < |I_j|$. Dans le cas où $t_{\ell,j} = 0$, d'après la propriété 4(ii), l'algorithme n'attribue pas une vitesse à la tâche J_i après avoir traiter la tâche J_ℓ. Donc, la vitesse s_i est aussi dans ce cas supérieure ou égale à s_ℓ.

Afin de montrer maintenant que toute solution retournée par l'algorithme *SSM* vérifie la $3^{ème}$ propriété du lemme 1, nous considérons une instance telle qu'il existe un intervalle I_j vérifiant $|D_j| \leqslant m$ et soit une tâche quelconque telle que $I_j \subseteq [r_i, d_i)$ et telle que la tâche devient critique à la $k^{ème}$ itération de l'algorithme. Montrer que $t_{i,j} = |I_j|$ dans la solution retournée revient à montrer que dans le graphe $G^{(k)}$, tout (s, p)-flot maximum sature l'arc (x_i, y_j). Supposons, par contradiction qu'il existe un (s, p)-flot maximum $\mathcal{F}$ dans $G^{(k)}$ tel que $f(x_i, y_j) < |I_j|$ et tel que $f(x_\ell, y_j) \leqslant |I_j|$ pour tout $J_\ell \in D_j \backslash \{J_i\}$. Dans ce cas, on a $f(y_j, p) = \sum_{J_\ell \in D_j} f(x_\ell, y_j) < |D_j||I_j| \leqslant m|I_j|$, et donc le

flot $\mathcal{F}$ ne sature ni l'arc (y_j, p) ni l'arc (x_i, y_j), ce qui contredit le fait que la tâche J_i est critique. On conclut que le temps d'exécution de toute tâche disponible pendant l'intervalle I_j est égal à $|I_j|$.

Nous prouvons finalement que l'algorithme *SSM* retourne des solutions qui vérifient la deuxième propriété du lemme 1. Tout d'abord, étant donné un intervalle quelconque I_j tel que $|D_j| \leqslant m$, d'après la $3^{ème}$ propriété il est clair que $\sum_{J_i \in D_j} t_{i,j} = |D_j||I_j|$ et la propriété est prouvée dans ce cas. Ensuite, supposons que $\min\{|D_j|, m\} = m$. Dans ce cas il n'est pas possible d'avoir $\sum_{J_i \in D_j} t_{i,j} > m|I_j|$ puisque l'algorithme *SSM* est basé sur le calcul de flots dans des graphes où la capacité de chaque arc de type (y_j, p) ne peut pas dépasser $m|I_j|$. Maintenant, supposons par contradiction que $\sum_{J_i \in D_j} t_{i,j} < m|I_j|$. Il existe donc au moins une tâche $J_\ell \in D_j$ telle que $t_{\ell,j} < |I_j|$. Il existe aussi au moins une machine qui n'est pas utilisée dans la solution au cours de l'intervalle I_j, ainsi l'arc (y_j, p) est présent dans tout graphe $G^{(k)}$ pour toute itération k de l'algorithme. Soit l'itération h pendant laquelle la tâche J_ℓ devient critique. Étant donné que $t_{\ell,j} < |I_j|$ et qu'il existe au moins une machine non utilisée durant I_j alors aucun flot maximum dans $G^{(h)}$ ne peut saturer le chemin (x_ℓ, y_j, p) ce qui contredit le fait que la tâche J_ℓ est critique.

Finalement, l'algorithme identifie correctement les tâches critiques à chaque itération à cause du lemme 5. □

Complexité de *SSM*. Nous analysons maintenant la complexité de l'algorithme *SSM*. À chaque itération, l'algorithme attribue la vitesse critique trouvée par recherche binaire à au moins une tâche. Ceci est garanti par le lemme 3 qui stipule que pour chaque instance critique, il existe au moins une tâche critique. Ainsi, l'algorithme *SSM* effectue au plus n itérations pour une instance $\mathcal{J}$ du problème, où $|\mathcal{J}| = n$.

Il s'agit de réaliser dans chacune des itérations une recherche binaire pour trouver la vitesse critique courante. Cette recherche est basée sur le calcul de $O(log(P))$ flots maximums, où P représente le nombre de vitesses candidates. En effet, la valeur P peut être calculée en divisant la taille de l'intervalle de recherche $[v_\ell, v_u]$ par la précision désirée. Il est clair que cette valeur est majorée par P_{max} qui correspond à la valeur calculée de la même manière sur l'intervalle $[\max_{J_i \in \mathcal{J}}\{\delta_i\}, \max_{I_j \in \mathcal{I}}\{\frac{\sum_{J_i \in D_j} w_i}{|I_j|}\}]$.

Par ailleurs, notons par $C(x,y)$ le temps nécessaire pour calculer un flot maximum dans un graphe en couche contenant x sommets et y arcs. Ainsi, la complexité totale de l'algorithme *SSM* est en $O(nC(n,n^2)log(P_{max}))$ étant donné que dans les graphes correspondants aux différentes instances du problème *AVC*, le nombre de sommets est en $O(n)$ et le nombre d'arcs est en $O(n^2)$.

1.5 C

Ce chapitre a été dédié à l'étude du problème d'ordonnancement sur des machines parallèles avec préemption et migration des tâches dans le but de minimiser la consommation de l'énergie dans le modèle d'adaptation de la vitesse (*Speed Scaling*).
Tout d'abord, il était question de trouver la structure de toute solution optimale du problème. Nous avons donc déduit des conditions nécessaires d'optimalité en formulant le problème en terme de progarmme convexe et en appliquant les conditions de Karush-Kuhn-Tucker (conditions KKT). Ensuite, nous avons montré que ces critères étaient suffisantes pour l'optimalité.
Guidés par cette structure optimale, nous avons défini les notions d'instance et de tâche critiques et nous avons fait usage d'une réduction du problème en un problème de flot maximum pour construire un algorithme combinatoire qu'on a noté *SSM* et qui retourne une solution optimale à toute instance du problème en un temps polynomial.

Dans le chapitre 3, nous allons étudié un problème d'optimisation avec un double objectif à savoir la minimisation de l'énergie et du temps de complétude moyen des tâches. Dans ce contexte, il semble que l'algorithme *SSM* peut être exploité pour résoudre efficacement quelques problèmes d'ordonnancement multi-critères dont l'un des objectifs est la minimisation de l'énergie consommée. Nous citons par exemple le problème qui consiste à ordonnancer sous le modèle d'adaptation de la vitesse (*Speed Scaling*) un ensemble de tâches avec des dates d'arrivées r_i et des quantités de travail positifs w_i sur m machines identiques dans le but de minimiser le temps d'exécution total (*makespan*) sans que l'enérgie consommée ne dépasse un seuil E fixé à l'avance. Si la migration et la préemption sont autorisées, alors il est possible de résoudre ce problème comme suit : supposons que C_ℓ et C_u représentent respectivement une borne inférieure et une

borne supérieure sur le temps d'exécution total. En fixant une valeur quelconque $C \in [C_\ell, C_u]$, il est possible de savoir s'il existe un ordonnancement réalisable dont le temps d'exécution total soit égal à C et tel que l'énergie consommée ne dépasse pas la borne E. Il suffit, en effet, d'attribuer à toutes les tâches une date d'échéance fictive commune égale à C et de lancer l'algorithme *SSM* sur l'instance obtenue. Si l'ordonnancement construit dépense une énergie $E_{SSM} \leqslant E$ alors il constitue une solution réalisable du problème. Il s'agit donc de réaliser une recherche binaire dans l'intervalle $[C_\ell, C_u]$ pour trouver la plus petite valeur du *makespan* telle qu'il existe une solution réalisable du problème. Pour une instance donnée, une borne inférieure sur le *makespan* peut être définie par $C_\ell = \frac{1}{m}(\frac{W^\alpha}{E})^{\frac{1}{\alpha-1}}$ où $W = \sum_{J_i} w_i$. Cette borne est obtenue en fixant la vitesse d'exécution de toutes les tâches à $\frac{W}{m.x}$ et en utilisant chaque machine pendant une durée égale à x. Par ailleurs, la borne supérieure sur le temps d'exécution total peut être définie simplement par $C_u = \max_{J_i}\{r_i\} + (\frac{W^\alpha}{E})^{\frac{1}{\alpha-1}}$.
D'autre part, il est possible d'utiliser la même technique pour résoudre le problème de minimisation du retard maximum avec un budget E d'énergie. Dans ce problème, on attribue à chaque tâche J_i une date de livraison $q_i \geqslant 0$ et étant donné un ordonnancement quelconque le retard de J_i est donné par $L_i = C_i + q_i$ où C_i est le temps de fin d'exécution de la tâche J_i. Pour une valeur quelconque $L > 0$, il est possible d'utiliser l'algorithme *SSM* en attribuant à chaque tâche J_i une date d'échéance $d_i = L - q_i$ pour savoir s'il existe ou non un ordonnancement réalisable dont l'énergie consommée ne dépasse pas le seuil E. En effet, dans l'ordonnancement retourné par l'algorithme *SSM*, le retard relatif à chaque tâche J_i vérifie $L_i = C_i + q_i \leqslant d_i + q_i = L$. Il suffit donc de définir un intervalle de recherche $[L_\ell, L_u]$ et d'effectuer une recherche binaire pour trouver la plus petite valeur possible L telle qu'il existe un ordonnancement rélaisable respectant le budget d'énergie. Les bornes de l'intervalle de recherche peuvent être définies par $L_\ell = 0$ et $L_u = \max_{J_i}\{r_i\} + (\frac{W^\alpha}{E})^{\frac{1}{\alpha-1}} + \max_{J_i}\{q_i\}$.

Une perspective intéressante de ce travail serait d'augmenter le modèle d'adaptation de la vitesse (*Speed Scaling*) par l'introduction d'un ou plusieurs états de veille (voir chapitre 2) où le

système pourrait passer en un mode de basse consommation d'énergie, voire un mode d'hibernation où la consommation est nulle.

Finalement, il serait intéressant de savoir si la formulation en terme de flots pourrait conduire à résoudre éfficacement d'autres problèmes faisant intervenir l'énergie consommée.

C 2

O ´ ’´ :

’

’´ `

2.1 I

Comme nous l’avons vu dans le chapitre précédent, le mécanisme d’adaptation dynamique de la vitesse d’une machine (*"Speed Scaling"*) permet de conserver l’énergie consommée qui dépend de la vitesse d’exécution : plus la vitesse est grande plus l’énergie consommée est importante. Il est ainsi possible d’imaginer des modèles d’ordonnancement visant à minimiser l’énergie tout en garantissant une certaine qualité de service.

Dans le modèle étudié dans le chapitre précédent, nous avons supposé que si la vitesse d’une machine est nulle pendant une période donnée, alors l’énergie dépensée par cette machine pendant cette période est aussi nulle. Cependant, dans un modèle plus réaliste, il existe toujours une consommation d’énergie au niveau de la machine même dans le cas où la vitesse est nulle. Il s’agit d’une puissance statique dissipée, constamment présente et ce en présence ou en absence de tâches en exécution. Par ailleurs, un ou plusieurs états de veille où le système d’exploitation peut choisir de suspendre l’activité de la machine sont aussi disponibles. Le passage à un état de veille a lieu si la machine reste inactif, *i.e.* n’exécute pas des tâches, pendant une certaine période

seuil que le système peut modifier et fixer à l'avance. Pendant ces états de veille, le taux de consommation de l'énergie statique est minimal, voire nul mais le retour à un état actif nécessite une quantité fixe d'énergie.

Dans le but de minimiser l'énergie consommée, une approche algorithmique consiste à utiliser le mécanisme de passage dans un état de veille en essayant d'y rester le plus longtemps possible et de minimiser le nombre de passage dans l'état actif [20]. Une autre approche combine ce mécanisme avec la technique d'adaptation de la vitesse de la machine (*"Speed Scaling"*). Dans cette approche, il est parfois plus utile d'accélérer l'exécution des tâches afin de basculer plus rapidement dans un état de veille et dépenser ainsi une énergie minimale ou nulle.

2.2 D´ `

Une instance du problème de minimisation d'énergie sur une seule machine avec un état de veille consiste en un ensemble de tâches $\mathcal{J} = \{J_1, \ldots, J_n\}$, tel que chaque tâche J_i, $1 \leq i \leq n$ soit caractérisée par une quantité de travail w_i, une date d'arrivée r_i et une date d'échéance d_i, et tel que la tâche J_i ne puisse être ordonnancée que dans l'intervalle $[r_i, d_i)$. Une instance de ce problème est dite agréable s'il est possible de rénuméroter les tâches de telle sorte à ce que les dates d'arrivée et les dates d'échéance soient données dans un ordre croissant, *i.e.* étant données deux tâches J_k et $J_{k'}$, $k < k'$ implique que $r_k \leq r_{k'}$ et $d_k \leq d_{k'}$. Nous introduisons, dans ce cas un ordre total $"\leq"$ tel que pour tout couple de tâches $(J_k, J_{k'})$, $J_k \leq J_{k'}$ si et seulement si $k \leq k'$.

Un ordonnancement est défini par 3 fonctions :

$$\begin{aligned} \text{mode} &: \mathbb{R} \rightarrow \{On, Off\} \\ \text{vitesse} &: \mathbb{R} \rightarrow \mathbb{R}^+ \\ \text{tâche} &: \mathbb{R} \rightarrow \{vide, J_1, \ldots, J_n\}, \end{aligned}$$

où les fonctions *mode*, *vitesse* et *tâche* déterminent respectivement l'état du système, la vitesse

de la machine et la tâche en cours d'exécution à chaque instant de l'ordonnancement. Ces trois fonctions vérifient les propriétés suivantes :

1. $\forall t$: vitesse$(t) > 0 \Rightarrow$ mode$(t) = On$.
2. $\forall t$: vitesse$(t) = 0 \Leftrightarrow$ tâche$(t) = vide$.
3. $\forall t$: tâche$(t) = J_i, J_i \neq vide \Rightarrow t \in [r_i, d_i)$.
4. $\forall J_i \neq vide$: $\int$ vitesse(t)d$t = w_i$ où l'intégrale est appliquée sur tous les instants t tels que tâche$(t) = J_i$.
5. pour tout instant t, il existe un intervalle de longueur non nulle I tel que $t \in I$ et durant lequel l'ordonnancement est constant, *i.e.* la vitesse reste constante. De plus I est de la forme $(-\infty, u)$, $[t', u)$ ou $[t', +\infty)$ pour des instants donnés t' et u.

La dernière propriété est une hypothèse simplificatrice qui permet d'éviter qu'un ordonnancement ne dégénère. Un exemple d'un tel ordonnancement peut être décrit comme suit :

$$\text{vitesse}(t) = \begin{cases} 0 & \text{si } t \notin [0, 1) \\ 1 & \text{si } t \in [0, 1) \cap \mathbb{R} \backslash \mathbb{Q} \\ 2 & \text{si } t \in [0, 1) \cap \mathbb{Q}. \end{cases}$$

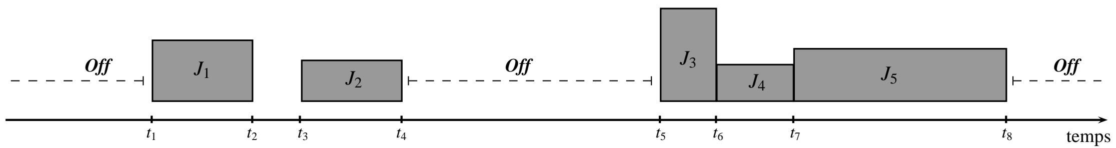
Les différents états du système peuvent être résumés comme suit : si à un instant donné t on a tâche$(t) = vide$, alors on dit que la machine est au repos. Dans ce cas, la machine peut être soit en état de marche et dans ce cas mode$(t) = On$, soit éteinte et dans ce cas mode$(t) = Off$. Dans le cas où tâche$(t) \neq vide$, alors la machine, dite dans ce cas active, exécute la tâche tâche(t) et elle est, ainsi, en état de marche, *i.e.* mode$(t) = On$.

Le coût de l'énergie consommée d'un ordonnancement quelconque est caractérisé par trois paramètres : un exposant $\alpha > 1$, un coût de mise en marche de la machine $L > 0$ et un coût de dissipation d'énergie statique ou de fonctionnement représenté par un taux de consommation

$g > 0$. Par ailleurs, ce coût d'énergie possède deux composantes :

(i) Le *coût d'exécution*, à savoir l'énergie consommée pendant tous les instants t tel que tâche$(t) = J_i \neq$ *vide*, *i.e.* machine active. Ce coût est défini par $c_{\text{vitesse}} = \int \text{vitesse}(t)^{\alpha} \mathrm{d}t$, où l'intégrale est appliquée sur toute la durée de l'ordonnancement.

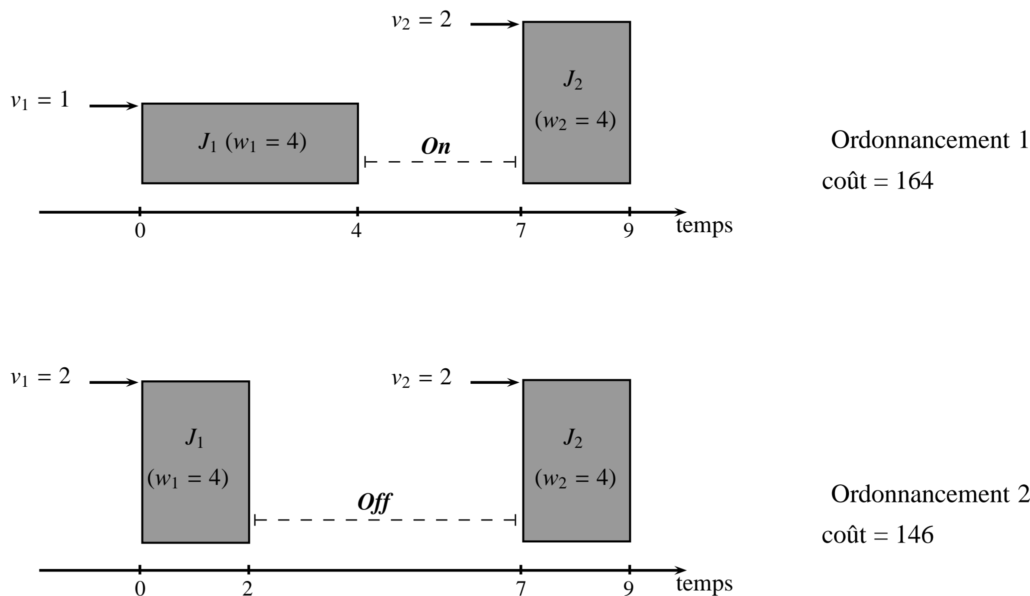
(ii) Le *coût du mode du système*, il s'agit du coût de l'énergie statique dépensée lorsque le système est en état de marche, *i.e.* pour tout instant t tel que mode $= On$, plus le coût de l'énergie nécessaire pour la mise en marche de la machine à chaque fois que le système passe d'un mode *Off* à un mode *On*. Un ordonnancement avec la propriété (5) partitionne l'axe du temps en une séquence S d'intervalles disjoints et maximaux au sens de l'inclusion, tel que mode$(t) = On$ si et seulement si $t \in \cup S = \cup_{I_k \in S} I_k$. La séquence S est dite le *support* de l'ordonnancement, et la consommation d'énergie générée par ce support constitue le coût du mode du système défini par $c_{\text{mode}} = L \cdot (|S| + 1) + g \cdot |\cup S|$. Il est à noter que dans ce calcul, nous comptons le coût de mise en marche L pour les deux intervalles semi-infinis qui bornent le support S. Dans l'exemple de la figure 2.1 nous présentons un ordonnancement de 5 tâches sur une machine ainsi que le coût correspondant en terme d'énergie consommée.



F . 2.1 – Ordonnancement d'une instance composée de 5 tâches. Nous supposons que chaque tâche J_i est exécutée à une vitesse fixe v_i. La machine est en mode *On* durant les intervalles $[t_1, t_4)$ et $[t_5, t_8)$, et elle est en mode *Off* ailleurs. Ainsi l'énergie totale consommée dans cet ordonnancement se décompose en : $c_{\text{vitesse}} = v_1^{\alpha}.(t_2 - t_1) + v_2^{\alpha}.(t_4 - t_3) + v_3^{\alpha}.(t_6 - t_5) + v_4^{\alpha}.(t_7 - t_6) + v_5^{\alpha}.(t_8 - t_7)$ et $c_{\text{mode}} = 3.L + g.((t_4 - t_1) + (t_8 - t_5))$.

Ainsi, le coût total de l'énergie consommée est donné par $c_{\text{vitesse}} + c_{\text{mode}}$ et le problème

étudié consiste à trouver un ordonnancement réalisable qui minimise le coût d'énergie pour une instance agréable. Notons que pour minimiser l'énergie dans ce modèle, il n'est pas toujours optimal de minimiser les vitesses d'exécution des tâches le plus possible mais il s'agit plutôt de trouver un compromis entre le coût d'exécution c_{vitesse} et le coût relatif au mode du système c_{mode}. Cette situation est illustrée dans la figure 2.2 où le modèle de coût de l'énergie consommée est caractérisé par les paramètres suivants : $\alpha = 3$, $L = 50$ et $g = 16$. Nous considérons par ailleurs une instance avec 2 tâches : la tâche J_1 avec $w_1 = 4$, $r_1 = 0$ et $d_1 = 4$, et la tâche J_2 avec $w_2 = 4$, $r_2 = 7$ et $d_2 = 9$, et nous montrons deux ordonnancements différents tels que le deuxième soit meilleur que le premier en terme d'énergie dépensée.



F . 2.2 – Exemple d'ordonnancement où l'augmentation de la vitesse d'exécution n'entraîne pas une augmentation dans la consommation d'énergie. Nous supposons que toutes les tâches sont exécutées à des vitesses constantes. Dans le premier ordonnancement la dépense d'énergie dans l'intervalle $[4, 7)$ est égale à $3.g < L$, donc le système reste en mode *On*, alors que dans le deuxième ordonnancement cette dépense devient $5.g \geqslant L$, d'où le passage en mode *Off* serait plus économique.

2.3 T

Notre problème d'ordonnancement pour des instances générales *(non agréables)* a été étudié dans [48], où les auteurs ont fait référence à deux sous problèmes étudiés et résolus indépendamment. Le premier sous problème ne considère pas le mécanisme d'adaptation de la vitesse en se limitant à des vitesses nulles ou unitaires en fonction du mode dans lequel se trouve le système. Dans ce cas, il est clair que le but essentiel est de minimiser c_{mode} uniquement. Ce premier sous problème a été résolu en $O(n^5)$ en utilisant la programmation dynamique dans [20]. Pour les instances agréables la complexité à été améliorée en $O(n^2)$ [9]. Le second sous problème, en revanche, ne considère qu'un seul état du système, à savoir l'état *On*, et restreint le coût de dissipation d'énergie statique de fonctionnement à $g = 0$. L'objectif ici étant de minimiser uniquement c_{vitesse}. Ce sous problème correspond au problème résolu par le célébre algorithme glouton de Yao, Demers et Shenker [83] en $O(n^3)$. La complexité de cet algorithme que nous notons par YDS a passé à $O(n^2 logn)$ dans [63] avant d'être améliorée en $O(n^2)$ pour des instances agréables [81].

Afin d'être exhaustif, nous résumons ici l'algorithme *YDS* [83] . À chaque étape de l'algorithme, on séléctionne l'intervalle $I^\star$ de densité maximale parmi les $O(n^2)$ intervalles de la forme $[r_i, d_j]$. La densité d'un intervalle est définie par le ratio $v = W/(d_j - r_i)$, où W est la quantité de travail totale de toutes les tâches J_k telles que $[r_k, d_k] \subseteq [r_i, d_j]$. Ensuite, toutes ces tâches sont ordonnancées dans $I^\star$ à la vitesse v en donnant la priorité à la tâche ayant la plus courte date d'échéance. Il faut préciser qu'aucune tâche ne rate sa date d'échéance étant donné que la densité v est maximale. Dans la suite de l'algorithme, l'intervalle $I^\star$ est supprimé de l'axe du temps, ce qui revient à exclure cet intervalle lors du calcul de la densité maximale à l'itération suivante. Cet intervalle est aussi exclut lors de l'ordonnancement des tâches restantes. L'algorithme *YDS* prend fin lorsque toutes les tâches auraient été ordonnancées.

La question d'existence d'un algorithme polynomial pour le problème d'ordonnancement

avec un état de veille et vitesse variable a été posée par Irani et Pruhs dans [48] où les auteurs ont évoqué une difficulté particulière pour déterminer la complexité du problème baptisé *"Speed Scaling with Sleep State"*. Dans [49], Irani et al. ont initié l'étude de ce problème en présentant un algorithme polynomial 2-approché. Cet algorithme est basé sur la partition des tâches en deux sous ensembles : *les tâches rapides* et *les tâches lentes*. Une tâche J_i est dite *rapide* si sa vitesse critique définie par $w_i/(d_i - r_i)$ dépasse une vitesse bien définie v_{crit} et l'ensemble de ces tâches sont ordonnancées en utilisant l'algorithme *YDS*. Toutes les tâches restantes sont dites *lentes* et sont ordonnancées à la vitesse v_{crit}.

Récemment, et pendant la rédaction de ce travail, Albers et al. [4] ont réussi à prouver que le problème général est NP-difficile à l'aide d'une réduction à partir du problème de 2-partition. Ce résultat est valable même pour des instances ayant une structure d'arbre, *i.e.* pour 2 tâches j_i et j_j quelconques, on a soit $[r_i, d_i) \subseteq [r_j, d_j)$ ou $[r_j, d_j) \subseteq [r_i, d_i)$ ou $[r_i, d_i) \cup [r_j, d_j) = \emptyset$. Ils ont, par ailleurs, présenté un algorithme générique pouvant induire des rapports d'approximation intéressants dans le cas de fonctions de puissance convexes générales et aussi dans le cas de fonctions de puissance de la forme $P(v) = \beta v^{\alpha} + g$, où v est la vitesse d'exécution, et pour $\alpha > 1$, et β,$g > 0$.

2.4 S ’ -

Nous nous proposons dans cette partie de déduire quelques propriétés d'optimalité du problème décrit ci-dessus. Nous rappelons, tout d'abord, que si une tâche J_i est exécutée à une vitesse constante v alors w_i/v unités de temps seraient nécessaires afin d'achever la tâche J_i. L'énergie consommée dans ce cas est égale à $(v^{\alpha} + g)w_i/v$ et cette quantité d'énergie est minimisée lorsque la vitesse prend une valeur particulière $v^{\star} = (g/(\alpha - 1))^{1/\alpha}$ que nous appelons la vitesse *critique*. Cette vitesse est obtenue en annulant la dérivée de $(v^{\alpha} + g)w_i/v$ et ne dépend pas de la tâche en question puisqu'elle est indépendante de la quantité de travail w_i.
La densité d'un intervalle I étant définie par $\sum w_i/|I|$ sur toutes les tâches J_i avec $[r_i, d_i) \subseteq I$,

un intervalle est dit *dense* si sa densité est au moins égale à $v^\star$. Il est dit *épars* dans le cas contraire.

Dans la suite, nous supposons que toute instance du problème est agréable et que les tâches sont numérotées en respectant l'ordre total "$\preceq$" précédemment défini. Le lemme suivant montre l'existence d'un ordonnancement optimal avec quelques propriétés intéressantes permettant de déduire la structure de l'ordonnancement optimal.

Lemme 6 *[49] Soit une instance du problème de minimisation d'énergie avec un état de veille. Il existe un ordonnancement optimal* $(mode, vitesse, tâche)$ *vérifiant les propriétés suivantes.*

Job Span : *pour chaque instant t, si $tâche(t) = J_i \neq vide$ alors pour tout instant $u \in [r_i, d_i)$ avec $mode(u) = On$, on a $vitesse(u) \geqslant vitesse(t)$.*

Earliest Deadline First : *pour tout couple d'instants $t < u$ si $tâche(t), tâche(u) \neq vide$, alors $tâche(t) \preceq tâche(u)$.*

Intervalles Denses : *tout intervalle dense I est ordonnancé selon l'algorithme* YDS.

En particulier, la première propriété implique que chaque tâche est exécutée à une vitesse constante. Les deux autres propriétés impliquent que les intervalles divisent le problème en des sous problèmes indépendants, comme il sera décrit dans la suite.

Définition 3 *Une* sous-instance *de notre problème est spécifiée par une paire d'entier (i, j) avec $i \in \{1, \ldots, n\}$ et $j \in \{i-1, \ldots, n\}$. Pour plus de clarté, nous notons $d_0 = r_1 - L/g$ et $r_{n+1} = d_n + L/g$. Il s'agit, donc, d'un intervalle $I = [d_{i-1}, r_{j+1})$ et d'un ensemble de tâches J. Si $i = j+1$, alors $J = \emptyset$, sinon $J = \{J_i, \ldots, J_j\}$. Les intervalles constitués par les dates d'arrivée et les dates d'échéance de ces tâches sont restreints par intersection avec I.*

Notons que dans le cas où $d_{i-1} \geq r_{j+1}$ ou $d_{i-1} = d_i$ ou $r_j = r_{j+1}$, la sous-instance (i, j) n'est pas réalisable puisque l'intervalle de disponibilité d'au moins une des tâches $J_i, \ldots, J_j$ sera restreint à l'intervalle vide.

Par ailleurs, nous étendons la définition de la fonction de coût afin d'englober les sous-instances définies ci-dessus. L'ordonnancement d'une sous-instance (i, j), composée d'un ensemble de tâches J et d'un intervalle I, est définie par les fonctions vitesse : $I \to \mathbb{R}^+$, mode : $I \to \{On, Off\}$ et tâche : $I \to \{vide\} \cup J$. Ainsi, le coût d'exécution est donné par $c_{\text{vitesse}} = \int \text{vitesse}(t)^\alpha dt$ où l'integrale est calculée sur l'intervalle I. Pour le coût du mode du système, on note par $S := \{t \in I : \text{mode}(t) = On\}$ le support de l'ordonnancement, et k le nombre d'intervalles dans $I \setminus \cup S$. Dans ce cas, $c_{\text{mode}} := kL + g|\cup S|$ et si le système est en mode *On* immédiatement avant et après l'intervalle I, alors l'interprétation consiste à comptabiliser le coût de mise en marche L pour les intervalles *Off* sur les extrémités de I s'il en existe dans un ordonnancement optimal.

Nous fixons la valeur d_0 à une distance suffisante de r_1 telle que, sans perte de généralité, l'ordonnancement optimal de toute sous-instance de la forme $(1, k)$ débute par un intervalle *Off*. La propriété symétrique concernant la date r_{n+1} est aussi valable pour toute sous-instance de la forme (k, n). De ce fait, le coût de la sous-instance $(1, n)$ est cohérent avec la définition de la fonction du coût pour l'instance entière.
Notons, par ailleurs, que le coût d'une solution optimale d'une sous-instance de la forme $(i, i-1)$ est égal à $\min\{L, g(r_i - d_{i-1})\}$.

Considérons maintenant tous les intervalles denses et maximaux au sens de l'inclusion. Ces intervalles partitionnent l'axe du temps en une séquence d'intervalles alternant entre des intervalles denses et des intervalles épars.
Le lemme suivant est une conséquence directe de la définition des *sous-instances*. Nous insistons ici sur le fait que l'hypothèse des dates d'échéance agréables impliquent l'indépendance des sous-instances. Comme il a été précisé dans le lemme 6, il existe un ordonnancement optimal obéissant à la discipline *EDF* (earliest deadline first), ce qui veut dire que si une tâche J_j est ordonnancée, alors toutes les tâches $J_i \leq J_j$ ont déjà terminé leurs exécutions. Ainsi, l'hypothèse des instances

agréables est assez forte et donne une décomposition du problème qui permet de suivre une approche de programmation dynamique. Cependant, le problème ne devient pas trivial étant donné qu'il reste à décider des modes du système, *i.e.* quand est ce qu'on éteint la machine et quand est ce qu'on la réveille.

Lemme 7 *Les intervalles épars I sont associés à des paires* (i, j)*, telles que la partie d'un ordonnancement optimal pour l'instance originale qu'on restreint à l'intervalle I soit aussi un ordonnancement optimal pour la sous-instance* (i, j)*. De plus, aucune de ces sous-instances ne contient des intervalles denses.*

2.5 S P ´

Dans cette partie du chapitre, nous considérons un ordonnancement optimal pour une sous-instance arbitraire composée par un ensemble de tâches J et un intervalle I tel que tout sous intervalle de I soit épars. Tout au long de cette section, à chaque fois que nous faisons référence à des dates d'arrivée/d'échéance r_k, d_ℓ, ceux-ci sont restreintes à l'intervalle I.

Lemme 8 *Pour chaque instant* $t \in I$*, vitesse*$(t) \leqslant v^\star$*.*

Preuve Soit t un instant dans l'intervalle I qui maximise vitesse(t), et supposons, par contradiction, que vitesse$(t) > v^\star$.

Considérons un intervalle maximal au sens de l'inclusion $A \ni t$ pendant lequel la vitesse est constamment égale à vitesse(t). Soit $J_i, \ldots, J_k$, $J_i \preceq$ tâche$(t) \preceq J_k$, les tâches ordonnancées dans cet intervalle.

Si $A = [r_i, d_k)$, alors A est un intervalle dense et ceci contredit le Lemme 7. Ainsi, l'inclusion $A \subseteq [r_i, d_k)$ est stricte.

Supposons que $d_k > u$ avec $u = \max A$ (l'autre cas est symétrique). Par la propriété *Job Span* évoquée dans le Lemme 6, nous avons mode(u) = *Off*, et il existe un instant t' tel que la tâche J_k soit ordonnancée pendant $[t', u)$. Ainsi, pour une valeur $\delta \geqslant 1$ suffisamment petite, il est possible d'étirer l'exécution de la tâche J_k à l'intervalle $[t', u')$ pour $u' = t' + \delta(u - t')$ et diminuer sa vitesse

d'exécution jusqu'à vitesse$(t)/\delta$. Ceci donne une solution avec un coût total strictement inférieur, ce qui contredit l'optimalité de l'ordonnancement. □

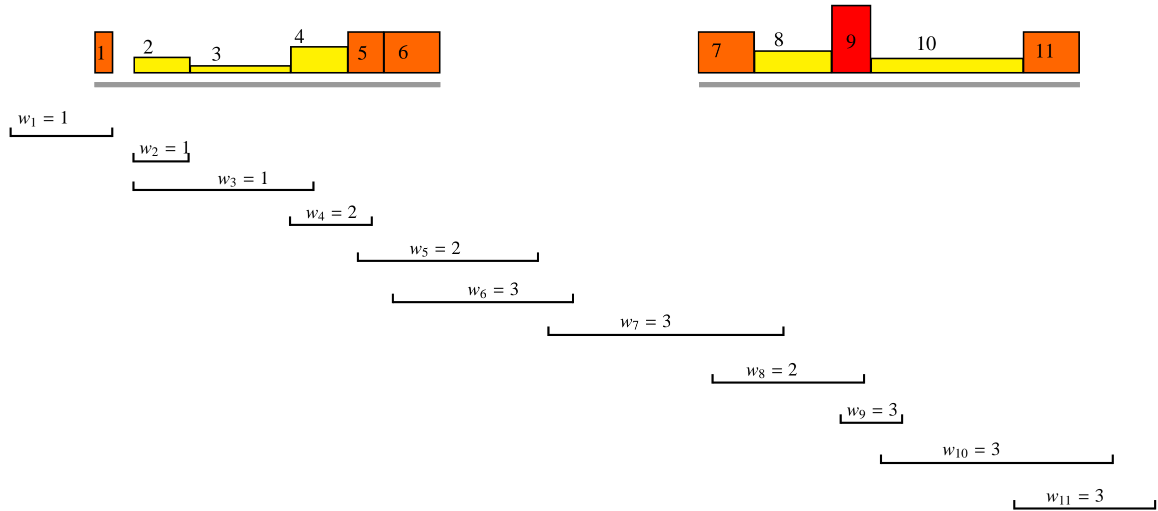
Le support de l'ordonnancement est composé de blocs séparés par des intervalles d'arrêt, *i.e.* système en état *Off*. Nous nous proposons, maintenant, de montrer que les bords de ces blocs ont une structure particulière (Figure 2.3).

Définition 4 *Un* suffixe *est une paire de tâches* (J_a, J_b) *telle que* $J_a \preceq J_b$ *et telle que toutes les tâches* $J_a, \dots, J_b$ *soient exécutées à la vitesse critique* $v^\star$ *entre les dates* r_a *et* u *avec* $u = r_a + (w_a + \dots + w_b)/v^\star$*, et mode*$(u) =$ *Off. La définition d'un* préfixe *est symétrique,* i.e. *il s'agit d'une paire de tâches* (J_a, J_b) *telle que* $J_a \preceq J_b$ *et où les tâches* $J_a, \dots, J_b$ *sont exécutées à la vitesse critique* $v^\star$ *entre les dates* s *et* d_b *avec* $s = d_b - (w_a + \dots + w_b)/v^\star$*, et le système passe de l'état Off à l'état On à l'instant* s *(mode*$(s - \delta) =$ *Off pour une valeur* $\delta > 0$ *suffisament petite).*

Lemme 9 *Soit* $[t, u)$ *un intervalle d'arrêt maximal au sens de l'inclusion dans* I*, c'est à dire mode*$(t') =$ *Off pour tout* $t' \in [t, u)$*. Si* t *ne représente pas la borne inférieure de l'intervalle* I*, alors il existe un suffixe* (a, b) *se terminant à l'instant* $t = r_a + (w_a + \dots + w_b)/v^\star$*. Si* u *ne représente pas la borne supérieure de l'intervalle* I*, alors il existe un préfixe* $(b + 1, c)$ *qui débute à l'instant* $u = d_c - (w_{b+1} + \dots + w_c)/v^\star$*. De plus, si ces deux conditions sont vérifiées,* i.e. $\inf I < t < u < \sup I$*, alors, sans perte de généralité, on a* $r_{b+1} > t$*.*

Preuve Supposons, par contradiction, qu'il existe un intervalle $[t_0, t)$ où le système est en état de marche et tel qu'une tâche $J_b =$ tâche(t_0) soit ordonnancée à la vitesse vitesse$(t_0) < v^\star$. Pour une valeur suffisamment petite $\varepsilon > 0$, soit $t' := t_0 + (t - t_0)/(1 + \varepsilon)$. Considérons, maintenant, un nouvel ordonnancement où l'intervalle $[t_0, t)$ est compressé à l'intervalle $[t_0, t')$ et où la vitesse de la machine est augmentée par un facteur de $1 + \varepsilon$. Dans ce cas l'intervalle d'arrêt $[t, u)$ sera étendu à $[t', u)$ et il est clair que le coût total induit par ce nouvel ordonnancement est strictement inférieur, ce qui contredit l'optimalité de la solution initiale.

Ceci montre que si t n'est pas la date de début de l'intervalle I, alors il existe une tâche J_b s'exécutant juste avant l'instant t, c'est-à-dire dans l'intervalle $[t_0, t)$, à la vitesse critique $v^\star$. Nous allons,



F . 2.3 – Structure d'un ordonnancement optimal pour une instance composée de 11 tâches (les segments d'intervalles représentent les disponibilités des tâches et indiquent leurs dates d'arrivée et d'échéance ainsi que leurs quantités de travail). L'ordonnancement optimal résultant est composé de 2 blocs séparés par un intervalle d'arrêt où le système est en mode *Off* (limité par la fin de la tâche 6 et le début de la tâche 7). Les tâches 5 et 6 forment le suffixe du premier bloc. Les tâches 1,5,6,7 et 11 sont exécutées à la vitesse critique $v^\star$, alors que la tâche 9 est ordonnancée à une vitesse supérieure, puisque $[r_9, d_9)$ est un intervalle dense.

à présent, montrer qu'il existe une tâche J_a telle que toutes les tâches $J_a \preceq \ldots \preceq J_b$ soient ordonnancées à la vitesse critique entre les dates r_a et t. Dans le cas où $t_0 = r_b$, nous avons $a = b$. Sinon, admettons que $r_b < t_0$. Dans l'ordonnancement considéré, si le système, juste avant l'instant t_0, est en mode *Off*, alors il sera possible de décaler légèrement l'exécution de la tâche J_b dans l'intervalle $[t_0 - \varepsilon, t - \varepsilon)$ afin d'obtenir un ordonnancement de même coût, mais en exécutant la tâche au plutôt possible. Sans perte de généralité, nous pouvons supposer que, juste avant t_0, une tâche J_{b-1} est ordonnancée dans un intervalle $[t_1, t_0)$. Par la propriété du *Job Span* dans le lemme 6 et par le lemme 8, la tâche J_{b-1} est exécutée à la vitesse critique $v^\star$. Il suffit, donc, d'itérer ces mêmes arguments pour t_1 et J_{b-1} jusqu'à obtenir une tâche J_a avec les propriétés requises.

Le même argument est appliqué d'une façon symétrique pour montrer l'existence d'un préfixe $(b + 1, c)$ si l'instant u n'est pas la borne supérieure de l'intervalle I. Maintenant, si le suffixe et

le préfixe existent tous les deux et $r_{b+1} \leqslant t$, alors il est possible de décaler l'exécution de la tâche J_{b+1} de l'intervalle $[u, w_{b+1}/v^\star)$ à l'intervalle $[t, w_{b+1}/v^\star)$ donnant lieu à un ordonnancement avec un coût qui peut être soit le même, soit réduit de L, si J_{b+1} était la seule tâche dans le bloc correspondant. Nous pouvons, ainsi, supposer que $t < r_{b+1}$ sans perte de généralité. □

Avant de passer à la description de notre algorithme basé sur la programmation dynamique, nous avons besoin d'une dernière propriété sur les suffixes et les préfixes et qui peut être déduite de la définition suivante.

Définition 5 *Soit une sous-instance (i, j). Nous définissons deux fonctions $f, h : \{i, \ldots, j\} \to \{i, \ldots, j\}$ comme suit : $f(a)$ est le plus grand indice de tâche $a < b \leqslant j$ tel que pour tout $k \in \{a, \ldots, b-1\}$, $r_a + (w_a + \ldots + w_k)/v^\star \geqslant r_{k+1}$, tandis que $h(k)$ est le plus grand indice de tâche $k < c \leqslant j$ tel que pour tout $\ell \in \{k+1, \ldots, c\}$, $d_c - (w_\ell + \ldots + w_c)/v^\star \leqslant d_{\ell-1}$. Dans le cas où il n'existe pas un indice b (*resp. *c) vérifiant les conditions décrites ci-dessus, alors $f(a) = a$ (*resp. *$h(k) = k$).*

Lemme 10 *Chaque suffixe (a, b) satisfait $b = f(a)$ et chaque préfixe (k, c) satisfait $c = h(k)$.*

La fonction f requiert un peu plus d'attention afin de pouvoir décrire correctement notre algorithme de programmation dynamique. En effet, fixons une tâche J_b appartenant à la sous-instance (i, j) et supposons qu'il existe plusieurs tâches $J_a \preceq \ldots \preceq J_k$ telles que $f(\ell) = b$ pour tout $\ell \in \{a, \ldots, k\}$. Dans ce cas, étant donné que, par le lemme 8, aucune de ces tâches ne peut dépasser la vitesse critique, il est possible de supposer qu'un suffixe (a, b) est tel que a soit le plus petit indice de tâche vérifiant $f(a) = b$. Ainsi, nous limitons, pour la suite, le domaine de définition de la fonction f à ces tâches. Ceci permet à f d'être inversible, *i.e.* $a = f^{-1}(b)$. Notons que par définition de f, la tâhe J_j est dans le codomaine de f, ce qui implique que $f^{-1}(j)$ est bien définie.

2.6 L

Pour chaque sous-instance (i, j), nous désignons par $Y_{i,j}$ le coût d'exécution c_{vitesse} minimum plus $g(r_{j+1} - d_{i-1})$, et par $O_{i,j}$ le coût d'une solution optimale donnée par le minimum de $c_{\text{vitesse}} + c_{\text{mode}}$. Dans le cas où la sous-instance (i, j) n'est pas réalisable, *i.e.* si $d_{i-1} \geq r_{j+1}$ ou $d_{i-1} = d_i$ ou $r_j = r_{j+1}$, alors nous fixons les valeurs de $Y_{i,j}$ et $O_{i,j}$ à $+\infty$. Pour plus de simplicité, nous définissons $g^\star := (g + (v^\star)^\alpha)/v^\star$.

Théorème 3 *La valeur donnée par $O_{i,j}$ satisfait la récursion suivante :*
Si $j = i - 1$, alors $O_{i,j} = min\{L, g(r_{j+1} - d_{i-1})\}$, sinon, soit $k = f^{-1}(j)$,

$$O_{i,j} = \min \begin{cases} Y_{i,j} \\ L + g^\star(w_i + \ldots + w_{h(i)}) + O_{h(i)+1,j} \\ Y_{i,k-1} + g^\star(w_k + \ldots + w_j) + L \\ \min Y_{i,a-1} + g^\star(w_a + \ldots + w_b) + L + g^\star(w_{b+1} + \ldots + w_c) + O_{c+1,j}, \end{cases} \tag{2.1}$$

où la minimisation dans le dernier cas porte sur toutes les tâches $a \in \{i + 1, \ldots, j\}$ et $\{b, c\}$ avec $b = f(a), b < j$ et $c = h(b + 1)$. S'il n'existe pas de telles tâches, alors la valeur de cette minimisation sera égale à $+\infty$.

Preuve Le cas où $j = i - 1$ est simple, puisque si l'ordonnancement est vide alors dans la solution optimale le système sera soit en mode *On* mais n'exécutant aucune tâche, ou tout simplement éteint, *i.e.* en mode *Off*. Il est clair que le choix du mode dépend uniquement de la longueur de l'intervalle $[r_{j+1}, d_{i-1})$.

Considérons, maintenant, une sous-instance (i, j) telle que $i \leqslant j$. Il est possible de montrer, par induction sur $j - i$, que pour chacun des quatre cas décrits dans (2.1) il existe un ordonnancement réalisable ayant le coût correspondant. Dans la suite de cette preuve, nous considérons un ordonnancement S qui minimise $c_{\text{vitesse}} + c_{\text{mode}}$ pour cette sous-instance, et nous nous proposons

que ce coût est donné par un parmi les quatre cas dans (2.1).

Si l'ordonnancement S ne met jamais le système en mode *Off*, alors la contribution de c_{mode} dans le coût total est equivalente à $g(r_{j+1} - d_{i-1})$. Par ailleurs, la contribution de c_{vitesse} correspond à la valeur minimale donnée par $Y_{i,j}$. Ainsi, le premier cas s'applique.

Supposons maintenant qu'il existe un intervalle $[t, u)$ pendant lequel la machine est en mode *Off* dans l'ordonnancement S, et considérons que $[t, u)$ est le premier intervalle maximal par rapport à l'inclusion. Il existe dans ce cas quelques cas à considérer en fonction des conditions $t = \min I$, $u = \max I$, où I est l'intervalle associé à la sous-instance (i, j).

Tout d'abord, il n'est pas possible que les deux conditions soient vraies car ceci implique que l'ordonnancement est vide ce qui contredit l'hypothèse $i \leq j$.
Maintenant, si $t = \min I$ et $u < \max I$, alors le Lemme 9 implique l'existance d'un préfixe (i, c) tel que $c = h(i)$. La portion de l'ordonnancement S qui s'étale de l'instant t jusqu'à l'instant d_c contribue au coût total de la solution par $L + g^\star(w_i + \ldots + w_{h(i)})$, et par composition des ordonnancements, la suite de S, *i.e.* la portion sur l'intervalle $[d_c, \max I)$ contribue au coût par $O_{h(i)+1,j}$. Ainsi, le second cas de (2.1) s'applique.

Si $t > \min I$ et $u = \max I$, d'une façon similaire il existe un suffixe (k, j) et le coût de l'ordonnancement du début de I jusqu'à r_k est donné par $Y_{i,k-1}$, puisque la machine n'est jamais éteinte durant cet intervalle, *i.e.* $[\min I, r_k)$. Le reste de l'ordonnancement s'étalant de r_k jusqu'à $u = \max I$ contribue au coût par $g^\star(w_k + \ldots + w_j) + L$. Dans ce cas la troisième alternative de (2.1) est appliquée.

Finalement, si $t > \min I$ et $u < \max I$, de nouveau à cause du Lemme 9, il existe un suffixe (a, b) et un préfixe $(b + 1, c)$ entourant l'intervalle $[t, u)$ où la machine est éteinte. Par le Lemme

10 nous avons $b = f(a), c = h(b+1)$. Ainsi, le coût de l'ordonnancement est décomposé en $Y_{i,a-1}$ pour la portion qui précède r_a et qui ne contient pas d'intervalle où le mode est *Off*, le coût $g^\star(w_a + \ldots + w_c) + L$ pour la portion représentée par l'intervalle $[r_a, d_c)$, et, enfin, $O_{c+1,j}$ qui correspond au coût du reste de l'ordonnancement par composition. Il s'agit, donc, d'appliquer le dernier cas de (2.1). □

2.7 A ´

Le programme dynamique décrit dans la section précédente utilise $O(n^2)$ variables $O_{i,j}$, et pour chacune de ces variables une minimisation sur un ensemble de $O(n)$ valeurs est requise. Ainsi, ce programme peut être exécuté en $O(n^3)$.

Pour une sous-instance fixée (i, j), il est possible de calculer les fonctions f et h indépendamment du programme dynamique à l'aide de simples procédures en temps linéaire comme suit :

- Initialement, $\ell := i$ et $t := r_i$. Pour chaque $k = i, i+1, \ldots, j$: si $t < r_k$ alors $\ell := k$, $t := r_k$. Sinon, $f(\ell) := k$ et $t := t + \frac{w_k}{v^\star}$.

- Initialement, $\ell := j$ et $t := d_j$. Pour chaque $k = j, j-1, \ldots, i$: si $t > d_k$ alors $\ell := k$, $t := d_k$. Sinon, $h(k) := \ell$ et $t := t - \frac{w_k}{v^\star}$.

La détermination des valeurs $Y_{i,j}$ est, cependant, cruciale puisque il en existe $O(n^2)$ et le meilleur algorithme d'ordonnancement pour résoudre ces problèmes retourne la solution optimale en $O(n^2)$ [81]. Ceci mène à un temps total en $O(n^4)$ pour calculer ces valeurs. Nous décrivons, dans la suite, une procédure permettant de calculer $Y_{i,j}$ d'une façon itérative à partir de $Y_{i-1,j}$ en $O(n)$, ce qui permettra de calculer tous les sous-ordonnancement c_{vitesse}-optimaux en un temps total en $O(n^3)$.

Calcul des $Y_{i,j}$

Nous présentons le schéma général pour le calcul des valeurs $Y_{i,j}$. Tout d'abord, nous déterminons $Y_{1,n}$ en $O(n^2)$ en utilisant l'algorithme décrit dans [81]. Ensuite, nous calculons toutes les valeurs $Y_{1,j}$ pour $j = n-1, \ldots, 1$ en un premier passage de droite à gauche. Finalement, pour chaque $j = 2, \ldots, n$ fixé, nous effectuons un passage de gauche à droite pour calculer toutes les valeurs $Y_{i,j}$ pour $i = 2, \ldots, j$. Ce dernier passage de gauche à droite est calculé de la façon suivante.

Soit en entrée l'ordonnancement c_{vitesse}-optimal S pour la sous-instance $(1, j)$. L'ordonnancement S consiste en une séquence de blocs telle que chaque bloc s'étale sur un certain intervalle $[t, u)$ et contienne une séquence de tâches qui s'exécutent à une vitesse constante dépendante du bloc en question. Nous appliquons à S la procédure de compression que nous décrivons maintenant.

Durant cette procédure, nous gardons la trace du premier bloc et nous supposons que ce bloc s'étale sur un intervalle $[t, u)$ durant lequel les tâches $J_i, \ldots, J_b$ sont exécutées à une vitesse v.

• Initialement, $i = 2$, et nous considérons l'opération qui consiste à accroître la vitesse v dans le but de compresser le bloc de tâches à l'intervalle $[u - \ell, u)$, avec $\ell := u - (w_i + \ldots + w_b)/v$.

• Tant que $i \leqslant j$ et au fur et à mesure que l'opération de compression est appliquée, nous décidons du premier événement qui survient parmi les événements suivants (figure 2.4), et appliqons les actions correspondantes :

Événement de Non Faisabilité : Cet événement a lieu si $d_{i-1} = d_i$. Étant donné que dans la sous-instance (i, j) toutes les tâches sont restreintes à l'intervalle $[d_{i-1}, r_{j+1}]$, il s'en suit que la tâche J_i serait restreinte à un intervalle vide et de ce fait, ne peut être ordonnancée à une

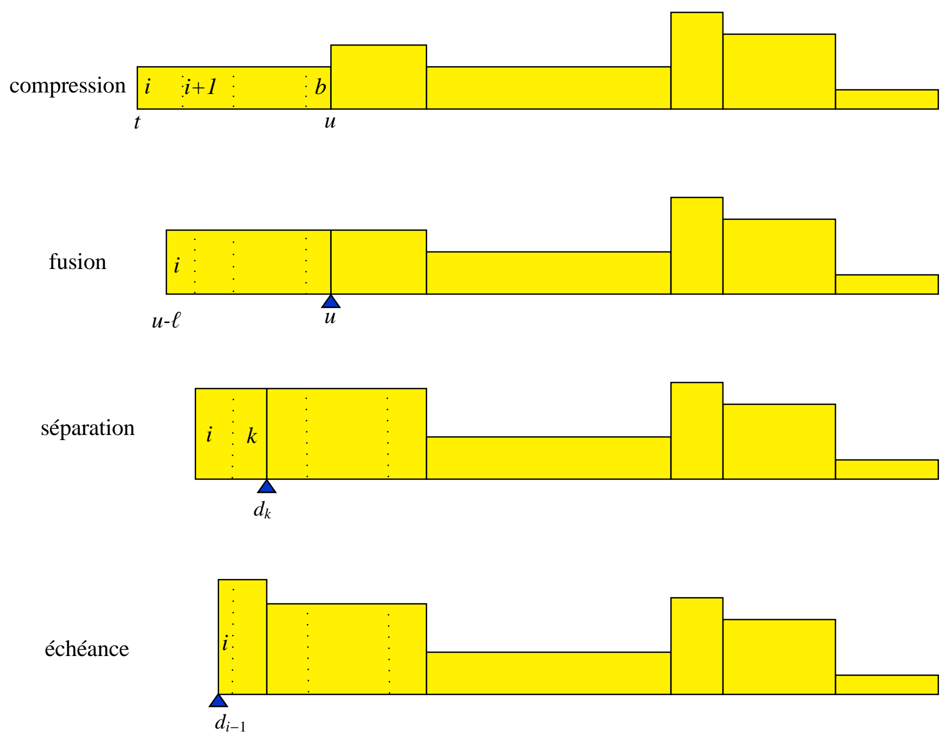
vitesse finie. Dans ce cas, l'action consiste à annoncer que la sous-instance (i, j) n'est pas réalisable, fixer $Y_{i,j} = +\infty$, et incrémenter i. Notons que si cet événement se réalise, alors ça sera le premier événement puisqu'il ne dépend pas du changement de la vitesse mais uniquement de la structure de l'instance.

Événement de Fusion : Cet événement a lieu quand la vitesse v devient égale à vitesse(u). Dans ce cas, l'action consiste à fusionner les deux premiers blocs. Notons que si $u = d_b$, alors cet événement sera immédiatement suivi par un événement de séparation appliqué au nouveau bloc résultant de la fusion.

Événement de Séparation : À un moment donné, la fin d'exécution d'une tâche $J_i \preceq J_k \prec J_b$ appartenant au premier bloc pourrait coincider avec sa date d'échéance. Ceci peut arriver lorsque la vitesse v atteint une vitesse définie par $\hat{s}(k, b, u) = (w_{k+1} + \ldots + w_b)/(u - d_k)$. Dans ce cas, le bloc sera scindé en deux nouveaux blocs avec le premier bloc restreint à l'intervalle $[t, d_k)$ et les tâches $J_i, \ldots, J_k$. On fixe $b := k$, $u := d_k$ et on continue à compresser le premier bloc obtenu.

Événement d'Échéance : Quand $v = (w_i + \ldots + w_b)/(u - d_{i-1})$, l'ordonnancement S obtenu sera l'ordonnancement c_{vitesse}-optimal pour la sous-instance (i, j). Dans ce cas la procédure retourne S comme étant l'ordonnancement qui correspond à $Y_{i,j}$, élimine la tâche J_i de S, et incrémente la valeur de i.

Il faut préciser qu'à chaque instant, l'algorithme maintient un ordonnancement S pour la sous-instance composée de toutes les tâches $J_i, \ldots, J_j$ dont les dates d'arrivée et les dates d'échéance sont restreintes à l'intervalle $[u - \ell, r_{j+1}]$. Comme il a été mentionné, précédemment, le calcul des valeurs $Y_{1,j}$ pour $j = n - 1, \ldots, 1$ se fait par un premier passage de droite à gauche. En effet, lors de ce passage, il s'agit d'appliquer la même procédure de compression mais d'une



F . 2.4 – Les différents événements durant la procédure de compression

façon totalement symétrique. Ainsi, l'événement de *Non Faisabilité* par exemple aura lieu si $r_j = r_{j+1}$ et dans ce cas on fixe $Y_{1,j} = +\infty$ et on décrémente j.

Il reste à spécifier la façon avec laquelle les différents événements sont détérminés en un temps constant. Les événements de *Fusion* et d'*Échéance* sont détérminés chacun par une expression unique et simple donnant la valeur ℓ à laquelle ces événements ont lieu. Cependant, en ce qui concerne l'événement de *Séparation*, la situation est plus subtile.

En effet, il existe $b - i$ candidats $\hat{s}(k, b, u)$, *i.e.* un pour chaque tâche $J_i \preceq J_k \prec J_b$. Ainsi, un pré-calcul de ces valeurs est nécessaire. Notons que pour une tâche donnée J_b, il existe $O(n)$ instants différents u à considérer, et ils sont tous de la forme d_b, r_{b+1} et r_{j+1} pour tout $1 \leqslant j \leqslant n$. Ceci est dû au fait que chaque bloc dans un ordonnancement optimal se termine soit à la fin de l'intervalle I s'il s'agit du dernier bloc, soit à l'une des valeurs d_b, r_{b+1}, selon que le bloc suivant possède une plus grande ou plus petite vitesse.

Ainsi, il existe $O(n^3)$ valeurs de la forme $\hat{s}(k, b, u)$ à calculer, et pour chaque paire (b, u), ceci peut être réalisé en un temps linéaire en faisant varier k de $b - 1$ à 1. Dans la procédure de compression, nous avons besoin de déterminer la tâche k, $i \leqslant k \leqslant b$, qui minimise $\hat{s}(k, b, u)$. Il est clair que cette tâche k peut être calculée en un temps constant pour chaque triplet (i, b, u) en faisant itérer i de $b - 1$ à 1 pour chaque pair (b, u).

Finalement, dans la procédure de compression décrite précédemment, chaque tâche peut engendrer au plus 3 événements. Ainsi, pour une tâche J_j fixée, la complexité est de $O(n)$. Ceci induit un temps d'exécution total en $O(n^3)$.

2.8 C

Nous avons considéré dans ce chapitre le problème qui consiste à ordonnancer sur une seule machine un ensemble de tâches caractérisées par des dates d'arrivée, des dates d'échéance et des quantités de travail. L'objectif est de minimiser l'énergie consommée en adaptant non seulement la vitesse mais aussi l'état de la machine. Il est ainsi question d'éteindre la machine aux moments opportuns et de choisir des vitesses d'exécution appropriées permettant de minimiser le nombre d'alternance entre les périodes de veille et d'activité et de favoriser des périodes de veille longues.

Nous avons donc proposé un algorithme polynomial lorsque les dates d'échéance des tâches sont agréables. Cette hypothèse nous a aidé à déduire des propriétés structurelles des solutions optimales faisant apparaître une décomposition possible du problème et permettant ainsi de construire un algorithme de programmation dynamique qui retourne des ordonnancements sans préemption des tâches. Nous concluons donc qu'il n'est pas possible de généraliser cet algorithme pour traiter des instances avec des dates d'échéance arbitraires. Cependant, nous pensons que la procédure de compression qui permet de calculer les ordonnancements c_{vitesse}-optimaux peut être améliorée d'avantage afin d'aboutir à une complexité en temps plus intéressante.

Finalement, une perspective intéressante de ce travail consiste à étendre ce modèle dans le cas de plusieurs machines avec ou sans migration et/ou préemption des tâches.

C 3

O

3.1 I

Dans la littérature, la pluspart des travaux sur les problèmes d'ordonnancement en rapport avec la minimisation d'énergie ont considéré des environnements avec une machine unique [72, 6], ou des environnements avec des machines homogènes parallèles [28, 61, 73, 7]. Cependant, le recours aux systèmes composés de processeurs hétérogènes semble être une tendance dominante dans les architectures futures [23, 58, 65, 66].

Dans ce contexte, l'utilisation de dispositifs de calcul hétérogènes à la place d'un unique système composé de processeurs homogènes parallèles rend nécessaire l'introduction de modèles qui tiennent compte de l'hétérogénéité des processeurs.

Nous présentons dans ce chapitre un modèle basé sur une extension naturelle du modèle d'ordonnancement classique sur des machines non uniformes (*hétérogènes*) [15]. Nous considérons, ainsi, un ensemble $\mathcal{J}$ de n tâches $\{J_1, \ldots, J_n\}$ et un ensemble de m machines non uniformes où

chaque machine peut changer sa vitesse d'exécution en la choisissant parmi un ensemble fini de vitesses $\mathcal{V}$. Chaque tâche J_j est caractérisée par une pondération w_j, une date d'arrivée qui dépend de la machine, un temps d'exécution et une énergie consommée fonctions de la machine et de la vitesse. Plus précisément, r_{ij} est la date d'arrivée de la tâche J_j si elle est exécutée sur la machine i, tandis que p_{ijv} et E_{ijv} représentent, respectivement, le temps d'exécution et l'énergie consommée par la tâche J_j si elle est exécutée sur la machine i à la vitesse $v \in \mathcal{V}$. Nous supposons que toutes ces valeurs sont entières et positives. Ce modèle est bien adapté à des architectures composées de machines spécialisées qui exécutent efficacement des tâches de types particuliers. La dépendance des dates d'arrivée aux machines est une hypothèse pertinente dans le cas où les machines sont connectées par un réseau de communication. Dans ce cas, nous supposons que toutes les tâches sont disponibles à l'instant 0 sur une machine particulière, et qu'une durée r_{ij} doit s'écouler pour qu'une tâche quelconque J_j migre sur une machine i. Ce modèle d'ordonnancement dans les réseaux a été introduit dans [14, 38]. Par ailleurs, les temps d'exécution ainsi que l'énergie consommée sont pré-calculés pour tout triplet de machine, tâche et vitesse. Il est clair que le choix des vitesses des machines dépend des tâches en exécution et la puissance de consommation d'énergie n'obeit pas à une loi particulière comme il était le cas dans les chapitres précédents. De plus, la préemption et la migration des tâches ne sont pas autorisées. Le but est de trouver un ordonnancement réalisable minimisant la somme des temps de complétude pondérés plus l'énergie consommée, *i.e.* $\sum_{j=1}^{n} w_j C_j + E$, où C_j représente le temps de complétude (*ou temps de fin d'exécution*) de la tâche J_j et E l'énergie totale consommée par l'ordonnancement. Si on étend la notation classique en trois champs de Graham et al. [42], ce problème peut être noté par $RS|\, r_{ij}|E + \sum w_j C_j$.

3.2 T

Généralement, la minimisation de la consommation d'énergie dans les systèmes informatiques est en opposition avec les critères d'optimisation classiques qui traduisent la qualité

d'un ordonnancement donné. Dans ce contexte, une multitude de problèmes ont été étudiés. Dans [72], Pruhs et al. ont initié l'étude des problèmes où un budget énergétique est donné, le but étant d'optimiser une fonction objectif classique à savoir le temps de réponse total[1] (*total flow time*) sans dépasser ce budget. Dans le cas où les quantités de travail des tâches sont unitaires, les auteurs ont présenté un algorithme polynomial lorsque les tâches sont à ordonnancer sur une seule machine. L'optimalité de cet algorithme a été prouvé en modélisant le problème sous la forme d'un programme convexe et en appliquant les conditions de Karush-Kuhn-Tucker (*conditions KKT*) [54, 57] pour déduire les conditions d'optimalité. Par aileurs, Albers et Fujiwara se sont intéressés dans [6] au problème de minimisation du temps de réponse total plus l'énergie consommée. En utilisant une approche de programmation dynamique, les auteurs ont décrit un algorithme optimal dans le cas off-line. Dans le cas on-line ils ont présenté une solution compétitive si les quantités de travail sont unitaires. Ils ont également montré qu'aucun algorithme on-line ne peut réaliser un rapport de compétitivité constant si les quantités de travail sont arbitraires. Dans [30], Chan et al. considèrent le modèle d'adaptation de la vitesse (*speed scaling*) et imposent une borne maximale sur la vitesse des machines. Étant données des tâches avec des dates d'échéance, l'objectif est de maximiser le débit, *i.e.* le nombre de tâches exécutées, tout en minimisant l'énergie consommée lorsque la fonction de puissance est cubique. Il s'agit en d'autres termes de choisir parmi tous les ordonnancements qui maximisent le débit, ceux qui dépensent le moins d'énergie. Dans ce contexte, Chan et al. présentent un algorithme on-line dont le rapport de compétitivité est constant.

Dans le modèle classique sur plusieurs machines et sans introduction de l'énergie, il existe un nombre important de travaux sur les problèmes d'ordonnancement visant à minimiser la somme des temps de complétude pondérés [33]. Dans le cas où les machines sont homogènes (*related*) et en présence de dates d'arrivée, Chekuri et Khanna [32] ont présenté un shéma d'approximation polynomial (*PTAS*) pour le problème $Q| r_j | \sum w_j C_j$. Dans le cas des machines hétérogènes (*unrelated*), la situation est la suivante : si le nombre de machines est une constante

[1] Étant donné un ordonnancement quelconque, le temps de réponse d'une tâche J_i est le temps qui s'écoule entre sa date d'arrivée r_i et sa date de complétude C_i.

fixée, $Rm||\sum w_j C_j$, un shéma d'approximation polynomial a été présenté par Afrati et al. dans [1]. Dans le cas général où le nombre de machines fait partie de l'entrée du problème, il existe une $(3/2+\epsilon)$-approximation pour le problème $R||\sum w_j C_j$ et une $(2+\epsilon)$-approximation pour le problème $R|\ r_j\ |\sum w_j C_j$ proposées par Shulz et Skutella [75]. Ces derniers résultats sont basés sur l'approche de l'*Arrondi Aléatoire* [74].

L'approche de l'arrondi aléatoire présentée par Shulz et al. [75] consiste à proposer une relaxation du problème en utilisant un programme linéaire dont les variables sont indexées sur le temps. Ensuite, en se basant sur la solution fractionnaire du PL, on construit une solution entière à l'aide de l'arrondi aléatoire qui mène à une 2-approximation. Dans le cas où toutes les tâches sont relachées simultanément, cette technique donne une $3/2$-approximation. Étant donné que le nombre de variables indexées sur le temps est exponentiel dans la relaxation linéaire du problème, les auteurs ont proposé un autre PL de taille polynomial dont les variables sont indexées par des intervalles de tailles géomériques.

En utilisant la même technique de l'*Arrondi Aléatoire*, nous sommes capables de proposer un algorithme randomisé donnant une $2(1+\epsilon)$-approximation pour le problème qui tient compte de l'énergie consommée. Avec cette même technique, nous trouvons le même rapport d'approximation pour le problème d'ordonnancement avec minimisation de la somme des temps de complétude pondérés avec un budget d'énergie fixe à ne pas dépasser. À l'aide de la notation de Graham, ce dernier problème peut être noté par $RS|r_{ij}, E|\sum w_j C_j$.
Indépendamment, Carrasco et al. [29] ont considéré un modèle proche au notre mais dans le cas d'une machine unique. Plus précisément, ils ont décrit des algorithmes d'approximation avec des facteurs constants pour le problème d'ordonnancement sur une machine unique et visant à minimiser la somme des temps de complétude pondérés plus l'énergie consommée. Dans ce problème, la préemption n'est pas autorisée et les fonctions de consommation d'énergie sont dépendantes des tâches qui possèdent des dates d'arrivée et d'échéance. Ils ont aussi étendu ces algorithmes pour le

problème de minimisation de la somme des retards pondérés plus l'énergie (le retard d'une tâche est nul si sa date de complétude ne dépasse pas sa date d'échéance, et il est égal à la durée de dépassement sinon). Pour établir ces résultats, les auteurs ont eu recours à la technique des *α-points*, basée sur la programmation linéaire entière.

3.3 R ´

Dans cette partie du chapitre, nous présentons une relaxation linéaire de taille polynomiale du problème $RS|r_{ij}|E + \sum w_j C_j$. Soit $T = \max_{ij} r_{ij} + \sum_{J_j \in \mathcal{J}} \max_{iv} p_{ijv}$ l'horizon du temps. On discrétise l'intervalle $[0, T]$ en des intervalles de tailles géométriques comme suit : soit un réel $\epsilon > 0$ et soit L le plus petit entier tel que $(1+\epsilon)^L \geqslant T$. On définit, ainsi, l'ensemble des intervalles I_ℓ, $0 \leqslant \ell \leqslant L$, par :

$$I_\ell = \begin{cases} [0, 1] & \text{si } \ell = 0, \\ ((1+\epsilon)^{\ell-1}, (1+\epsilon)^{\ell}] & \text{si } 1 \leq \ell \leq L. \end{cases}$$

Pour tout $\ell \in \{0, \ldots, L\}$, on note par $|I_\ell|$ la taille de l'intervalle I_ℓ, et on a $I_0 = 1$ et $I_\ell = \epsilon(1+\epsilon)^{\ell-1}$ pour $1 \leqslant \ell \leqslant L$. Par ailleurs, notons que puisque la taille de ces intervalles augmente d'une façon géométrique, leur nombre est polynomial en la taille de l'instance du problème.

Soit $y_{ij\ell v}$ pour tout $i = 1, \ldots, m$, $J_j \in \mathcal{J}$, $\ell = 0, \ldots, L$ et $v \in \mathcal{V}$ l'ensemble des variables tel que $y_{ij\ell s}.|I_\ell|$ soit la durée de temps que prend l'exécution de la tâche J_j pendant l'intervalle $|I_\ell|$ sur la machine i si celle-ci tourne à la vitesse v. Ainsi, la variable $y_{ij\ell v}$ peut être considérée comme étant la fraction de l'intervalle $|I_\ell|$ occupée par la tâche J_j sur la machine i lorsqu'elle tourne à la vitesse v. D'une façon équivalente, $\frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}$ est la fraction de la tâche J_j exécutée pendant I_ℓ sur la machine i à la vitesse v. Nous considérons, également, un second ensemble de variables : C_j^{LP} pour tout $j = 1, \ldots, n$ où chaque C_j^{LP} correspond à la date de complétude de la tâche J_j dans la relaxation. Finalement, pour $\epsilon > 0$, le programme linéaire que nous notons par LP^ϵ est décrit comme suit :

$$\min \sum_{i=1}^{m} \sum_{J_j \in \mathcal{J}} \sum_{\ell=0}^{L} \sum_{v \in \mathcal{V}} E_{ijv} \frac{y_{ij\ell v}|I_\ell|}{p_{ijv}} + \sum_{J_j \in \mathcal{J}} w_j C_j^{LP}$$

$$\text{s.c.} \quad \sum_{i=1}^{m} \sum_{\ell=0}^{L} \sum_{v \in \mathcal{V}} \frac{y_{ij\ell v}|I_\ell|}{p_{ijv}} = 1, \qquad \text{pour tout } j, \tag{3.1}$$

$$\sum_{J_j \in \mathcal{J}} \sum_{v \in \mathcal{V}} y_{ij\ell v} \leq 1, \qquad \text{pour tout } i \text{ et } \ell, \tag{3.2}$$

$$C_j^{LP} \geq \sum_{i=1}^{m} \sum_{v \in \mathcal{V}} \frac{1}{2} y_{ij0v}|I_0| \left(1 + \frac{1}{p_{ijv}}\right) + \sum_{i=1}^{m} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} \left(\frac{y_{ij\ell v}\,|I_\ell|}{p_{ijv}} (1+\epsilon)^{\ell-1} + \frac{1}{2} y_{ij\ell v}|I_\ell| \right), \qquad \text{pour tout } j, \tag{3.3}$$

$$C_j^{LP} \geq \sum_{i=1}^{m} \sum_{\ell=0}^{L} \sum_{v \in \mathcal{V}} y_{ij\ell v}|I_\ell|, \qquad \text{pour tout } j, \tag{3.4}$$

$$y_{ij\ell v} = 0, \quad \text{pour tout } i, j \ , \ (1+\epsilon)^\ell \leq r_{ij}, \ v, \tag{3.5}$$

$$y_{ij\ell v} \geq 0, \qquad \text{pour tout } i, j \ , \ell, \text{et } v, \tag{3.6}$$

$$C_j^{LP} \geq 0, \qquad \text{pour tout } j. \tag{3.7}$$

Proposition 1 *Le programme linéaire LP^ϵ est une relaxation du problème $RS \mid r_{ij}|E + \sum w_j C_j$.*

Preuve L'ensemble des égalités 3.1 garantissent que chaque tâche sera exécutée en sa totalité. Les inégalités 3.2 expriment le fait que la durée d'exécution totale sur une machine i et pendant un intervalle I_ℓ ne dépasse jamais sa durée, *i.e.* $|I_\ell|$. En d'autres termes, chaque machine peut exécuter au plus une seule tâche à chaque instant. L'ensemble des contraintes 3.3 donne une borne inférieure sur le temps de complétude de chaque tâche dans tout ordonnancement réalisables. L'exactitude de cet ensemble de contraintes découle du lemme 11 qu'on décrit dans la suite. La partie droite de chaque inégalité dans 3.4 représente le temps d'exécution total de la tâche $J_j \in \mathcal{J}$ correspondante, et il s'agit, ainsi, d'une borne inférieure sur son temps de complétude. Finalement, les contraintes 3.5 impliquent que chaque tâche J_j peut être exécutée dans n'importe quel intervalle à condition

que sa date d'arrivée sur une machine quelconque ne dépasse pas la borne supérieure de l'intervalle en question.

Notons que dans LP^{ϵ}, le nombre de variables est polynomial en la taille de l'instance du problème. Notons, aussi, que le programme linéaire LP^{ϵ} est en effet une relaxation du problème $RS|\, r_{ij}|E + \sum w_j C_j$, puisqu'il autorise la préemption et l'exécution parallèle des tâches, *i.e.* une tâche peut être exécutée simultanément sur deux machines différentes. Finalement, le lemme suivant termine la preuve. □

Lemme 11 *Pour chaque tâche $J_j \in \mathcal{J}$, l'inégalité*

$$C_j^{LP} \geq \sum_{i=1}^{m} \sum_{v \in \mathcal{V}} \frac{1}{2} y_{ij0v} |I_0| \left(1 + \frac{1}{p_{ijv}}\right) + \sum_{i=1}^{m} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} \left(\frac{y_{ij\ell v}\, |I_\ell|}{p_{ijv}} (1+\epsilon)^{\ell-1} + \frac{1}{2} y_{ij\ell v} |I_\ell| \right)$$

est une contrainte valide dans la relaxation donnée par le programme linéaire LP^{ϵ}.

Preuve Dans [75], les auteurs ont présenté, dans un premier temps, une relaxation du problème $R|r_{ij}| \sum w_j C_j$ où les variables sont indéxées sur le temps et où la contrainte qui correspond à l'inégalité (3.3) de LP^{ϵ} a été donnée par :

$$C_j^{LP} \geq \sum_{i=1}^{m} \sum_{t=r_{ij}}^{T} \sum_{v \in \mathcal{V}} \left(\frac{y_{ijtv}}{p_{ijv}} (t + \frac{1}{2}) + \frac{1}{2} y_{ijtv} \right), \quad \text{pour tout } \; j = 1, \ldots, n, \tag{3.8}$$

où la variable y_{ijtv} représente la durée de temps octroyée à l'exécution de la tâche J_j durant l'intervalle de temps $(t, t+1]$ et sur la machine i lorsque celle-ci tourne à la vitesse $v \in \mathcal{V}$.

Considérons, par la suite, un ordonnancement quelconque réalisable σ dans lequel la tâche J_j est totalement exécutée et sans interruption sur une machine k entre les dates $C_j^{\sigma} - p_{kjs}$ et C_j^{σ}, C_j^{σ} étant la date de fin de la tâche J_j dans l'ordonnancement σ. Dans ce cas, la partie droite de l'inégalité (3.8) correspond à la date de fin exacte C_j^{σ} de la tâche J_j si on attribue des valeurs aux variables y_{ijtv} comme suit : $y_{ijtv} = 1$ si $i = k$, $v = s$ et $t \in \{C_j^{\sigma} - p_{kjs}, \ldots, C_j^{\sigma} - 1\}$, et $y_{ijtv} = 0$ sinon.

Finalement, nous remarquons que la partie droite de l'inégalité (3.8) domine celle de l'inégalité (3.3) puisque la borne inférieure t de tout intervalle $\tau = (t, t+1]$ dans la relaxation indéxée sur le temps est avancée à la borne inférieure $(1+\epsilon)^{\ell-1}$ de l'intervalle I_ℓ tel que $t \in I_\ell$ dans le programme linéaire LP^ϵ. □

3.4 A ´ ` $RS| \; r_{ij}|E \; + \sum w_j C_j$

Dans cette partie, nous nous inspirons de [75] afin de proposer un algorithme randomisé basé sur la relaxation linéaire décrite dans la section précédente. Cet algorithme, noté ,
tente de minimiser la somme des temps de complétude pondérés plus l'énergie consommée et il est décrit comme suit,

ALGORITHME

(1) Calculer une solution optimale y de LP^ϵ.

(2) Attribuer chaque tâche J_j à un triplet machine-intervalle-vitesse (i, I_ℓ, v) d'une façon indépendante et aléatoire en utilisant la probabilité $\frac{y_{ij\ell v}|I_\ell|}{p_{ijv}}$, fixer, ensuite, t_j à la valeur $\max\{r_{ij}, (1+\epsilon)^{\ell-1}\}$.

(3) Ordonnancer sur chaque machine i les tâches qui lui ont été affectées sans pré-emption et dans l'ordre croissant des valeurs t_j ; en cas d'égalité la sélection de la tâche se fait d'une façon indépendante et aléatoire. Chaque tâche est ordonnancée le plus tôt possible en tenant compte sa date d'arrivée.

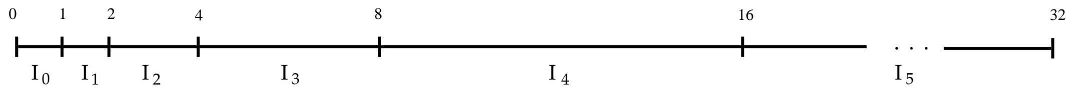
Considérons par exemple l'instance suivante : soit 4 tâches J_1, J_2, J_3 et J_4 et soit deux machines M_1 et M_2 telles que chaque machine puisse fonctionner avec deux vitesses différentes, *i.e.* $\mathcal{V} = \{v_1, v_2\}$. Le tableau 3.1 donne les différents paramètres relatifs à cette instance.

Dans l'exemple du tableau 3.1, l'horizon du temps est $T = 24$ et si nous fixons $\epsilon = 1$ alors

J_i	w_i	r_{1i}	r_{2i}	p_{1iv_1}	p_{1iv_2}	p_{2iv_1}	p_{2iv_2}	E_{1iv_1}	E_{1iv_2}	E_{2iv_1}	E_{2iv_2}
J_1	1	2	0	2	4	4	6	3	2	5	1
J_2	2	1	2	3	4	3	5	4	1	4	3
J_3	4	1	1	4	7	2	4	5	5	4	2
J_4	6	4	2	3	5	1	2	6	3	7	5

T . 3.1 – Exemple d'application de l'algorithme : instance avec 4 tâches, 2 machines et 2 vitesses différentes pour chaque machine.

nous obtenons six intervalles de tailles géométriques décrits dans la figure 3.1.



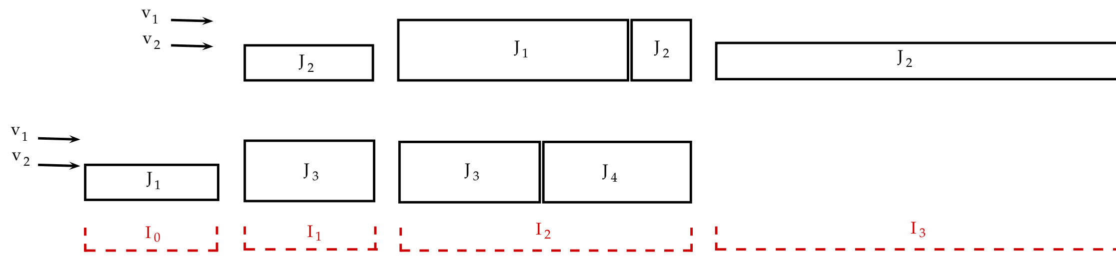
F . 3.1 – L'ensemble des intervalles géométriques déduits de l'instance du problème $RS|\, r_{ij}|E + \sum w_j C_j$ résumée dans le tableau 3.1.

En appliquant, étape par étape, l'algorithme à cette instance, nous obtenons une solution optimale du programme linéaire LP^1 donnée par $y = \{y_{1121} = \frac{5}{6}; y_{2102} = 1; y_{1212} = 1; y_{1222} = \frac{1}{6}; y_{1232} = \frac{2}{3}; y_{2311} = 1; y_{2321} = \frac{1}{2}; y_{2421} = \frac{1}{2}\}$.Cette solution est schématisée dans la figure 3.2.

Ensuite, une affectation aléatoire suivant les probabilités $\frac{y_{ij\ell v}|I_\ell|}{p_{ijv}}$ peut conduire à placer :

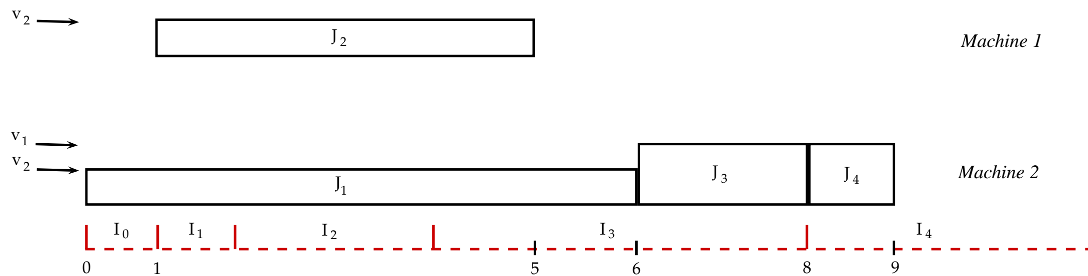
- la tâche J_1 sur la machine M_2, dans l'intervalle I_0 et à la vitesse v_2.
- la tâche J_2 sur la machine M_1 dans l'intervalle I_1 et à la vitesse v_2.
- la tâche J_3 sur la machine M_2 dans l'intervalle I_1 et à la vitesse v_1.
- la tâche J_4 sur la machine M_2 dans l'intervalle I_2 et à la vitesse v_1.

Ainsi, les valeurs t_j fixées pour chaque tâche sont données par : $\{t_1 = 0; t_2 = 1; t_3 = 1; t_4 = 2\}$



F . 3.2 – Solution optimale du programme linéaire appliqué sur l'instance du problème $RS|\, r_{ij}|E+\sum w_jC_j$ résumée dans le tableau 3.1.

et l'ordonnancement final est décrit dans la figure 3.3.



F . 3.3 – Solution retournée suite à l'exécution de l'algorithme sur l'instance du problème $RS|\, r_{ij}|E+\sum w_jC_j$ du tableau 3.1.

Dans la suite, nous étendons l'analyse présentée dans [75] pour inclure l'énergie consommée.

Lemme 12 *L'epérance sur le temps de complétude de chaque tâche $J_j \in \mathcal{J}$ dans l'ordonnancement construit par l'algorithme est au plus $2(1+\epsilon)C_j^{LP}$ pour tout $\epsilon > 0$. Dans le cas où toutes les tâches sont disponibles à la date 0, cette espérance est majorée par $3/2(1+\epsilon)C_j^{LP}$.*

Preuve Soit une instance quelconque du problème $RS|\, r_{ij}|E+\sum w_jC_j$, et soit une tâche $J_j \in \mathcal{J}$. Notons par (i, h, v_j) le triplet machine-intervalle-vitesse auquel la tâche J_j est affectée dans l'ordonnancement construit par l'algorithme et soit $t_j = \max\{r_{ij}, (1+\epsilon)^{h-1}\}$ la valeur attribuée à J_j. Par ailleurs, soit K l'ensemble des tâches exécutées sur la machine i durant l'intervalle de temps $(\tau, C_j]$, τ étant la date au plutôt telle qu'il n'existe pas de période d'inactivité dans

l'ordonnancement construit durant $(\tau, C_j]$. Supposons aussi que chaque tâche $J_k \in K$ est exécutée à une vitesse v_k. Ainsi, nous avons :

$$\sum_{J_k \in K} p_{ikv_k} = C_j - \tau. \tag{3.9}$$

Étant donné qu'aucune des tâches $J_k \in K$ ne commence après la tâche J_j, alors elles ont forcément été triées suivant les valeurs t_k attribuées par l'algorithme tel que $t_k \leq t_j$ $\forall J_k \in K$. En particulier, $r_{ik} \leq t_k \leq t_j$ $\forall J_k \in K$, et puisqu'il n'existe pas de période d'inactivité durant l'intervalle $(\tau, C_j]$, on a $\tau \leq t_j$ *(si $t_j < \tau$ alors il existerait une période d'inactivité durant $[t_j, \tau)$, or ceci n'est pas possible étant donné que les tâches sont exécutées au plus tôt et qu'on aurait dans ce cas $r_{ik} \leq \tau$ pout toute tâche $k \in K$)*. Avec l'égalité (3.9) nous obtenons :

$$C_j \leq t_j + \sum_{k \in K} p_{ikv_k}.$$

Afin d'analyser l'espérance sur le temps de complétude $\mathbb{E}[C_j]$ de la tâche J_j, nous commençons par fixer l'affectation de J_j au triplet machine-intervalle-vitesse (i, I_h, v_j), et prouvons, par la suite, l'existence d'une borne supérieure sur l'espérance conditionnelle $\mathbb{E}_{i,h,v_j}[C_j]$:

$$\mathbb{E}_{i,h,v_j}[C_j] \leq t_j + \mathbb{E}_{i,h,v_j}[\sum_{J_k \in K} p_{ikv_k}] \leq t_j + p_{ijv_j} + \sum_{J_k \neq J_j} \sum_{v \in \mathcal{V}} p_{ikv}.Pr[J_k \text{ affectée à } i \text{ avant } J_j \text{ à la vitesse } v]$$

$$= t_j + p_{ijv_j} + \sum_{J_k \neq J_j} \sum_{v \in \mathcal{V}} p_{ikv} \left(\sum_{\ell=0}^{h-1} \frac{y_{ik\ell v}|I_\ell|}{p_{ikv}} + \frac{1}{2} \frac{y_{ikhv}|I_h|}{p_{ikv}} \right). \tag{3.10}$$

Notons que le facteur $\frac{1}{2}$ qui précède le terme $\frac{y_{ikhv}|I_h|}{p_{ikv}}$ résulte de la séléction aléatoire en cas d'égalité des valeurs t_j. Ensuite, en utilisant les contraintes (3.2), nous obtenons :

$$\mathbb{E}_{i,h,v_j}[C_j] \leq t_j + p_{ijv_j} + \sum_{\ell=0}^{h-1} |I_\ell| + \frac{1}{2}|I_h|$$

Dans le cas où $h = 0$, en utilisant les contraintes 3.5 du LP^ϵ et comme les dates d'arrivée sont entières positives, nous avons $r_{ij} = 0$, donc $t_j = \max\{r_{ij}, 0\} = 0$. Ainsi,

$$t_j + p_{ijv_j} + \sum_{\ell=0}^{h-1} |I_\ell| + \frac{1}{2}|I_h| = p_{ijv_j} + \frac{1}{2}|I_0|$$

sinon ($h \geq 1$), nous avons :

$$t_j + p_{ijv_j} + \sum_{\ell=0}^{h-1} |I_\ell| + \frac{1}{2}|I_h| = t_j + p_{ijv_j} + (1 + \frac{\epsilon}{2})(1 + \epsilon)^{h-1}$$

(comme $t_j < (1 + \epsilon)^h$ pour tout $h = 0, \ldots, L$)

$$\mathbb{E}_{i,0,v_j}[C_j] \leq p_{ijv_j} + \frac{1}{2}$$

et si $h > 0$,

$$\mathbb{E}_{i,h,v_j}[C_j] \leq p_{ijv_j} + (1 + \epsilon)^h + (1 + \frac{\epsilon}{2})(1 + \epsilon)^{h-1} \leq p_{ijv_j} + 2(1 + \epsilon)^h$$

Finalement, l'espérance sur le temps de complétude de la tâche J_j peut être formulée comme suit :

$$\mathbb{E}[C_j] = \sum_{i=1}^{m} \sum_{\ell=0}^{L} \sum_{v \in \mathcal{V}} \frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}.\mathbb{E}_{i,\ell,v}[C_j]$$

$$= \sum_{i=1}^{m} \sum_{v \in \mathcal{V}} \frac{y_{ij0v}.|I_0|}{p_{ijv}}.\mathbb{E}_{i,0,v}[C_j] + \sum_{i=1}^{m} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} \frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}.\mathbb{E}_{i,\ell,v}[C_j]$$

$$\leq \sum_{i=1}^{m} \sum_{v \in \mathcal{V}} \frac{y_{ij0v}.|I_0|}{p_{ijv}}.(p_{ijv_j} + \frac{1}{2}) + \sum_{i=1}^{m} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} \frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}.(p_{ijv} + 2(1 + \epsilon)^\ell)$$

$$= \sum_{i=1}^{m} \sum_{v \in \mathcal{V}} y_{ij0v}.|I_0|.(1 + \frac{1}{2}.\frac{1}{p_{ijv}}) + \sum_{i=1}^{m} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} y_{ij\ell v}.|I_\ell| + 2(1 + \epsilon) \sum_{i=1}^{m} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} \frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}(1 + \epsilon)^{\ell-1}$$

$$= 2(1+\epsilon)\Big(\sum_{i=1}^{m} \sum_{v \in \mathcal{V}} \frac{1}{2(1 + \epsilon)}.y_{ij0v}.|I_0|.(1+\frac{1}{2}.\frac{1}{p_{ijv}})+\sum_{i=1}^{m} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} \frac{1}{2(1 + \epsilon)}.y_{ij\ell v}.|I_\ell|+\sum_{i=1}^{m} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} \frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}(1+\epsilon)^{\ell-1}\Big)$$

$$\leq 2(1+\epsilon)\Big(\sum_{i=1}^{m} \sum_{v \in \mathcal{V}} \frac{1}{2(1 + \epsilon)}.y_{ij0v}.|I_0|.(1+\frac{1}{p_{ijv}})+\sum_{i=1}^{m} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} \frac{1}{2(1 + \epsilon)}.y_{ij\ell v}.|I_\ell|+\sum_{i=1}^{m} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} \frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}(1+\epsilon)^{\ell-1}\Big)$$

$$\leq 2(1+\epsilon)\Big(\sum_{i=1}^{m}\sum_{v\in\mathcal{V}}\frac{1}{2}.y_{ij0v}.|I_0|.(1+\frac{1}{p_{ijv}})+\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{1}{2}.y_{ij\ell v}.|I_\ell|+\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}(1+\epsilon)^{\ell-1}\Big)$$

Cette dernière inégalité est valide étant donné que $\epsilon > 0$. En utilisant les contraintes (3.3) de LP^{ϵ}, on obtient :

$$\mathbb{E}[C_j] \leq 2(1+\epsilon)C_j^{LP}.$$

Maintenant, notons que si toutes les tâches sont relachées à la date 0 alors $\tau = 0$. En utilisant les mêmes arguments, nous avons :

$$\mathbb{E}_{i,0,v_j}[C_j] \leq p_{ijv_j} + \frac{1}{2}$$

dans le cas où $h > 0$,

$$\mathbb{E}_{i,h,v_j}[C_j] \leq p_{ijv_j} + (1+\frac{\epsilon}{2})(1+\epsilon)^{h-1} \leq p_{ijv_j} + (1+\epsilon)^{h}$$

Ensuite, l'espérance sur le temps de complétude peut être bornée par :

$$\mathbb{E}[C_j] \leq (1+\epsilon)\Big(\sum_{i=1}^{m}\sum_{v\in\mathcal{V}}\frac{1}{(1+\epsilon)}.y_{ij0v}.|I_0|.(1+\frac{1}{2}.\frac{1}{p_{ijv}})+\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{1}{(1+\epsilon)}.y_{ij\ell v}.|I_\ell|+\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}(1+\epsilon)^{\ell-1}\Big)$$

$$= (1+\epsilon)\Big(\sum_{i=1}^{m}\sum_{v\in\mathcal{V}}\frac{1}{2(1+\epsilon)}.y_{ij0v}.|I_0|.(2+\frac{1}{p_{ijv}})+\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{1}{(1+\epsilon)}.y_{ij\ell v}.|I_\ell|+\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}(1+\epsilon)^{\ell-1}\Big)$$

$$= (1+\epsilon)\Big(\frac{1}{2}\sum_{i=1}^{m}\sum_{v\in\mathcal{V}}\frac{1}{(1+\epsilon)}.y_{ij0v}.|I_0|+\frac{1}{2}\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{1}{(1+\epsilon)}.y_{ij\ell v}.|I_\ell|+\sum_{i=1}^{m}\sum_{v\in\mathcal{V}}\frac{1}{2(1+\epsilon)}.y_{ij0v}.|I_0|.(1+\frac{1}{p_{ijv}})+$$
$$\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{1}{2(1+\epsilon)}.y_{ij\ell v}.|I_\ell|+\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}(1+\epsilon)^{\ell-1}\Big)$$

étant donné que $\epsilon > 0$,

$$\leq (1+\epsilon)\Big(\frac{1}{2}\sum_{i=1}^{m}\sum_{v\in\mathcal{V}} y_{ij0v}.|I_0|+\frac{1}{2}\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}} y_{ij\ell v}.|I_\ell|+\sum_{i=1}^{m}\sum_{v\in\mathcal{V}}\frac{1}{2}.y_{ij0v}.|I_0|.(1+\frac{1}{p_{ijv}})+\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{1}{2}.y_{ij\ell v}.|I_\ell|+$$

$$\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}(1+\epsilon)^{\ell-1}\Big)$$

$$= (1+\epsilon)\Big(\frac{1}{2}\sum_{i=1}^{m}\sum_{\ell=0}^{L}\sum_{v\in\mathcal{V}} y_{ij\ell v}.|I_\ell|+\sum_{i=1}^{m}\sum_{v\in\mathcal{V}}\frac{1}{2}.y_{ij0v}.|I_0|.(1+\frac{1}{p_{ijv}})+\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{1}{2}.y_{ij\ell v}.|I_\ell|+\sum_{i=1}^{m}\sum_{\ell=1}^{L}\sum_{v\in\mathcal{V}}\frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}(1+\epsilon)^{\ell-1}\Big)$$

Finalement, les contraintes (3.4) et (3.3) conduisent à :

$$\mathbb{E}[C_j] \leq (1+\epsilon)(\frac{1}{2}C_j^{LP} + C_j^{LP}) = \frac{3}{2}(1+\epsilon)C_j^{LP}$$

. □

Notons par OPT la valeur optimale de la fonction objective définie par $OPT = E^* + \sum_{J_j\in\mathcal{J}} w_j C_j^*$, où E^* est la consommation d'énergie optimale et C_j^* est le temps de complétude optimal pour chaque tâche $J_j \in \mathcal{J}$.

Théorème 4 *Étant donnée une instance du problème $RS \mid r_{ij}|E + \sum w_j C_j$ et un réel $\epsilon > 0$, l'algorithme construit un ordonnancement tel que $\mathbb{E}[E + \sum w_j C_j] \leq 2(1+\epsilon)OPT$.*

Preuve Par le lemme 12, nous avons $\mathbb{E}[C_j] \leq 2(1+\epsilon)C_j^{LP}$. Par ailleurs, étant donné que les poids w_j sont positifs, il s'en suit que $\mathbb{E}[\sum_{J_j\in\mathcal{J}} w_j C_j] \leq 2(1+\epsilon)\sum_{J_j\in\mathcal{J}} w_j C_j^{LP}$.

Rappelons que E_{ijv} représente la quantité d'énergie que la machine i consomme si elle exécute la tâche J_j à la vitesse v. Soit E_j l'énergie totale consommée suite à l'exécution de la tâche J_j dans l'ordonnancement retourné par l'algorithme . Nous avons alors, $\mathbb{E}[E_j] = \sum_{i=1}^{m}\sum_{v\in\mathcal{V}} Pr[J_j$ affectée à la machine i à la vitesse $v] E_{ijv}$ avec $Pr[J_j$ affectée à la machine i à la vitesse $v] = \sum_{\ell=0}^{L}\frac{y_{ij\ell v}|I_\ell|}{p_{ijv}}$. Ainsi, $\mathbb{E}[E_j] = \sum_{i=1}^{m}\sum_{\ell=0}^{L}\sum_{v\in\mathcal{V}}\frac{y_{ij\ell v}|I_\ell|}{p_{ijv}}E_{ijv}$. L'espérance sur l'énergie totale consommée par toutes les machines dans l'ordonnancement est donnée par :

$\mathbb{E}[E] = \sum_{J_j \in \mathcal{J}} \mathbb{E}[E_j]$. Donc, $\mathbb{E}[E] = E^{LP}$ où E^{LP} est la consommation de l'énergie calculée par le programme linéaire LP^{ϵ}.

Comme le programme linéaire est une relaxation du problème $R|\, r_{ij}|E + \sum w_j C_j$, la solution optimale de LP^{ϵ} représente une borne inférieure pour la solution d'un ordonnancement optimal. Donc :

$$\mathbb{E}[E + \sum_{J_j \in \mathcal{J}} w_j C_j] \leq E^{LP} + 2(1+\epsilon) \sum_{J_j \in \mathcal{J}} w_j C_j^{LP}$$
$$\leq 2(1+\epsilon)[E^{LP} + \sum_{J_j \in \mathcal{J}} w_j C_j^{LP}] \leq 2(1+\epsilon)[E^* + \sum_{J_j \in \mathcal{J}} w_j C_j^*].$$

□

Pour le problème $RS\|E + \sum w_j C_j$ où toutes les tâches sont disponibles à la date 0, l'algorithme retourne une solution qui vérifie $\mathbb{E}[\sum_{j \in \mathcal{J}} w_j C_j] \leq \frac{3}{2}(1+\epsilon) \sum_{J_j \in \mathcal{J}} w_j C_j^{LP}$. Ce résultat peut être prouvé d'une manière simple en utilisant le lemme 12 et les mêmes arguments dans la preuve du théorème 4.

3.4.1 Dérandomisation

L'algorithme , décrit préalablement, est un algorithme randomisé qui retourne une solution réalisable dont la valeur de l'espérance de la fonction objectif est bornée en fonction de la solution optimale. Ainsi, ne fournit pas une garantie ferme sur la qualité de la solution retournée. Il est donc intéressant de concevoir un algorithme déterministe dans le but d'avoir une performance fixe dans le pire cas. Ceci peut être fait à l'aide de la dérandomisation qui consiste à séléctionner le meilleur choix des valeurs attribuées aux variables aléatoires qui minimise le coût global de la solution. Une méthode classique de dérandomisation appelée la méthode des *probabilités conditionnelles* considère les décisions aléatoires une à une et choisit toujours l'alternative la plus intéressante. Une alternative est dite plus intéressante si l'espérance conditionnelle sur la valeur de la solution correspondante est la plus petite possible.

Dans la suite, nous allons utiliser cette méthode pour dérandomiser l'algorithme où

l'objectif est de minimiser la somme des temps de complétude pondérés plus l'énergie.

Afin d'utiliser la technique des *probabilités conditionnelles*, nous avons besoin de calculer à chaque étape les valeurs exactes des espérances conditionnelles définies plus loin dans cette section. Nous avons, donc, besoin de faire une légère modification sur l'algorithme
en remplaçant sa dernière étape par :

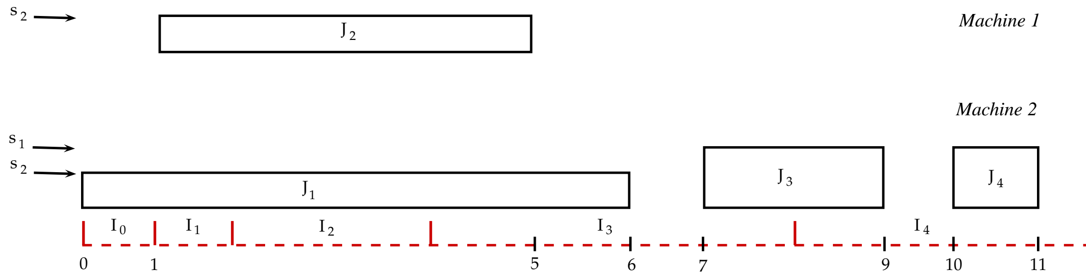
(3′) Ordonnancer sur chaque machine i les tâches qui lui ont été affectées sans préemption et dans l'ordre croissant des valeurs t_j ; en cas d'égalité la sélection de la tâche se fait d'une façon indépendante et aléatoire. Chaque tâche est ordonnancée le plus tôt possible en tenant compte de sa date d'arrivée et de telle manière à ce que, sur la machine i, la période d'inactivité totale avant le début d'exécution d'une tâche J_j, affectée à cette machine, soit égale à t_j.

Afin d'expliquer brièvement cette modification, supposons qu' on affecte un ensemble de tâches $J_1, \ldots, J_p$ à une machine i et aux vitesses $v_1, \ldots, v_p$ avec une valeur t_j attribuée à chaque tâche J_j. Supposons, aussi que ces tâches sont données dans l'ordre croissant des t_j. L'étape (3′), stipule qu'au début de l'exécution de chaque tâche J_j, la période totale d'inactivité sur la machine est égale à t_j. Ainsi l'ordonnancement retourné respecte les règles suivantes :

1. Exécuter la première tâche J_1 à la date t_1 à la vitesse v_1. Cette tâche se termine donc à la date $t_1 + p_{i1v_1}$.
2. Supposons, qu'à la fin d'une tâche J_j, la période totale d'inactivité sur la machine i est égale à t_j et soit f_j la date de fin de la tâche J_j. Maintenant, pour ordonnancer la tâche J_{j+1}, il faut considérer deux cas différents :

   - si $t_{j+1} \leqslant f_j$, alors exécuter la tâche J_{j+1} à la date $f_j + (t_{j+1} - t_j)$.
   - si $t_{j+1} > f_j$, alors exécuter la tâche J_{j+1} à la date $t_{j+1} + p_{ijv_j}$.

Si on applique cette modification sur l'ordonnancement retourné par pour



F . 3.4 – Solution retournée suite à l'exécution de l'algorithme modifié sur l'instance du problème $RS \mid r_{ij} \mid E + \sum w_j C_j$ du tableau 3.1.

l'instance du tableau 3.1 décrite dans les sections précédentes, nous obtenons un second ordonnancement qui, dans ce cas, s'écarte légèrement du premier. Ce nouvel ordonnancement est schématisé dans la figure 3.4.

Dans la preuve du lemme 12, nous avons borné le temps d'inactivité avant la date de début de la tâche J_j par t_j. Donc, cette analyse reste valable dans le cas de la modification apportée sur l'algorithme . L'avantage principal de la modification de l'étape (3) est la possibilité d'exprimer d'une façon précise les espérances sur les temps de complétude des tâches.

Soit y une solution optimale du programme linéaire LP^{ϵ} retournée par la première étape de l'algorithme . En utilisant les mêmes arguments que dans la preuve du lemme 12, l'algorithme modifié construit une solution pour laquelle l'espérance sur le temps de complétude de chaque tâche J_j est égale à :

$$\mathbb{E}[C_j] = \sum_{i=1}^{m} \sum_{\ell=0}^{L} \sum_{v \in \mathcal{V}} \frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}.(t_j + p_{ijv} + \sum_{J_k \neq J_j} \sum_{v \in \mathcal{V}} \sum_{h=0}^{\ell-1} y_{ikhv}.|I_h| + \sum_{J_k \neq J_j} \sum_{v \in \mathcal{V}} \frac{1}{2} y_{ik\ell v}.|I_\ell|)$$

Rappelons, aussi, l'espérance sur la consommation d'énergie pour chaque tâche J_j donnée dans la preuve du théorème 4 est :

$$\mathbb{E}[E_j] = \sum_{i=1}^{m} \sum_{\ell=0}^{L} \sum_{v \in \mathcal{V}} \frac{y_{ij\ell v}|I_\ell|}{p_{ijv}} E_{ijv}$$

Par ailleurs, nous définissons l'espérance conditionnelle sur le temps de complétude et sur la consommation d'énergie d'une tâche J_j si les tâches dans un sous ensemble $\mathcal{K} \subseteq \mathcal{J}$ ont déjà été

affectées à des triplets de machine-intervalle-vitesse. Pour chaque tâche $J_k \in \mathcal{K}$ on associe une variable binaire (0/1) $\chi_{ik\ell v}$ qui indique si la tâche J_k a été affectée à un triplet machine-intervalle-vitesse (i, ℓ, v) ($\chi_{ik\ell v} = 1$) ou pas ($\chi_{ik\ell v} = 0$). Ceci nous permet de donner les expressions suivantes pour formuler les espérances conditionnelles sur le temps de complétude et la consommation d'énergie des tâches J_j.

Dans le cas où $J_j \notin \mathcal{K}$ nous avons :

$$\mathbb{E}_{\mathcal{K},\chi}[Cj] = \sum_{i=1}^{m}\sum_{\ell=0}^{L}\sum_{v\in\mathcal{V}} \frac{y_{ij\ell v}.|I_\ell|}{p_{ijv}}.(t_j + p_{ijv} + \sum_{J_k\in\mathcal{K}}\sum_{v\in\mathcal{V}}\sum_{h=0}^{\ell-1}\chi_{ikhv}.p_{ikv} + \sum_{J_k\in\mathcal{K}}\sum_{v\in\mathcal{V}}\frac{1}{2}\chi_{ik\ell v}.p_{ikv}$$
$$+ \sum_{J_k\in\mathcal{J}\setminus(\mathcal{K}\cup\{J_j\})}\sum_{v\in\mathcal{V}}\sum_{h=0}^{\ell-1} y_{ikhv}.|I_h| + \sum_{J_k\in\mathcal{J}\setminus(\mathcal{K}\cup\{J_j\})}\sum_{v\in\mathcal{V}}\frac{1}{2}y_{ik\ell v}.|I_\ell|),$$

et,

$$\mathbb{E}_{\mathcal{K},\chi}[E_j] = \mathbb{E}[E_j].$$

Dans le cas où $J_j \in \mathcal{K}$, alors on a :

$$\mathbb{E}_{\mathcal{K},\chi}[Cj] = t_j + p_{ijv} + \sum_{J_k\in\mathcal{K}\setminus\{J_j\}}\sum_{v\in\mathcal{V}}\sum_{h=0}^{\ell-1}\chi_{ikhv}.p_{ikv} + \sum_{J_k\in\mathcal{K}\setminus\{J_j\}}\sum_{v\in\mathcal{V}}\frac{1}{2}\chi_{ik\ell v}.p_{ikv}$$
$$+ \sum_{J_k\in\mathcal{J}\setminus\mathcal{K}}\sum_{v\in\mathcal{V}}\sum_{h=0}^{\ell-1} y_{ikhv}.|I_h| + \sum_{J_k\in\mathcal{J}\setminus\mathcal{K}}\sum_{v\in\mathcal{V}}\frac{1}{2}y_{ik\ell v}.|I_\ell|,$$

et,

$$\mathbb{E}_{\mathcal{K},\chi}[E_j] = E_{ijv}.$$

où (i, ℓ, v) est le triplet machine-intervalle-vitesse auquel la tâche J_j est affectée, i.e. $\chi_{ij\ell v} = 1$, et $t_j = \max\{r_{ij}, (1 + \epsilon)^{\ell-1}\}$. Notons que l'espérance conditionnelle sur l'énergie consommée est égale à l'espérance non conditionnelle puisque la consommation de l'énergie d'une tâche dépend uniquement de son affectaion à une machine et une vitesse. Finalement, l'espérance conditionnelle

sur le temps de complétude plus l'énergie pour une tâche quelconque J_j peut être calculée par $\mathbb{E}_{\mathcal{K},\chi}[C_j + E_j] = \mathbb{E}_{\mathcal{K},\chi}[C_j] + \mathbb{E}_{\mathcal{K},\chi}[E_j]$.

Étant donnée une tâche $J_j \notin \mathcal{K}$, le lemme suivant montre qu'il existe toujours une affectation déterministe de J_j qui améliore l'espérance conditionnelle correspondante.

Lemme 13 *Soit une solution optimale y du programme linéaire LP^ϵ et soit $\mathcal{K} \subseteq \mathcal{J}$. Soit, aussi, une affectation fixe χ des tâches dans $\mathcal{K}$ à des triplets de machine-intervalle-vitesse. Considérons, par ailleurs, une tâche $J_j \in \mathcal{J} \backslash \mathcal{K}$. Il existe, alors, une affectation de la tâche J_j à un triplet de machine-intervalle-vitesse (i, ℓ, v) (*i.e. *$\chi_{ij\ell v} = 1$) avec $r_{ij} \leq (1+\epsilon)^{\ell-1}$ tel que :*

$$\mathbb{E}_{\mathcal{K}\cup\{J_j\},\chi}[\sum_{J_k \in \mathcal{J}} w_k C_k + E_k] \leq \mathbb{E}_{\mathcal{K},\chi}[\sum_{J_k \in \mathcal{J}} w_k C_k + E_k]. \tag{3.11}$$

Preuve L'espérance conditionnelle dans la partie droite de l'inégalité (3.11) peut être formulée comme étant une combinaison convexe des espérances conditionnelles $\mathbb{E}_{\mathcal{K}\cup\{j\},\chi}[\sum_{J_k \in \mathcal{J}} w_k C_k + E_k]$ sur toutes les affectations possibles de la tâche J_j à des triplets de machine-intervalle-vitesse (i, ℓ, v) comme suit :

$$\mathbb{E}_{\mathcal{K},\chi}[\sum_{J_k \in \mathcal{J}} w_k C_k + E_k] = \sum_{i=1}^{m} \sum_{\ell=0}^{L} \sum_{v \in \mathcal{V}} \Big(\mathbb{E}_{\mathcal{K}\cup\{J_j\},\chi}[\sum_{J_k \in \mathcal{J}} w_k C_k + E_k].\frac{y_{ik\ell v}.|I_\ell|}{p_{ikv}}\Big).$$

□

Finalement, la version dérandomisée de l'algorithme peut être décrite par :

ALGORITHM

(1) Calculer une solution optimale y de LP^{ϵ}.

(2) Fixer $\mathcal{K} := \emptyset$; $\chi = 0$; $\forall J_j \in \mathcal{J}$ faire

i) pour toute affectation possible de la tâche J_j à un triplet machine-intervalle-vitesse (i, ℓ, v) (i.e., $\chi_{ij\ell v} = 1$) calculer $\mathbb{E}_{\mathcal{K}\cup\{J_j\},\chi}[\sum_{J_k\in\mathcal{J}} w_k C_k + E_k]$;

ii) attribuer la tâche J_j au triplet machine-intervalle-vitesse (i, ℓ, v) qui minimise l'espérance conditionnelle $\mathbb{E}_{\mathcal{K}\cup\{J_j\},\chi}[\sum_{J_k\in\mathcal{J}} w_k C_k + E_k]$;

iii) fixer $\mathcal{K} := \mathcal{K} \cup \{J_j\}$.

(3) Ordonnancer sur chaque machine i les tâches qui lui ont été affectées sans préemption et dans l'ordre croissant des valeurs t_j ; en cas d'égalité, la sélection de la tâche se fait d'une façon indépendante et aléatoire. Chaque tâche est ordonnancée le plus tôt possible en tenant compte sa date d'arrivée et de manière à ce que, sur la machine i, la période d'inactivité totale avant le début d'exécution d'une tâche J_j, affectée à cette machine, soit égale à t_j.

Nous obtenons ainsi une version déterministe de l'algorithme et qui préserve la qualité des solutions retournées.

Théorème 5 *L'algorithme est un algorithme déterministe et polynomial dont la garantie de performance est au moins aussi bien que la garantie de performance espérée de l'algorithme probabiliste .*

Preuve Le théorème découle directement du lemme 13 et du théorème 4. □

3.5 E $RS|r_{ij}, E|\sum w_j C_j$

Dans cette partie, nous montrons qu'il est possible d'étendre l'approche précédente pour approcher le problème $RS|r_{ij}, E|\sum w_j C_j$. Ce problème consiste à ordonnancer, sous le même modèle, un ensemble des tâches $\mathcal{J}$ pour minimiser la somme des temps de complétude pondérés sans que l'énergie consommée ne dépasse une borne fixe B.

Il est possible de relaxer ce problème en considérant une légère modification du programme linéaire LP^{ϵ} précédemment décrit :

$$\min \sum_{J_j \in \mathcal{J}} w_j C_j^{LP}$$

$$\text{s.t.} \sum_{i=1}^{m} \sum_{\ell=0}^{L} \sum_{v \in \mathcal{V}} \frac{y_{ij\ell v}|I_\ell|}{p_{ijv}} = 1, \quad \text{pour tout } j, \tag{3.12}$$

$$\sum_{J_j \in \mathcal{J}} \sum_{v \in \mathcal{V}} y_{ij\ell v} \leq 1, \quad \text{pour tout } i \text{ et } \ell, \tag{3.13}$$

$$C_j^{LP} \geq \sum_{i=1}^{m} \sum_{v \in \mathcal{V}} \frac{1}{2} y_{ij0v}|I_0| \left(1 + \frac{1}{p_{ijv}}\right) + \sum_{i=1}^{m} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} \left(\frac{y_{ij\ell v}|I_\ell|}{p_{ijv}} (1+\epsilon)^{\ell-1} + \frac{1}{2} y_{ij\ell v}|I_\ell| \right), \quad \text{pour tout } j, \tag{3.14}$$

$$C_j^{LP} \geq \sum_{i=1}^{m} \sum_{\ell=0}^{L} \sum_{v \in \mathcal{V}} y_{ij\ell v}|I_\ell|, \quad \text{pour tout } j, \tag{3.15}$$

$$\sum_{i=1}^{m} \sum_{J_j \in \mathcal{J}} \sum_{\ell=0}^{L} \sum_{v \in \mathcal{V}} E_{ijv} \frac{y_{ij\ell v}|I_\ell|}{p_{ijv}} \leq B, \tag{3.16}$$

$$y_{ij\ell v} = 0, \quad \text{pour tout } i, j \ , \ (1+\epsilon)^{\ell} \leq r_{ij}, \quad \text{et } v, \tag{3.17}$$

$$y_{ij\ell v} \geq 0, \quad \text{pour tout } i, j \ , \ell, \text{et } v, \tag{3.18}$$

$$C_j^{LP} \geq 0, \quad \text{pour tout } j. \tag{3.19}$$

Tout d'abord, notons ce programme linéaire par $LP^{\epsilon}(B)$ et remarquons que la contrainte supplémentaire 3.16 a été ajoutée pour exprimer le fait que la consommation globale d'énergie ne doit pas dépasser un budget donné B. Dans la suite, nous allons appliquer l'algorithme probabiliste décrit précédemment et exhibons une borne probabiliste sur la somme des temps de complétude pondérés avec un dépassement du seuil d'énergie autorisé.

Soit $OPT(B)$ la valeur optimale de la fonction objective $\sum w_j C_j$ pour un seuil d'énergie fixé B.

Théorème 6 *Soit une instance du problème $RS|\ r_{ij}, E\ |\sum w_j C_j$ et un réel $\epsilon > 0$. L'algorithme*

construit un ordonnancement réalisable tel que $\mathbb{E}[\sum w_jC_j] \leq 2(1+\epsilon)OPT(B)$ *et* $\mathbb{E}[E] \leq B$.

Preuve Nous montrons, de la même façon que dans la preuve du théorème 4, que $\mathbb{E}[\sum_{J_j\in\mathcal{J}} w_jC_j] \leq 2(1+\epsilon)OPT(E)$ et que $\mathbb{E}[E] = \sum_{J_j\in\mathcal{J}} \mathbb{E}[E_j] = \sum_{J_j\in\mathcal{J}} \sum_{i=1}^{m} \sum_{\ell=0}^{L} \sum_{v\in\mathcal{V}} \frac{y_{ij\ell v}|I_\ell|}{p_{ijv}} E_{ijv}$. Nous concluons à l'aide de la contrainte 3.15 que $\mathbb{E}[E] \leq B$. □

Il faut noter que le théorème 6 ne garantit pas que ces bornes sur la somme des temps de complétude pondérés et sur l'énergie sont respectées simultanément pour toute instance du problème. Dans le but de pallier à cet inconvénient, nous proposons une garantie probabiliste sur les deux critères pour toute instance du problème.

Théorème 7 *Soit une instance du problème* $RS\,|\,r_{ij}, E\,|\sum w_jC_j$ *et soit deux réels* $u, w > 0$ *tels que* $\frac{1}{u} + \frac{1}{w} \leq 1$ *et un réel* $\epsilon > 0$. *L'algorithme construit un ordonnancement réalisable qui vérifie :*

$$Pr[\sum_j w_jC_j < 2u(1+\epsilon)OPT(B) \text{ et } E < wB] \geq 1 - \frac{1}{u} - \frac{1}{w}.$$

Preuve Rappelons qu'étant donnée une variable aléatoire X et un réel $a > 0$, l'inégalité de Markov stipule que $Pr[X \geq a] \leq \mathbb{E}(X)/a$. Ainsi, nous avons :

$$Pr[\sum_j w_jC_j \geq u\mathbb{E}(\sum_j w_jC_j)] \leq 1/u$$

et

$$Pr[E \geq w\mathbb{E}(E)] \leq 1/w.$$

Dans le théorème 4 nous avons montré que,

$$\mathbb{E}(\sum_j w_jC_j) \leq 2(1+\epsilon)OPT(B)$$

et

$$\mathbb{E}(E) \leq B,$$

d'où, $Pr[\sum_j w_j C_j \geq 2u(1+\epsilon)OPT(B)] \leq 1/u$ et $Pr[E \geq wB] \leq 1/v$. En utilisant l'inégalité de *Boole* [22], nous avons :

$$Pr[\sum_j w_j C_j \geq 2u(1+\epsilon)OPT(B) \text{ ou } E \geq wB] \leq 1/u + 1/w,$$

et finalement,

$$Pr[\sum_j w_j C_j < 2u(1+\epsilon)OPT(B) \text{ et } E < wB] \geq 1 - 1/u - 1/w.$$

□

3.6 C

Dans ce chapitre nous avons introduit une extension naturelle du problème d'ordonnancement classique sur des machines hétérogènes en tenant compte de l'énergie consommée. Nous avons, donc, considéré le modèle qui consiste à ordonnancer sur des machines hétérogènes, sans préemption, des tâches caractérisées par des dates d'arrivée, des durées d'exécution et des quantités d'énergie consommées. Dans ce modèle, nous avons introduit un ensemble fini de vitesses auxquelles les machines peuvent tourner et tel que chaque tâche soit exécutée d'une façon non-préemptive et à une vitesse constante. L'hétérogéneité des machines réside dans le fait que pour chaque tâche la durée d'exécution ainsi que la quantité d'énergie consommée dépendent de la machine et de la vitesse d'exécution. En revanche, les dates d'arrivées des tâches dépendent uniquement des machines auxquelles elles sont affectés.

Dans un premier temps, nous avons considéré le problème baptisé $RS|r_{ij}|E + \sum w_j C_j$ qui consiste à minimiser la somme des temps de complétude pondérés plus l'énergie totale consommée par toutes les machines. Nous avons prouvé que l'analyse de la performance de l'algorithme randomisé présenté dans [75] et basé sur la technique de l'arrondi aléatoire peut être étendue pour le problème en question et que la qualité de la solution retournée reste préservée avec ou sans des dates d'arrivée. Nous avons aussi dérandomisé cet algorithme en utilisant la méthode des *probabilités conditionnelles*.

Dans un deuxième temps, nous avons condidéré le problème $RS|r_{ij}, E|\sum w_jC_j$ dans lequel nous fixons un seuil d'énergie total à ne pas dépasser et nous cherchons à minimiser la somme des temps de complétude pondérés. En utilisant le même algorithme randomisé, nous avons proposé une borne probabiliste intéressante sur la fonction objectif avec un dépassement du seuil d'énergie.

Pour ce dernier problème, *i.e.* $RS|r_{ij}, E|\sum w_jC_j$ et d'un point de vue pratique, une question intéressante serait de savoir s'il est possible de concevoir un moyen efficace de dérandomisation permettant de préserver les bornes obtenues. Dans ce contexte, la difficulté réside dans l'aspect antagoniste entre la minimisation du temps de complétude et l'énergie consommée, et aussi dans l'absence de corrélation entre ces deux critères dans le modèle étudié. Ainsi, la méthode de dérandomisation basée sur les *probabilités conditionnelles* s'avère inefficace étant donné que pour faire un choix à une étape donnée, la notion de solution intéressante n'est pas définie.

Par ailleurs, nous soulignons que Skutella a proposé dans [77] une approche alternative basée sur la relaxation quadratique et sur la même technique d'arrondi aléatoire pouvant améliorer la qualité de la solution du problème de minimisation de la somme des temps de complétude pondérés. Ainsi, une autre question intéressante est de savoir si cette approche pourrait être adaptée dans le cas des problèmes traités dans ce chapitre.

Finalement, une perspective de cette étude consiste à considérer le cas où chaque machine peut choisir sa vitesse dans un spectre continu de valeurs réelles et positives. Il serait, ainsi nécessaire de définir des fonctions continues de la vitesse décrivant, d'une part, l'évolution des temps d'exécution des tâches, et, d'autre part, l'évolution des quantités d'énergie consommées par chaque tâche.

C 4

O

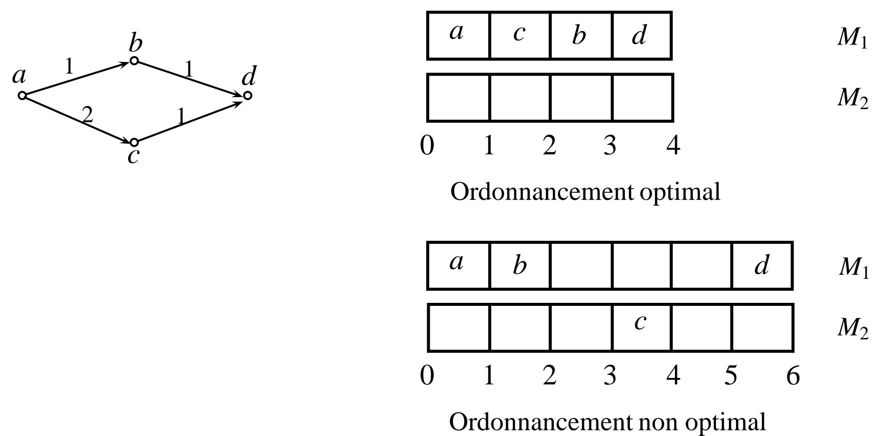
´ ´

4.1 I

Le déploiement élargi des architectures parallèles a fait que les problèmes d'ordonnancement sur des machines parallèles soient au coeur du développement et de l'efficacité de ces architectures. Ce type de problèmes a été sujet de recherches intensives depuis plusieurs années [46]. En particulier, les problèmes d'ordonnancement qui visent à minimiser le temps d'exécution total (*makespan*), sur des machines identiques et en présence de contraintes de précédence ont été largement étudiés. Dans ce type de problèmes, les contraintes de précédence sont modélisées par un graphe orienté et acyclique *DAG* dont les sommets représentent les tâches à ordonnancer et les arcs représentent les contraintes de précédence. Ainsi, s'il existe un arc partant d'une tâche a vers une tâche b dans le graphe de précédence : $a \rightarrow b$, alors l'exécution de b ne peut commencer que si l'exécution de a prend fin. Généralement, les contraintes de précédence sont issues de la parallélisation d'un programme séquentiel et traduisent des dépendances de données entre les tâches, *i.e.* une tâche ne peut commencer que si ses prédecesseurs lui fournissent des données nécessaires pour son exécution. La figure 4.1 montre un ordonnancement possible sur deux machines dans un graphe de précédence avec 4 tâches unitaires. Notons que cet ordonnancement est optimal par rapport à la minimisation du *makespan*.

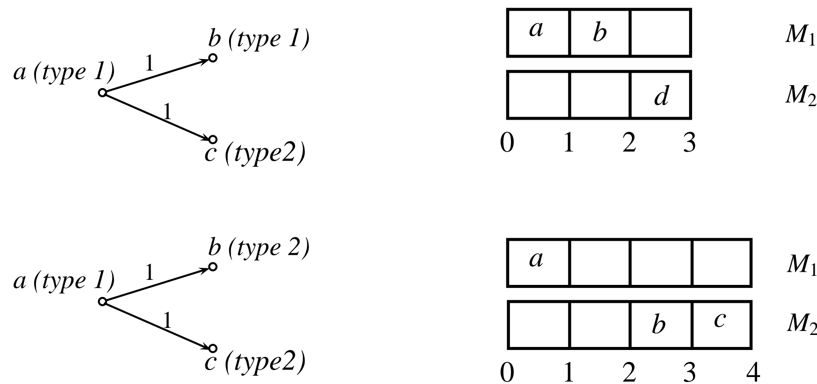
F . 4.1 – Ordonnancement d'un graphe de précédence avec 4 tâches unitaires : les tâches a, b et d s'exécutent sur la machine M_1 et la tâche c sur la machine M_2.

Un modèle plus réaliste considère des délais supplémentaires nécessaires au tranfert des données entre deux tâches dans le cas où elles sont reliées par une contrainte de précédence et sont exécutées sur deux machines différentes. Il s'agit en effet d'un délai qui doit s'écouler à la suite de l'exécution d'une tâche et avant que l'exécution de son successeur ne puisse commencer sur une autre machine. La valeur de ce délai, connu sous le nom de *temps de communication*, dépend de la paire des tâches considérée et elle est nulle si les tâches successives s'exécutent sur la même machine. Dans la figure 4.2, nous reprenons l'exemple de la figure précédente en imposant des temps de communication unitaires pour les paires des tâches (a, b), (b, d) et (c, d), et un temps de communication égal à 2 pour la paire de tâches (a, c). Nous considérons la minimisation du *makespan* et présentons deux solutions : une solution optimale et une autre non optimale. Notons, aussi, que ces temps de communication sont dits *grands* si leurs valeurs dépassent les durées d'exécution des tâches.



F . 4.2 – Ordonnancement d'un graphe de précédence en présence des temps de communication entre les tâches. Les arcs sont pondérés par les temps de communication.

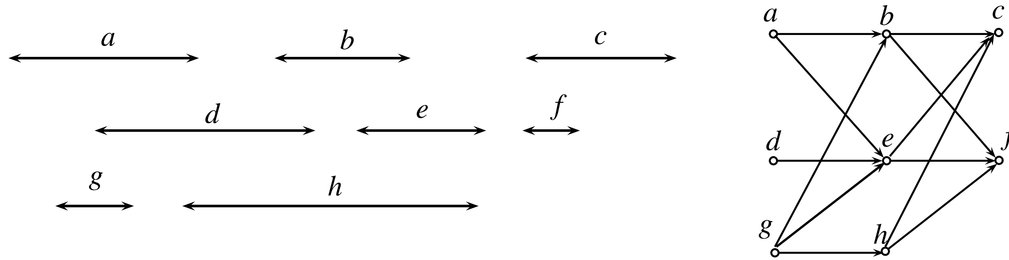
Par ailleurs, nous considèrons un système de tâches typées où chaque tâche est caractérisée par un type fixé à l'avance et qui détermine une seule machine sur laquelle la tâche peut s'exécuter. On parle alors d'un modèle de machines dédiées où chaque machine est spécialisée dans l'exécution d'un seul type de tâche et tel qu'il existe une seule machine par type. Dans ce cas, l'affectation des tâches aux machines est connue au préalable ce qui permet de savoir à l'avance si les temps de communication s'appliqueront ou non entre toute paire de tâches reliées par un arc de précédence. La figure 4.3 montre deux exemples d'ordonnancement de 3 tâches typées sur 2 machines dédiées avec des temps de communication unitaires. Dans le premier exemple, les tâches a et b sont de type 1 et la tâche c est de type 2. Dans le second exemple, la tâche a est de type 1 et les tâches b et c de type 2.



F . 4.3 – Ordonnancement d'un graphe de précédence avec 3 tâches unitaires typées, des temps de communication unitaires et 2 machines dédiées : la machine M_1 exécute les tâches de type 1, et la machines M_2 exécute les tâches de type 2.

Nous nous intéressons, dans ce chapitre au problème d'ordonnancement d'un ensemble de tâches unitaires et typées sur des machines dédiées avec des contraintes de précédence et des grands temps de communication entre les tâches. Aussi, nous allons nous focaliser sur un type particulier de graphe de précédence, à savoir les *ordres d'intervalles*. À notre connaissance, cette classe de graphes a été introduite par Papadimitriou et Yannakakis dans [71]. En effet, un graphe est un ordre d'intervalles s'il est possible d'associer chacun de ses sommets à un intervalle de réels de telle sorte à ce que deux sommets soient reliés par un arc si et seulement si les intervalles

correspondants ne se chevauchent pas. En d'autres termes, si l'intervalle $[a_1, a_2)$ (*resp.* $[b_1, b_2)$) est associé à la tâche a (*resp.* b), alors la tâche a précède la tâche b dans le graphe de précédence si et seulement si $a_2 < b_1$. Un exemple montrant un ensemble d'intervalles et le graphe d'ordre d'intervalles correspondant est illustré dans la figure 4.4.



F . 4.4 – Un graphe d'ordre d'intervalles et les intervalles correspondants. Les arcs transitifs ne sont pas représentés.

Pour un ensemble de tâches caractérisées par des dates d'arrivée et des dates d'échéance, le but sera, donc, de construire un ordonnancement réalisable.

4.2 D´ N

4.2.1 Formulation du modèle

Nous considérons un ensemble $\mathcal{J}$ constitué de n tâches typées avec des durées d'exécution unitaires, des dates d'arrivée $r_i \in \mathbb{N}$ et des dates d'échéance $d_i \in \mathbb{N}$, pour toute tâche $J_i \in \mathcal{J}$. D'autre part, nous considérons K types de tâches tels que chaque tâche $J_i \in \mathcal{J}$ soit caractérisée par un type noté $m_i \in \{1, \ldots, K\}$. Nous limitons cette étude à un graphe de précédence orienté et acyclique $G = (\mathcal{J}, \mathcal{A})$ ayant la structure d'ordre d'intervalles : à chaque tâche $J_i \in \mathcal{J}$ on associe un intervalle non vide $I_i = [a_i, b_i)$ avec $(a_i, b_i) \in \mathbb{R}^2$, $a_i < b_i$. Comme il a été expliqué dans l'introduction de ce chapitre, il existe une relation de précédence entre deux tâches $J_i, J_j \in \mathcal{J}$, *i.e.* $(J_i, J_j) \in \mathcal{A}$, si $b_i < a_j$ (voir figure 4.4). Dans ce cas, un temps de communication entier $c_{ij} \in \mathbb{N}$

est associé à l'arc (J_i, J_j).

Nous considérons, également, K classes de machines telles que chaque classe $p \in \{1, \ldots, K\}$ contienne exactement une seule machine dédiée à l'exécution des tâches de type p. Ainsi, un ordonnancement est défini uniquement par le vecteur $\sigma = (t)$, où t_i représente la date de début d'exécution de la tâche J_i. et il est réalisable si les trois conditions suivantes sont vérifiées :

1. Les dates de début d'exécution des tâches respectent les dates d'arrivée et les dates d'échéance correspondantes, *i.e.* $\forall J_i \in \mathcal{J}, r_i \leq t_i < d_i$.

2. Les relations de précédence entre les tâches sont aussi respectées, *i.e.* $\forall (J_i, J_j) \in \mathcal{A}$, soit $x_{ij} = 0$ si $m_i = m_j$, et $x_{ij} = 1$ *sinon*. L'inégalité suivante doit être vérifiée, $t_i + 1 + x_{ij}c_{ij} \leq t_j$.

3. À chaque instant t, au plus une tâche est exécutée par chaque machine.

Le problème général que nous considérons consiste à construire un ordonnancement réalisable pour un graphe de précédence ayant la structure d'ordre d'intervalles avec des temps de communication, des dates d'arrivée, des dates d'échéance et des machines dédiées.

4.2.2 Temps de communication monotones

Pour chaque tâche $J_i \in \mathcal{J}$, on note par $\Gamma^+(i)$ (*resp.* $\Gamma^-(i)$) l'ensemble des successeurs (*resp.* prédécesseurs) de la tâche J_i dans le graphe G.

Propriété 1 *[71] Soit un graphe de précédence $G = (\mathcal{J}, \mathcal{A})$ ayant la structure d'ordre d'intervalles. Pour toute paire de tâches $J_i, J_j \in \mathcal{J}$, $\Gamma^-(i) \subseteq \Gamma^-(j)$ ou $\Gamma^-(j) \subseteq \Gamma^-(i)$.*

Définition 6 *Soit un graphe de précédence $G = (\mathcal{J}, \mathcal{A})$ ayant la structure d'ordre d'intervalles. Les temps de communication associés à G sont dits monotones si*

$$\forall ((J_i, J_j), (J_i, J_k)) \in \mathcal{A}^2, \Gamma^-(j) \subseteq \Gamma^-(k) \Rightarrow c_{ij} \leq c_{ik}.$$

Il faut noter que $\Gamma^-(j) \subseteq \Gamma^-(k)$ est équivalent à $a_j \leq a_k$, donc à $a_j - b_i \leq a_k - b_i$. Ainsi, la monotonie des temps de communication implique que pour tout arc $(J_i, J_j) \in \mathcal{J}$, le temps de communication c_{ij} est croissant en fonction de la distance entre la borne supérieure b_i de l'intervalle I_i et la borne inférieure a_j de l'intervalle I_j.

En reprenant l'exemple du graphe d'ordre d'intervalles illustré dans la figure 4.4, nous présentons dans le tableau 4.1 des temps de communication monotones possibles entre les tâches avec des contraintes de précédence.

c_{ij}	b	c	e	f	h
a	1	3	2	3	*
b	*	2	*	2	*
d	*	2	1	2	*
e	*	1	*	1	*
g	2	3	2	3	2
h	*	1	*	1	*

T . 4.1 – Temps de communication monotones pour le graphe d'ordre d'intervalles présenté dans la figure 4.4

4.2.3 Algorithmes de liste

Les algorithmes de liste sont des algorithmes gloutons dont le principe est assez intuitif. Une liste de priorité est une bijection $L : \mathcal{J} \rightarrow \{1, \ldots, n\}$. Pour les problèmes d'ordonnancement, cette liste est souvent construite à l'aide de la structure d'un graphe, des dates d'arrivée, des dates d'échéance et des contraintes de ressources. La détermination d'une liste de priorité pour notre problème d'ordonnancement utilise une extension de l'algorithme *LPP* [62] et sera abordée dans la Section 4.5.

Supposons qu'une liste de priorité L est fixée. Un ordonnancement de liste avec des temps de communication peut être construit comme il est décrit dans [47] : soit une tâche $J_i \in \mathcal{J}$ telle que sa date de début d'exécution t_i ne soit pas encore fixée à l'instant t. La tâche J_i est dite prête à être

exécutée à l'instant t par la machine π de classe si :

1. $t \geq r_i$;

2. Chaque tâche $J_j \in \Gamma^-(i)$ est ordonnancée avant la date t. Par ailleurs, si une tâche $J_k \in \Gamma^-(i)$ n'est pas exécutée sur la machine π alors le temps de communication entre J_i et J_k est pris en compte si la tâche J_i est ordonnancée à l'instant t ;

3. La machine π est inactive à l'instant t.

Pour chaque couple de valeurs (t, p), pour $t \in \mathbb{N}$ et $p \in \{1, \ldots, K\}$, l'algorithme de liste ordonnance, sur la machine de la classe p, la tâche prête de type p possédant la plus haute priorité (*i.e.* la valeur $L(J_i)$ est minimale) en fixant $t_i = t$ si une telle tâche existe. Le déroulement de l'algorithme débute à l'instant $t = 0$ et incrémente la valeur de t dès qu'aucune tâche n'est prête et/ou dès qu'aucune machine n'est disponible à l'instant t.
Il faut noter que si le pas du temps est unitaire alors l'algorithme de liste, décrit ci-dessus, possède une complexité non polynomiale égale à $O(nK. \max_{(J_i,J_j)\in\mathcal{A}}\{c_{ij}\})$. Cependant, il a été prouvé dans [47] que les valeurs prises par t peuvent être limitées de façon à obtenir un algorithme de liste polynomiale dont la complexité est égale à $O(nK^2)$.

4.3 T

Il existe beaucoup de travaux dans la littérature portant sur l'ordonnancement d'un ensemble de tâches partiellement ordonnées et modélisées par un graphe de précédence orienté acyclique [59, 25, 40]. Pour un graphe de précédence quelconque, le problème de minimisation du *makespan* sur des machines identiques est *NP-difficile* [40]. Cependant, Papadimitriou et al. [71] ont construit une solution optimale dans le cas où le graphe de précédence possède une structure d'ordre d'intervalles. Plus récemment, Jansen [51] a étendu ce résultat pour des systèmes de tâches typées. Dans le modèle le plus général des tâches typées, les machines sont partitionnées en plusieurs classes, chaque tâche doit être exécutée sur une machine d'une classe fixée, et chacune de ces classes peut contenir plusieurs machines. Il est à noter qu'un système de tâches typées inclut les modèles de machines identiques (une seule classe de machines) et de machines dédiées

(une seule machine par classe).

Par ailleurs, l'introduction des temps de communication entre les tâches rajoute une couche de complexité supplémentaire au problème même dans le cas de graphes de précédence particuliers tels que les ordres d'intervalles. Rappelons dans ce contexte que les temps de communication sont dits grands s'ils sont plus grands que les temps d'exécution des tâches. Les problèmes d'ordonnancement sous cette hypothèse sont très difficiles à résoudre de façons exactes. Par exemple, pour des temps d'exécution unitaires et un délai de communication fixe égale à $c \geqslant 3$, Giroudeau et al. ont prouvé dans [41] que l'existence d'un ordonnancement de taille inférieure à $c + 4$ est *NP-complet* pour un graphe de précédence général sans limitation de ressources. Dans la littérature, des articles de synthèse sur les problèmes d'ordonnancement avec des temps de communication peuvent être trouvés dans [35, 80]. Il faut noter que les temps de communication monotones incluent les délais constants.

Dans ce contexte, la plupart des auteurs limitent leurs études à des durées d'exécution et des temps de communication unitaires. Ali et Rewini [8] ont développé un algorithme polynomial pour la minimisation du *makespan* pour un graphe de précédence d'ordre d'intervalles, m machines identiques et des temps de communication unitaires. Verriet [80] a développé un algorithme polynomial dans le but de résoudre le problème de détermination d'ordonnancements réalisables pour un graphe de précédence d'ordre d'intervalles avec des dates d'arrivée, des dates d'échéance, des temps de communication unitaires et m machines identiques.

Dans ce chapitre, nous étudions le problème de détermination d'un ordonnancement réalisable pour un graphe de précédence d'ordre d'intervalles avec des grands temps de communication, des dates d'arrivée, des dates d'échéance et des tâches typées. Nous montrons dans la section 4.4 que le problème est NP-complet au sens fort pour deux machines dédiées et des temps de communication quelconques. Dans la suite du chapitre, nous considérons uniquement des temps de communication monotones. Dans la section 4.5, nous nous intéressons au problème d'ordonnan-

cement d'un ensemble de tâches modélisées par un graphe de précédence d'ordre d'intervalles avec des dates d'arrivée, des dates d'échéance, des temps de communication monotones et des machines dédiées. Pour celà, nous présentons une extension de l'algorithme décrit dans [62] par Leung, Palem et Pnuelli (algorithme *"LPP"* en abréviation) où les auteurs décrivent un algorithme générique capable de déterminer éfficacement un ordonnancement pour un ensemble de tâches avec des dates d'arrivée et d'échéance, sur m machines identiques et en présence d'un graphe de précédence d'ordre d'intervalles avec des délais de latence. Il faut noter, dans ce contexte, que le délai de latence est une quantité de temps qui doit s'écouler entre la fin d'une tâche et le début de son successeur et que sa valeur est indépendante des machines sur lesquelles les deux tâches sont exécutées [26]. En effet, l'algorithme d'ordonnancement *LPP* est un algorithme de liste qui calcule ses priorités en fonction des dates d'échéance des tâches. Dans cette perspective, la priorité est donnée à la tâche qui possède la date d'échéance la plus courte. L'étape de construction de la liste des priorités sera précédée par une étape de modification des dates d'échéance qui consiste à relaxer le problème d'origine et à résoudre un sous-problème de maximisation des dates de début d'exécution des tâches. Il s'agit, en effet, de calculer et d'attribuer une nouvelle date d'échéance à chaque tâche, moyennant une procédure de calcul au plus tard. Le schéma de modification des dates d'échéance présenté dans [62] ainsi que les preuves seront aussi simplifiés. Dans la section 4.6, nous prouvons brièvement que l'algorithme *LPP* ne peut pas s'étendre pour m machines identiques puisque, dans ce cas, les algorithmes de listes ne peuvent pas fournir des solutions optimales pour de grands temps de communication.

4.4 C ´ ` ´ ´

Dans cette section, nous prouvons que le problème d'ordonnancement sur deux machines dédiées d'un graphe d'ordre d'intervalles avec des temps de communication quelconques est *NP*-complet au sens fort. Ce résultat justifie la limitation de notre étude aux temps de communication monotones. Ainsi, nous considérons les deux problèmes définis comme suit :

SCHEDULING BIPARTITE GRAPH WITH COMM. DELAYS (SBC)

Entrée : Soient $A = \{J_1, \dots, J_k\}$ et $B = \{J_{k+1}, \dots, J_{2k}\}$, $k \in \mathbb{N}^*$ deux ensembles de k tâches de durées unitaires. $\forall (J_i, J_j) \in A \times B$, un délai $c_{ij} \in \mathbb{N}$. $\forall J_i \in A \cup B$, $m_i \in \{1, 2\}$. 2 machines dédiées. Un entier $D \geq 0$.

Question : Existe-il un vecteur $t = (t_i)$, $\forall J_i \in A \cup B$ tel que, $\forall (J_i, J_j) \in (A \cup B)^2$, si $m_i = m_j$ alors $t_i \neq t_j$, $\max_{J_i \in A \cup B}\{t_i\} < D$, et $\forall (J_i, J_j) \in A \times B$, $x_{ij} = 0$ si $m_i = m_j$, $x_{ij} = 1$ sinon, et $t_i + x_{ij}c_{ij} + 1 \leq t_j$?

SEQUENCING COUPLES WITH LATENCIES (SCL)[84]

Entrée : Soient $A = \{J_1, \dots, J_k\}$ et $B = \{J_{k+1}, \dots, J_{2k}\}$, $k \in \mathbb{N}^*$ deux ensembles de k tâches de durées unitaires. k valeurs $\ell_i \in \mathbb{N}$, $i \in \{1, \dots, k\}$. Une seule machine. Un entier $D \geq 0$.

Question : Existe-il un vecteur $t = (t_i)$, $\forall J_i \in A \cup B$ tel que, $\forall (J_i, J_j) \in (A \cup B)^2$, $t_i \neq t_j$, $\max_{J_i \in A \cup B}\{t_i\} < D$ et $\forall i \in \{1, \dots, k\}$, $t_i + 1 + \ell_i \leq t_{i+k}$?

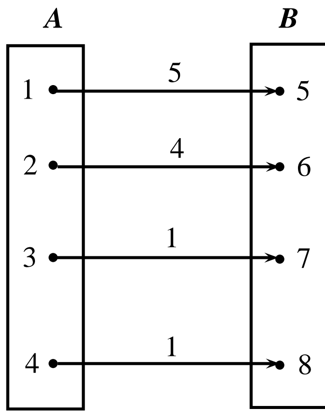
Théorème 8 *Il existe une réduction polynomiale du problème SCL vers le problème SBC.*

Preuve Soit Π une instance du problème SCL. Une instance $f(\Pi)$ du problème SBC est construite en fixant, $\forall (J_i, J_j) \in A \times B$, $c_{ij} = \ell_i$ si $j = k + i$ et $c_{ij} = 0$ sinon. En plus, $\forall J_i \in A, m_i = 1$ et $\forall J_i \in B, m_i = 2$. Cette transformation est illustrée dans la figure 4.5.

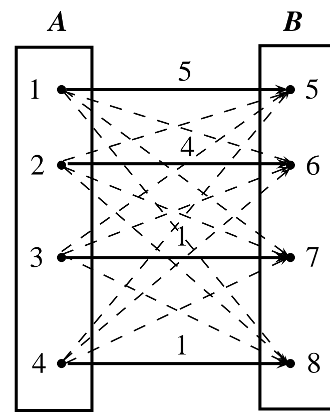
1. Soit $t = (t_i)$, $\forall J_i \in A \cup B$ une solution de l'instance Π. Nous montrons que toutes les tâches dans A peuvent être exécutées entièrement et sans interruption dans l'intervalle de temps $[0, k]$:

   - Supposons qu'il existe deux tâches successives J_i et J_j avec $J_i \in B$ et $J_j \in A$. Étant données les relations de précédence entre les tâches de A et les tâches de B, il est possible de permuter les tâches J_i et J_j sans influencer l'exécution des autres tâches. Ainsi, nous pouvons supposer que toutes les tâches de l'ensemble A peuvent être exécutées avant les tâches de l'ensemble B. Au pire cas, le nombre de permutations est borné par k^2.
   - Les tâches de l'ensemble A n'ont pas de prédécesseurs. Il est donc possible de les exécuter sans interruption.

   Ainsi $t = (t_i)$, $\forall J_i \in A \cup B$, est aussi une solution de $f(\Pi)$.



Instance Π du problème SCL



Instance $f(\Pi)$ du problème SBC

F . 4.5 – Transformation d'une instance du problème SCL en une instance du problème SBC avec des temps de communication.

2. Inversement, supposons que $t = (t_i)$, $\forall J_i \in A \cup B$ est une solution de $f(\Pi)$. Par construction du graphe, la première tâche de l'ensemble B est exécutée à la suite de la dernière tâche de l'ensemble A. Ainsi, pour tout couple de tâches $(J_i, J_j) \in (A \cup B)^2$, $t_i \neq t_j$ et $t = (t_i)$, $\forall J_i \in A \cup B$ est aussi une solution réalisable pour Π.

□

Finalement, il faut noter qu'un graphe biparti complet constitue clairement un ordre d'intervalles tel que chaque tâche dans A (*resp.* dans B) soit associée à un intervalle $I = [a, b)$ (*resp.* $I' = [a', b')$) avec $b \leq a'$. D'autre part, le problème SBC est clairement dans NP. Du fait que le problème SCL est NP-complet au sens fort [84], on a le corollaire suivant :

Corollaire 3 *Le problème d'ordonnancement pour des graphes de précédence d'ordre d'intervalles, avec des temps de communication quelconques et sur* 2 *machines dédiées est* NP*-complet au sens fort.*

4.5 E *LPP* ´ ´

L'objectif de cette section est de présenter une extension de l'algorithme *LPP* [62] pour un graphe de précédence d'ordre d'intervalles avec des temps de communication monotones et des machines dédiées. Nous présentons, tout d'abord, un algorithme polynomial simple pour calculer une borne supérieure sur le temps de complétude d'une tâche. Cet algorithme est basé sur la régle de *Jackson*[1] [50, 46], et sera par la suite appliqué d'une façon itérative dans le but d'améliorer la date d'échéance de chaque tâche. Finalement, nous prouvons qu'un algorithme de liste basé sur la priorité *EDF* (Earliest Deadline First) ainsi que sur les dates d'échéance modifiées mène à une solution réalisable si elle existe.

4.5.1 Une borne supérieure sur les temps de complétude des tâches

Soit $J_k \in \mathcal{J}$ et $S_k = \Gamma^+(k) \cup indep(k)$, où $indep(k)$ représente l'ensemble des tâches dans $\mathcal{J}$ sans relations de précédence avec la tâche J_k. Supposons que chaque tâche $J_\ell \in S_k \cup \{J_k\}$ possède une date d'arrivée r_ℓ et une date d'échéance d_ℓ. D'autre part, étant donné qu'il existe une seule machine par type de tâches, nous fixons, pour chaque arc $(J_i, J_j) \in \mathcal{A}$, $x_{ij} = 0$ si $m_i = m_j$, $x_{ij} = 1$ sinon.

Soit $t \in \{r_k + 1, \ldots, d_k\}$. Nous considérons, alors, le problème de décision $Existence(t, k, r, d)$ où r (*resp.* d) représente le vecteur des dates d'arrivée (*resp.* d'échéance) de l'ensemble des tâches $S_k \cup \{J_k\}$. Ce problème est défini comme suit : il s'agit de fixer un ordonnancement de la tâche J_k à l'instant $t - 1$ et de retourner la valeur *"vrai"* s'il existe un ordonnancement réalisable des tâches de l'ensemble $S_k \cup \{J_k\}$ en considérant les dates d'arrivée, les dates d'échéance, les contraintes de ressources et les relations de précédence entre la tâche J_k et ses successeurs.

Existence(t,k,r,d) :

Entrée : L'ensemble des tâches $S_k \cup \{J_k\}$ avec des durées d'exécution unitaires, des dates d'échéance définies par $d'_k = t$ et $\forall J_j \in S_k, d'_j = d_j$. Chaque tâche $J_j \in S_k \cup \{J_k\}$ doit être

[1] Dans la règle de *Jackson*, les tâches sont ordonnées selon leurs dates d'échéance croissantes et sont ordonnancées par un algorithme de liste qui suit la discipline Earliest Deadline First(*EDF*)

exécutée par la machine de la classe m_j. Les dates d'arrivée des tâches sont calculées comme suit :

1. $r'_k = t - 1$; $\forall J_j \in indep(k), r'_j = r_j$;
2. $\forall J_j \in \Gamma^+(k), r'_j = \max(r_j, t + x_{kj}c_{kj})$.

Question : Existe-il un ordonnancement des tâches de l'ensemble $S_k \cup \{J_k\}$ respectant les dates d'arrivée, les dates d'échéance et les contraintes de ressources ?

Soit l'exemple suivant : considérons le problème $Existence(5, e, r, d)$ pour l'instance définie par l'ensemble des tâches illustrées précédement dans la figure 4.4 avec les temps de communication reportés dans le tableau 4.1. Les dates d'arrivée initiales ainsi que les dates d'échéance initiales et les types des tâches sont donnés dans le tableau 4.2.

$J_i \in \mathcal{J}$	a	b	c	d	e	f	g	h
r_i	0	1	2	0	4	1	0	1
d_i	3	3	7	3	6	6	3	6
m_i	2	1	1	1	2	2	1	2

T . 4.2 – Dates d'arrivée, dates d'échéance et types des tâches pour l'exemple de la figure 4.4

Ainsi, $S_e = \{c, f\} \cup \{b, h\}$ et $Existence(5, e, r, d)$ consiste à trouver un ordonnancement des tâches $\mathcal{J}' = \{b, c, e, f, h\}$ avec des dates d'arrivée et des dates d'échéance précalculées. Ces dates sont résumées dans le tableau 4.3. La réponse à cette question est clairement positive. Nous donnons un exemple de solution réalisable dans la dernière ligne du tableau 4.3.

$J_i \in \mathcal{J}'$	b	c	e	f	h
r_i	1	6	4	5	1
d_i	3	7	5	6	6
t_i	1	6	4	5	1

T . 4.3 – Dates d'arrivée, dates d'échéance et dates de début d'exécution d'une solution réalisable pour le problème de décision $Existence(5, e, r, d)$.

Si un problème $Existence(t, k, r, d)$ retourne une réponse négative alors il n'est pas possible d'exécuter la tâche J_k à l'instant $t - 1$ sans violer une contrainte de précédence ou de ressource du problème initial. Ainsi, la valeur maximale $t^\star \in \{r_k + 1, \ldots, d_k\}$ telle que $Existence(t^\star, k, r, d)$ soit vrai constitue une borne supérieure sur le temps de complétude de la tâche J_k. Dans la suite, ces valeurs $t^\star$ seront calculées pour chaque tâche J_k afin d'améliorer les dates d'échéance des tâches, *i.e.* la date d'échéance d_i de chaque tâche J_i sera remplacée par $t_i^\star$, $d_i^\star := t_i^\star$.

D'un point de vue algorithmique, nous notons par $BS(k, r, d)$ un algorithme qui calcule la valeur $t^\star$ si elle existe. Le problème $Existence(t, k, r, d)$ peut être résolu simplement en $O(n\ log\ n)$ par un algorithme utilisant la régle de *Jackson*. Cependant, comme il a été expliqué dans [37, 62], deux recherches binaires sur l'intervalle $[r_k, d_k - 1]$ basées sur la régle de $Jackson$ sont nécessaires pour calculer la date $t^\star$. Ceci mène à un algorithme dont la complexité en temps est bornée par $O(log(d_k - r_k) \times n\ log\ n)$. Une propriété intéressante de la fonction BS est décrite dans le lemme suivant :

Lemme 14 *Soit d^1 et d^2 deux éléments de $\mathbb{N}^n$ avec $d^1 \leq d^2$ (i.e. $\forall J_i \in \mathcal{J}, d_i^1 \leq d_i^2$). Pour chaque vecteur $r \in \mathbb{N}^n$ tel que $r \leq d^1$ et pour chaque tâche $J_i \in \mathcal{J}$, $BS(i, r, d^1) \leq BS(i, r, d^2)$.*

Preuve Pour chaque valeur entière $t \in [r_i, d_i^1]$, si $Existence(t, i, r, d^1)$ est vrai alors $Existence(t, i, r, d^2)$ est aussi vrai. Ainsi, puisque BS est un algorithme de maximisation on conclut que $BS(i, r, d^1) \leq BS(i, r, d^2)$. □

4.5.2 Modification des dates d'échéance

L'idée dans cette partie est d'améliorer les dates d'échéance de toutes les tâches en calculant d'une façon successive des bornes supérieures sur les temps de complétude à l'aide de l'algorithme 2.

Si un appel à la fonction BS ne retourne pas de solution, alors l'Algorithme 2 s'arrête. Dans la suite, nous supposons qu'il est possible de calculer le vecteur $d^\star$. Pour chaque couple $(i, k) \in \{1, \ldots, n\}^2$, on note par $d_k^\star(i)$ la date d'échéance modifiée de la tâche J_k à la fin de la $i^{ème}$

Algorithme 2: Modification des dates d'échéance des tâches

Données : Un graphe de précédence G avec des temps de communication, dates d'arrivée r, dates d'échéance d et K machines dédiées.

Résultat : Des dates d'arrivée $r^\star$ et des dates d'échéance $d^\star$ modifiées.

début

Calculer $\forall i \in \{1, \ldots, n\}$, $r_i^\star = \max_{J_j \in \Gamma^-(i)}(r_i, r_j^\star + 1 + x_{ji}c_{ji})$ suivant un ordre topologique;

Renuméroter les tâches telles que $r_1^\star \geq r_2^\star \geq \ldots \geq r_n^\star$;

Calculer $\forall i \in \{1, \ldots, n\}$, $d_i^\star = \min_{J_j \in \Gamma^+(i)}(d_i, d_j^\star - 1 - x_{ij}c_{ij})$ suivant un ordre topologique inverse;

pour $i = 1$ *à* n **faire**

$d_i^\star = BS(i, r^\star, d^\star)$;

Calculer $\forall k \in \{1, \ldots, n\}$, $d_k^\star = \min_{J_j \in \Gamma^+(k)}(d_k, d_j^\star - 1 - x_{kj}c_{kj})$ suivant un ordre topologique inverse;

fin

fin

itération de la boucle **pour**.

Nous considérons comme exemple l'exécution de l'Algorithme 2 sur l'instance du problème d'ordonnancement présentée dans la figure 4.4. Le tableau 4.4 présente le vecteur $r^\star$ et une nouvelle numérotation possible des tâches. Les valeurs $d_k^\star(i)$, $(i, k) \in \{1, \ldots, n\}^2$ sont rapportées dans le tableau 4.5.

$J_i \in \mathcal{J}'$	a	b	c	d	e	f	g	h
$r_i^\star$	0	2	6	0	4	5	0	3
i	8	5	1	7	3	2	6	4

T . 4.4 – Dates d'arrivée $r^\star$ améliorées et une nouvelle numérotation des tâches.

$J_i \in \mathcal{J}'$	1	2	3	4	5	6	7	8
$d_i^\star(0)$	7	6	5	5	3	2	3	1
$d_i^\star(1)$	7	6	5	5	3	2	3	1
$d_i^\star(2)$	7	6	5	5	3	2	3	1
$d_i^\star(3)$	7	6	5	5	3	2	3	1
$d_i^\star(4)$	7	6	5	4	3	1	1	1
$d_i^\star(5)$	7	6	5	4	3	1	1	1
$d_i^\star(6)$	7	6	5	4	3	1	1	1
$d_i^\star(7)$	7	6	5	4	3	1	1	1
$d_i^\star(8)$	7	6	5	4	3	1	1	1

T . 4.5 – Dates d'échéance améliorées $d_k^\star(i), (i,k) \in \{1,\ldots,8\}^2$.

Les lemmes suivants caractérisent les vecteurs $d^\star(i) = (d_1^\star(i), d_2^\star(i), \ldots, d_n^\star(i))$, pour toute itération $i \in \{1,\ldots,n\}$:

Lemme 15 *Pour chaque tâche $J_k \in \mathcal{J}$, la fonction $i \to d_k^\star(i)$ est décroissante pour $i \in \{1,\ldots,n\}$. De plus, pour chaque valeur $i \in \{k,\ldots,n\}$, $d_k^\star(i) = d_k^\star(k) = BS(k, r^\star, d^\star(k-1))$.*

Preuve Par définition de la fonction BS, $BS(k, r^\star, d^\star(k-1)) \in \{r_k^\star + 1, \ldots, d_k^\star(k-1)\}$, d'où $BS(k, r^\star, d^\star(k-1)) \leq d_k^\star(k-1)$. Par ailleurs, l'ajustement des valeurs de $d^\star$ en suivant un ordre topologique inverse ne fait que décroitre le vecteur $d^\star$. Ainsi, $d^\star$ ne peut que décroitre et la première partie du lemme est prouvée.

Maintenant, par l'Algorithme 2, pour chaque $k \in \{1,\ldots,n\}$, $d_k^\star(k) = BS(k, r^\star, d^\star(k-1))$. Comme $r_n^\star \leq r_{n-1}^\star \leq \ldots \leq r_k^\star$, la tâche J_k n'admet pas de successeur dans l'ensemble des tâches $J_{k+1}, \ldots, J_n$. Ainsi, la modification de $d_j^\star$, $j \in \{k+1,\ldots,n\}$ n'influence pas la valeur de $d_k^\star$ et $d_k^\star(i) = d_k^\star(k)$ pour toute valeur $i \in \{k,\ldots,n\}$. □

Lemme 16 *Pour chaque arc $(J_k, J_j) \in \mathcal{A}$ et pour chaque $i \in \{1,\ldots,n\}$, $d_k^\star(i) < d_j^\star(i)$.*

Preuve Ce lemme découle de la procédure d'ajustement appelée à chaque étape en suivant un

ordre topologique inverse. □

Le lemme suivant prouve qu'à la fin de l'Algorithme 2, il n'est pas possible d'améliorer d'avantage les valeurs $d^\star_\ell$, pour $J_\ell \in \mathcal{J}$ en utilisant la fonction BS :

Lemme 17 *Soit $r^\star$ et $d^\star$ les vecteurs des dates d'arrivée et des dates d'échéance retournés par l'Algorithme 2. Si un ordonnancement réalisable existe, $\forall k \in \{1, \dots, n\}$, $d^\star_k = BS(k, r^\star, d^\star)$.*

Preuve Supposons, par l'absurde, qu'il existe une tâche $J_k \in \mathcal{J}$ telle que $d^\star_k \neq BS(k, r^\star, d^\star)$ où $r^\star$ et $d^\star$ sont les veteurs des dates d'arrivée et des dates d'échéance calculés par l'algorithme 2. Par le lemme 15, on a $d^\star_k = d^\star_k(k) = BS(k, r^\star, d^\star(k-1))$ et $d^\star(k-1) \geq d^\star$. Ainsi, par le lemme 14, $d^\star_k = BS(k, r^\star, d^\star(k-1)) > BS(k, r^\star, d^\star)$.

Par conséquent, $Existence(d^\star_k, k, r^\star, d^\star)$ est fausse. Soit r' et d' les dates d'arrivée et les dates d'échéance définies pour cet appel de la fonction $Existence$. $\forall J_\ell \in S_k \cup \{J_k\}$, soit t_ℓ la date de début d'exécution de la tâche J_ℓ définie par un ordonnancement de liste régi par la régle de $Jackson$. Soit t le premier instant tel qu'une tâche $J_j \in S_k$ rate sa date d'échéance d'_j, *i.e.* $t = d'_j$.

1. Si $t < d'_k$, alors il n'existe pas d'ordonnancement réalisable pour l'ensemble $A = \{J_\ell \in S_k \cup \{J_k\}, d'_\ell \leq t\}$ avec les dates d'arrivée r' et les dates d'échéance d'. Il est clair que $J_k \notin A$. De plus chaque tâche $J_\ell \in A$ vérifie $d'_\ell < d'_k$, ainsi, par le lemme 16 $J_\ell \notin \Gamma^+(k)$.
   La conséquence est que $A \subseteq indep(k)$ et donc $\forall J_\ell \in A, r'_\ell = r^\star_\ell$. Nous pouvons déduire qu'il n'existe pas d'ordonnancement réalisable pour le problème d'ordonnancement initial.

2. Sinon, $t \geq d'_k$. Soit Δ la plus petite valeur dans $\{-1, 0, \dots, d'_j - 1\}$ telle que la machine de la classe m_j soit occupée durant chaque slot de temps dans $[\Delta + 1, d'_j]$ par une tâche J_ℓ avec $d'_\ell \leq d'_j$. Soit $B = \{J_j\} \cup \{J_\ell \in S_k \cup \{J_k\} | m_\ell = m_j$ et $\Delta < t_\ell < d'_j\}$. Il est clair que $|B| = d'_j - \Delta$. De plus, $\forall J_\ell \in B, r'_\ell \geq \Delta + 1$ et $d'_\ell \leq d'_j$.
   Nous prouvons, dans la suite, que $\forall J_\ell \in B, r^\star_\ell \geq \Delta + 1$ et $d^\star_\ell \leq d'_j$.
   - Chaque tâche $J_\ell \in B \backslash \{J_k\}$ vérifie $r'_\ell = r^\star_\ell$ et $d'_\ell = d^\star_\ell$. Ainsi, par définition de B, $r^\star_\ell \geq \Delta + 1$ et $d^\star_\ell \leq d'_j$.
   - Maintenant, si $J_k \in B$, $d^\star_k = d'_k \leq d'_j$. Supposons, par l'absurde, que $r^\star_k < \Delta + 1$. Alors, par l'Algorithme 2, $r^\star_n \leq r^\star_{n-1} \leq \dots \leq r^\star_k < \Delta + 1$ et donc, $\forall J_\ell \in B - \{J_k\}, \ell < k$. Par le

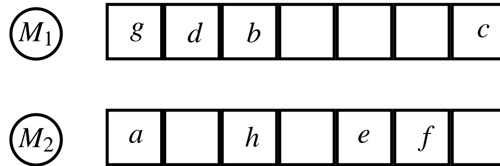
lemme 15, $d^\star_\ell(k-1) = d^\star_\ell = d'_\ell$. Ainsi, $Existence(d^\star_k, k, r^\star, d^\star(k-1))$ doit exécuter toutes les tâches de l'ensemble B durant l'intervalle de temps $[\Delta + 1, d'_j]$, ce qui est impossible. D'où, $Existence(d^\star_k, k, r^\star, d^\star(k-1))$ est fausse, et ceci constitue une contradiction puisque $d^\star_k(k) = BS(k, r^\star, d^\star(k-1))$. Nous concluons que $r^\star_k \geq \Delta + 1$.

Donc, toutes les tâches dans B avec les dates d'arrivée $r^\star$ et les dates d'échéance $d^\star$ doivent être exécutées sur la machine m_j durant l'intervalle de temps $[\Delta + 1, d'_j]$, ce qui est impossible en considérant $|B|$. Ainsi, il n'existe pas d'ordonnancement réalisable pour le problème initial. □

4.5.3 Optimalité de l'extension de l'algorithme *LPP*

L'extension de l'algorithme *LPP* consiste à modifier les dates d'échéance des tâches en utilisant l'Algorithme 2 puis à construire un ordonnancement basé sur une liste de priorité L telle que pour chaque couple de tâches $(J_i, J_j) \in \mathcal{J}^2$, $L(J_i) < L(J_j) \Rightarrow d^\star_i \leq d^\star_j$.

Nous considérons, comme exemple, l'ordonnancement obtenu pour l'instance illustrée par la figure 4.4 avec les dates d'échéance modifiées listées dans le tableau 4.5. Cet ordonnancement est présenté dans la figure 4.6. Nous montrons dans le théorème suivant que dans le cas de machines dédiées et de temps de communication monotones, il existe un ordonnancement réalisable si et seulement si l'ordonnancement retourné par l'extension de l'algorithme *LPP* est réalisable.



F . 4.6 – Extension du *LPP* pour l'exemple de la figure 4.4 avec les dates d'échéance modifiées reportées dans le tableau 4.5.

Théorème 9 *La version étendue de l'algorithme LPP résout d'une façon polynomiale le problème de la détermination d'un ordonnancement réalisable pour un ensemble de tâches unitaires et typées avec des dates d'arrivée et d'échéance arbitraires, des temps de communication monotones,*

des contraintes de précédence définies par un graphe d'ordre d'intervalles et un nombre limité de machines typées avec une seule machine par type.

Preuve Supposons, par contradiction, que l'extension de l'algorithme *LPP* retourne un ordonnancement $\sigma = \{t_1, \ldots, t_n\}$ qui ne soit pas réalisable. Soit J_i la première tâche qui rate sa date d'échéance modifiée dans l'ordonnancement σ, *i.e.* $t_i \geq d_i^\star$. Nous nous proposons de montrer dans la suite qu'il doit exister une tâche J_k antérieure à J_i dans σ qui rate aussi sa date d'échéance modifiée $d_k^\star$.

Une tâche $J_j \in \mathcal{J}$ est dite saturée si $m_j = m_i$ et $d_j^\star \leq d_i^\star$. Soit $[\Delta, \Delta+1]$, pour $\Delta < d_i^\star$, le dernier slot de temps qui ne soit pas occupé par une tâche saturée sur la machine de type m_i. Nous considérons alors les ensembles de tâches $\Sigma = \{J_\ell \textit{ saturée} : \Delta < t_\ell < d_i^\star\} \cup \{J_i\}$ et $\Sigma' = \{J_\ell \in \Sigma : r_\ell^\star \leq \Delta\}$.

Nous prouvons, tout d'abord et par contradiction, que $\Sigma' \neq \emptyset$ et $\Delta \geq 0$. En effet, si $\Sigma' = \emptyset$ alors $\forall J_\ell \in \Sigma$, $r_\ell^\star > \Delta$. On a $|\Sigma| = d_i^\star - \Delta > 0$ et il existe exactement $d_i^\star - (\Delta + 1)$ slots de temps disponibles entre $\Delta + 1$ et $d_i^\star$ sur la machine m_i. Ainsi, le problème d'ordonnancement n'est pas réalisable et, par conséquence, $\Sigma' \neq \emptyset$ et, donc, $\Delta \geq 0$.

Étant donné que l'ensemble $\Sigma' \neq \emptyset$ est fini, il existe une tâche $J_j \in \Sigma'$ telle que $|\Gamma^-(j)|$ est minimal dans le sens de l'inclusion. Du fait que J_j n'est pas ordonnancée à la date Δ ou avant, il existe nécessairement une tâche critique $J_k \in \Gamma^-(j)$ qui empêche J_j d'être exécutée avant Δ. Notons que, $\forall J_\ell \in \Sigma'$, on a $J_k \in \Gamma^-(\ell)$. De plus, à la suite du calcul de $r^\star$, on a $r_k^\star < r_j^\star$. Comme $J_j \in \Sigma'$, alors $r_j^\star \leq \Delta$. Ainsi, $r_k^\star < \Delta$.

La tâche J_k est soit de type m_i ou d'un type différent de m_i :

Cas 1 Supposons que $m_k = m_i$. Dans ce cas, il n'existe pas de délai de communication entre J_k et les tâches dans Σ'. En outre, la tâche J_k contraint la tâche J_j à être exécutée aprés la date Δ, ainsi $t_k \geq \Delta$. Nous prouvons que $J_k \notin \Sigma$: puisque $J_k \in \Gamma^-(j)$, alors on a $\Gamma^-(k) \subset \Gamma^-(j)$ et, ainsi, $J_k \notin \Sigma'$. Etant donné que $r_k^\star < \Delta$, nous avons $J_k \notin \Sigma$.

Maintenant, puisque $d_k^\star < d_j^\star \leq d_i^\star$, alors J_k est une tâche saturée. Aussi le fait que $J_k \notin \Sigma$ implique que $t_k \leq \Delta$ et donc $t_k = \Delta$, or ceci est impossible par définition de Δ.

Cas 2 Supposons maintenant que $m_k \neq m_i$. Dans ce cas, un delai de communication addition-

nel entre les tâches J_k et J_j sera pris en compte, *i.e.* $t_k + 1 + c_{kj} > \Delta$.

Maintenant, nous nous proposons de prouver que $\Sigma \subset S_k$. En effet, par la structure de graphe d'ordre d'intervalles, on a :

$$\mathcal{J} = \Gamma^-(k) \cup \{J_k\} \cup \Gamma^+(k) \cup indep(k) = S_k \cup \{J_k\} \cup \Gamma^-(k)$$

Nous montrons que $\Gamma^-(k) \cap \Sigma = \emptyset$. Pour chaque tâche $J_\ell \in \Gamma^-(k)$, l'inégalité $r^\star_\ell < r^\star_k$ est valide. Étant donné que $r^\star_k < \Delta$, nous avons que $r^\star_\ell < \Delta$. Ainsi, si $J_\ell \in \Sigma$, alors $J_\ell \in \Sigma'$. Cependant, si $J_\ell \in \Sigma'$, alors $J_k \in \Gamma^-(\ell)$ et, donc, il existe un circuit dans le graphe G, ce qui est impossible. Ainsi, $\Sigma \subset S_k$.

Maintenant, en utilisant le lemme 17, on a $d^\star_k = BS(k, r^\star, d^\star)$ et $Existence(d^\star_k, k, r^\star, d^\star)$ est vrai. Étant donné que $\Sigma \subset S_k$, un algorithme de liste suivant la régle de *Jackson* calcule des dates de début d'exécution réalisables pour les $d^\star_i - \Delta$ tâches de Σ avant la date $d^\star_i$. Ainsi, il existe au moins une tâche $J_{j'} \in \Sigma'$ exécutée à la date $t' \leq \Delta$ par ce dernier algorithme. De ce fait, on a $d^\star_k + c_{kj'} \leq t' \leq \Delta$. Maintenant, le fait que $\Gamma^-(j) \subseteq \Gamma^-(j')$ et que les temps de communication sont monotones impliquent que $c_{kj} \leq c_{kj'}$, d'où $d^\star_k + c_{kj} \leq \Delta$.

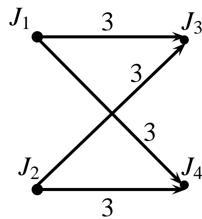
Finalement, puisque $t_k + 1 + c_{kj} > \Delta$ et $\Delta \geq d^\star_k + c_{kj}$, alors on a $t_k + 1 > d^\star_k$ et ainsi la tâche J_k rate son échéance, d'où une contradiction.

□

4.6 N ´

Dans cette section, nous nous proposons d'étudier l'influence du nombre de machines de chaque classe $\{1, \ldots, K\}$ sur la résolution du problème d'ordonnancement. En effet, tous les résultats précédents ont été établis sous l'hypothèse d'une seule machine par classe (machines dédiées), *i.e.* $\forall p \in \{1, \ldots, K\}, \tau_p = 1$ où τ_p représente le nombre de machines dans la classe p.

La figure 4.7 montre qu'aucun algorithme de liste ne peut retourner une solution optimale pour m machines identiques. Ainsi, l'extension de l'algorithme *LPP* à plusieurs machines identiques par



J_1				J_3
J_2				J_4

Un ordonnancement de liste non optimal

J_1	J_2	J_3	J_4

Un ordonnancement optimal

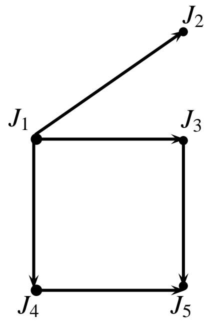
F . 4.7 – Les algorithmes de liste ne sont pas optimaux pour 2 machines identiques : exemple d'ordonnancement de tâches unitaires sur 2 machines identiques avec des temps de communication grands.

classe n'est pas optimale même en absence des dates d'arrivée et d'échéance.

La question reste, néanmoins, ouverte dans le cas où les délais de communication sont unitaires, *i.e.* $c_{ij} = 1, \forall (J_i, J_j) \in \mathcal{A}$. Dans ce cas, Jacques Verriet [80] a présenté un algorithme polynomial pour le problème de faisabilité avec des machines identiques. Cependant, la modification des dates d'échéance vue dans ce chapitre ne conduit pas à une solution réalisable comme le montre la figure 4.8. En effet, l'algorithme de modification des dates d'échéance n'améliore pas les dates d'échéance fixées initialement, et ainsi l'algorithme *LPP* pourrait retourner un ordonnancement non réalisable.

4.7 C

Nous avons montré dans ce chapitre que l'algorithme *LPP* introduit par Leung et al. [62] pouvait bien s'étendre pour résoudre le problème d'ordonnancement sur des graphes de précédence ayant la structure d'ordre d'intervalles avec des délais de communication monotones, des tâches unitaires caractérisées par des dates d'arrivée et des dates d'échéance et exécutées sur des machines dédiées (une seule machine par type).



$$c_{ij} = 1, \forall (J_i, J_j) \in \mathcal{A}$$

$$d_1 = 1$$

$$d_2 = d_3 = d_4 = 1$$

$$d_5 = 4$$

$$\forall J_i \in \mathcal{J}, d_i^\star = d_i$$

J_1	J_2	J_3		J_5
		J_4		

Un ordonnancement LPP non réalisable

J_1	J_4	J_3	J_5
		J_2	

Un ordonnancement réalisable

F . 4.8 – L'algorithme *LPP* risque de retourner un ordonnancement non réalisable si les temps de communication sont unitaires.

Dans le cas où l'hypothèse sur la monotonie des temps de communication est relachée, nous avons exhibé une réduction polynomiale à partir du problème *SCL* et nous avons ainsi montré que le problème devient *NP*-complet au sens fort.

Dans le cas monotone, la construction d'un ordonnancement réalisable repose sur une procédure de modification des dates d'échéance à travers un calcul successif de bornes supérieures sur le temps de complétude de chaque tâche. Ensuite une liste de priorité basée sur les nouvelle dates d'échéance est constituée et conduit à la construction d'un ordonnancement réalisable à l'aide d'un algorithme de liste classique. Nous avons finalement montré que l'extension obtenue de l'algorithme *LPP* résout d'une façon polynomiale le problème avec des temps de communication monotones.

Enfin, nous nous sommes intéressés au nombre de machines par classe et l'éventuelle influence sur la résolution du problème par un algorithme de liste. Ainsi, nous avons construit un contre exemple avec deux machines du même type permettant de conclure qu'il n'existe pas d'algorithme de liste qui retourne une solution optimale du problème. La question reste donc ouverte

quant à la résolution éfficace du problème avec un nombre quelconque de machines identiques sur des ordres d'intervalles en présence de délais de communication monotones et des tâches à durées d'exécution unitaires.

Nous avons aussi montré à travers un exemple simple que la procédure de modification des dates d'échéance n'améliore pas les dates d'échéance des tâches dans le cas où les temps de communication sont unitaires. Ceci nous conduit vers une perspective intéressante qui consiste en l'extension de ce travail au cas du système de tâches typées avec des délais de communication unitaires. Nous sommes, en effet, convaincus qu'une légère modification dans la procédure de calcul des dates d'échéance peut donner lieu à un algorithme polynomial.

C 5

A

5.1 I

Les réseaux de partage de fichiers pair-à-pair sont devenus un aspect très populaire de l'utilisation quotidienne de l'internet. Le succès de tels systèmes est basé sur l'exploitation d'une nouvelle ressource : le stockage distribué. L'utilisation massive de cette nouvelle ressource est dûe au fait que des capacités de stockage plus importantes sont actuellement disponibles avec un coût très faible et avec des temps d'accès très rapides. Ainsi les utilisateurs réagissent entre eux en installant un stockage local (*cache*), en répliquant les contenus les plus populaires et en les mettant à disposition des utilisateurs du voisinage. Ceci conduit à une diminution dans la consommation de la bande passante nécessaire pour accéder aux données à partir des serveurs originaux qui les hébergent.

Le problème du *placement de données* ($\mathcal{DP}$) offre un modèle abstrait approprié pour modéliser cette situation : un ensemble de clients (machines ou utilisateurs) reliés par une topologie est considéré où chaque client dispose d'une capacité de stockage donnée. Pour un ensemble de données (objets) disponibles et étant donnée la préférence de chaque client pour chaque sous ensemble d'objets, le but est de proposer un schéma de réplication d'objets, autrement dit un placement d'objets (et de leurs copies) sur les machines qui minimise le coût total d'accès pour

tous les clients et tous les objets.

Dans un modèle général du problème de placement de données, ce coût total d'accès est une fonction des coûts d'installation des objets et de leurs copies sur les noeuds utilisateurs et des coûts de transfert des données entre les différents utilisateurs du réseau (*coût de communication*). Or dans un réseau étendu, ce coût de communication est fonction de divers paramètres incluant les délais et les capacités des liens de connexion, les tailles des mémoires tampons et des entêtes de communication et les profils d'utilisation du réseau par les différents noeuds. Cependant, il faut noter que tous ces paramètres du réseau interagissent de façon complexe ce qui rend non réalisable leur introduction dans le modèle [17].

5.2 D´ `

On considère un réseau constitué d'un ensemble $\mathcal{M}$ de M utilisateurs, reliés entre eux selon une certaine topologie $\mathcal{N}$, ainsi qu'un ensemble $\mathcal{O}$ de N objets de tailles unitaires qui peuvent être répliqués.

Chaque utilisateur $i \in \mathcal{M}$ demande à accéder à un ensemble $R_i \subseteq \mathcal{O}$ d'objets selon une certaine fréquence. On note w_{io} la fréquence d'accès de l'utilisateur i à un objet $o \in R_i$. Le coût d'accès d'un utilisateur i à un objet $o \in R_i$ est nul dans le cas où celui-ci a été répliqué dans le cache de i. Sinon, si l'objet o est stocké dans le cache d'un utilisateur voisin j (selon la topologie $\mathcal{N}$), alors le coût d'accès est défini comme étant égal à $d_{j,i}w_{io}$, avec $d_{j,i}$ la distance entre l'utilisateur j et l'utilisateur i. Si, par contre, l'objet o n'est pas stocké localement, ni dans le cache d'un utilisateur voisin, alors l'utilisateur i doit faire appel à un serveur distant de capacité non bornée avec un coût d'accès $d \cdot w_{io}$, où d est une constante qui peut être égale à $+\infty$.

Il existe plusieurs variantes du modèle en fonction des distances entre les utilisateurs. Nous pouvons ainsi citer le modèle d'espace métrique où les distances entre les noeuds sont non néga-

tives, symétriques et respectent l'inégalité triangulaire, *i.e.* $d_{i,j} \leqslant d_{i,k} + d_{k,j}$. Nous pouvons citer, également, l'espace ultramétrique où, en plus des propriétés de la non négativité et de la symétrie, les distances respectent l'inégalité ultratriangulaire, *i.e.* $d_{i,j} \leqslant max\{d_{i,k}, d_{k,j}\}$. Dans notre étude, nous considérons le modèle général où les distances entre les noeuds utilisateurs sont quelconques. Par ailleurs, chaque utilisateur $i \in \mathcal{M}$ peut stocker au maximum C_i objets dans son cache. L'objectif est de déterminer un placement optimal minimisant le coût total d'accès des utilisateurs aux objets.

5.3 T

Le problème du placement de données a été largement étudié dans la littérature. Dans [17], Baev et al. ont considéré des distances métriques, *i.e.* vérifiant l'inégalité triangulaire, entre les noeuds utilisateurs et ont montré que le problème est MAX SNP-difficile, c'est à dire qu'il n'existe de schéma d'approximation polynomial (*PTAS*). Par ailleurs, ils ont proposé un algorithme 20.5-approché dans le cas où les objets sont de tailles uniformes, *i.e.* la taille est identique pour tous les objets. Ce résultat a été obtenu en se basant sur l'arrondi d'une solution optimale d'un programme linéaire approprié. Dans le cas d'objets de tailles non uniformes, les auteurs ont prouvé que le problème de décision est NP-complet et ont donné un algorithme polynomial 20.5-approché qui fournit une solution dans laquelle la capacité de stockage de chaque utilisateur dépasse sa capacité dans la solution optimale par au plus la taille ℓ_{max} du plus grand objet. Ces résultats ont été améliorés dans [16] où un algorithme 10-approché a été proposé dans le cas d'objets de tailles uniformes et 10-approché avec dépassement de capacité de stockage dans le cas d'objets de tailles non uniformes.

Plus récemment, un algorithme optimal a été proposé dans [13] dans le cas où tous les objets sont de tailles uniformes et le nombre de clients est borné par une constante. Les auteurs de [13] ont également proposé un algorithme optimal dans le cas où les tailles des objets sont non-uniformes à condition d'autoriser un dépassement de la capacité des clients par au plus $\epsilon\ell_{max}$

où $\epsilon > 0$ est un réel arbitrairement petit.

Nous décrivons, dans ce chapitre, un algorithme optimal, basé sur la programmation dynamique, dans le cas où la topologie du réseau est une ligne et sans violation de la capacité des clients. Nous étendons ce résultat lorsque la topologie est un anneau et une étoile généralisée. Ces résultats restent valides dans le cas où le nombre de clients est une entrée du problème (*i.e.* non-borné par une constante). Finalement, nous prouvons qu'il n'est pas possible d'étendre cet algorithme dans le cas d'une topologie quelconque où le nombre des noeuds de degré supérieur ou égal à 3 est constant.

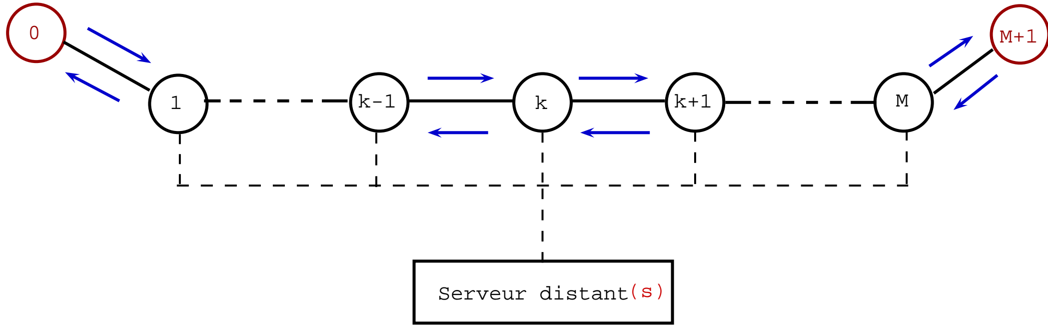
5.4 A -

Dans cette partie, nous présentons un algorithme de programmation dynamique optimal dans le cas d'un réseau de connexion dont la topologie est une chaîne. Nous étendons, par la suite, cet algorithme dans le cas d'une topologie en anneau puis en étoile généralisée.

5.4.1 Placement de données sur une chaîne

Notons par P_i un placement d'objets sur l'utilisateur i, *i.e.* P_i est un sous-ensemble d'objets qu'on choisit de répliquer localement sur i. Pour simplifier la présentation, nous introduisons deux utilisateurs fictifs 0 et $M + 1$ avec des placements vides sur chacun d'eux, i.e. $P_0 = P_{M+1} = \emptyset$, et nous supposons que les utilisateurs sont répartis le long d'une chaîne, avec l'utilisateur i ($1 \leqslant i \leqslant M$) voisin des utilisateurs $i - 1$ et $i + 1$ (voir la figure 5.1).

Soit $\mathcal{P}_i$ l'ensemble de tous les placements d'objets réalisables sur l'utilisateur i. Pour un utilisateur donné i, nous avons $|\mathcal{P}_i| \leqslant N^{C_i} \leqslant N^C$, avec $C = \max_{i \in \mathcal{M}}(C_i)$. Pour $1 \leqslant k \leqslant M$, nous notons $C_k(P_{k-1}, P_k, P_{k+1})$ le coût d'accès induit par le placement P_k sur l'utilisateur k, quand ses



F . 5.1 – Problème de placement de données sur une chaîne.

utilisateurs voisins $k-1$ et $k+1$ ont reçu des placements P_{k-1} et P_{k+1} respectivement. La fonction C_k est définie comme suit :

$$C_k(P_{k-1}, P_k, P_{k+1}) = \sum_{\substack{o \in R_k \cap P_{k-1} \cap P_{k+1} \\ o \notin P_k}} \min\{d_{k-1,k}, d_{k+1,k}\}.w_{ko} + \sum_{\substack{o \in R_k \cap P_{k-1} \\ o \notin P_k \cup P_{k+1}}} d_{k-1,k}.w_{ko} + \sum_{\substack{o \in R_k \cap P_{k+1} \\ o \notin P_k \cup P_{k-1}}} d_{k+1,k}.w_{ko} + \sum_{\substack{o \in R_k \\ o \notin P_k \cup P_{k-1} \cup P_{k+1}}} d.w_{ko}$$

L'interprétation de la fonction C_k pour des placements donnés sur les utilisateurs $k-1$, k et $k+1$ consiste à classer les objets requises par l'utilisateur k, *i.e.* R_k en 4 classes selon l'endroit où ces objets sont répliqués. Nous considérons, ainsi, un premier sous ensemble d'objets qui sont répliqués sur les utilisateurs $k-1$ et $k+1$ mais pas sur k et dans ce cas, ce dernier accèdent à chacun de ces objets en sollicitant soit $k-1$, soit $k+1$ selon la proximité de ces utilisateurs voisins. Le deuxième et le troisième sous ensembles correspondent au cas où l'objet est répliqué uniquement sur le voisin $k-1$ ou le voisin $k+1$, et le dernier sous ensemble d'objets de R_k correspond aux objets qui ne se répliqués sur aucun des trois utilisateurs. Dans ce cas, l'utilisateur k fait appel au serveur distant pour récupérer les objets requis. Le cas où les objets sont répliqués localement sur l'utilisateur k n'est pas pris en compte car le coût d'un accès local est nul.

Pour $k \in \{1, \ldots, M\}$, étant donné le placement P_{k-1} (resp. P_k) sur l'utilisateur $k-1$ (resp. k), nous notons $C_k^*(P_{k-1}, P_k)$ le coût d'une solution optimale minimisant la somme des coûts d'ac-

cès des utilisateurs $k, k+1, \ldots$ jusqu'à M. Le coût total optimal recherché est alors donné par $\min_{P_1 \in \mathcal{P}_1} C_1^*(P_0, P_1)$, et nous avons les relations de récurrences suivantes :

$$C_k^*(P_{k-1}, P_k) = \begin{cases} min_{P_{k+1} \in \mathcal{P}_{k+1}} \{C_k(P_{k-1}, P_k, P_{k+1}) + C_{k+1}^*(P_k, P_{k+1})\} & \forall k \in \{1, \ldots, M-1\}, \\ \\ C_k(P_{k-1}, P_k, P_{k+1}) & \text{si } k = M. \end{cases}$$

Afin de calculer $C_k^*(P_{k-1}, P_k)$ pour des placements fixés sur les utilisateurs $k-1$ et k, nous supposons que toutes les valeurs $C_{k+1}^*(P_k, P_{k+1})$ sont déterminées pour tous les placements possibles p_{k+1} sur l'utilisateur $k+1$. Il suffit donc de rajouter le coût d'accès induit par le placement P_k sur k et de minimiser la somme résultante par rapport aux placements $P_{k+1} \in \mathcal{P}_{k+1}$. Le cas initial correspond à $k = M$ et donc $C_M^*(P_{M-1}, P_M) = C_k(P_{M-1}, P_M, P_{M+1})$. Rappelons, aussi, que $P_{M+1} = \emptyset$. À partir de ces relations il est possible d'en déduire un algorithme de programmation dynamique, donné par *Algorithme* 3.

La complexité de *Algorithme* 3 est déduite à partir du deuxième ensemble de boucles imbriquées et qui consiste à effectuer $M-1$ étapes de calcul. Dans chacune de ces étapes, il s'agit pour chaque combinaison possible de placements d'objets sur deux noeuds voisins $k-1$ et k, *i.e.* $O(N^{2C})$ combinaisons, de minimiser $C_k^*(P_{k-1}, P_k)$ par rapport à tous les placements possibles sur le noeud $k+1$, *i.e.* $O(N^C)$ possibilités de placement. Ainsi, *Algorithme* 3 retourne une solution optimale en temps $O(MN^{3C})$.

5.4.2 Placement de données sur un anneau

De manière similaire, il est possible d'obtenir un algorithme de programmation dynamique pour une topologie en anneau. On note $OPT(P_M, P_1)$ le coût d'une solution optimale sous réserve d'avoir choisi le placement P_M (resp. P_1) pour l'utilisateur M (resp. 1). L'idée consiste à construire une instance du problème de placement de données sur une chaîne en ajoutant deux noeuds fictifs $1'$ et M', respectivement voisins des utilisateurs M et 1 et en éliminant l'arc $(M, 1)$.

Algorithme 3: Le programme dynamique pour le problème de placement de données sur une chaîne

Données : Une instance du problème de placement de données sur un réseau d'utilisateurs reliés suivant une chaîne.

Résultat : Un placement de données optimal.

début

  $\mathcal{P}_0 = \{P_0\}, \mathcal{P}_{M+1} = \{P_{M+1}\}$ avec $P_0 = P_{M+1} = \emptyset$;

  pour chaque $P_{M-1} \in \mathcal{P}_{M-1}$ **faire**

    pour chaque $P_M \in \mathcal{P}_M$ **faire**

      $C^*_M(P_{M-1}, P_M) = C_M(P_{M-1}, P_M, P_{M+1})$

    fin

  fin

  pour $k = M - 1$ *à* 1 *par pas de* -1 **faire**

    pour chaque $P_{k-1} \in \mathcal{P}_{k-1}$ **faire**

      pour chaque $P_k \in \mathcal{P}_k$ **faire**

        $C^*_k(P_{k-1}, P_k) = min_{P_{k+1} \in \mathcal{P}_{k+1}}\{C_k(P_{k-1}, P_k, P_{k+1}) + C^*_{k+1}(P_k, P_{k+1})\}$;

      fin

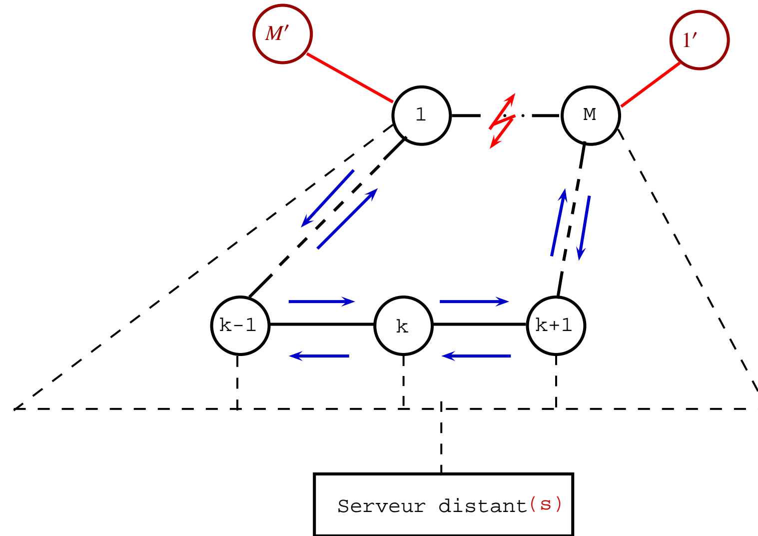
    fin

  fin

  Retourner $\min_{P_1 \in \mathcal{P}_1} C^*_1(P_0, P_1)$;

fin

Le contenu de l'utilisateur 1 (*resp.* M) est répliqué sur le noeud fictif $1'$ (*resp.* M'). Cette nouvelle instance est illustrée par la figure 5.2.



F . 5.2 – Problème de placement de données sur une topologie d'anneau.

Étant donné les placements sur les noeuds utilisateurs 1 et M, il est possible de trouver une façon optimale de placer les objets sur les utilisateurs allant du noeud 2 jusqu'au noeud $M-1$ et de calculer ainsi son coût $OPT(P_M, P_1)$. En effet, cette solution est donnée par l'application du programme dynamique décrit dans la section précédente sur la nouvelle instance construite pour le problème sur une chaîne. La solution optimale sur l'anneau est donnée simplement par $\min_{P_M \in \mathcal{P}_M, P_1 \in \mathcal{P}_1} OPT(P_M, P_1)$.

Nous décrivons ci-dessous un algorithme polynomial, noté *Algorithme* 4, et qui pour chaque instance du problème de placement de données sur un réseau en anneau, retourne une solution optimale en temps $\mathcal{O}(MN^{5C})$.

Algorithme 4: Le programme dynamique pour le problème de placement de données sur un anneau

Données : Une instance du problème de placement de données sur un réseau d'utilisateurs reliés suivant une topologie d'anneau.

Résultat : Un placement de données optimal.

début

  pour chaque $P_M \in \mathcal{P}_M$ **faire**

    pour chaque $P_1 \in \mathcal{P}_1$ **faire**

      pour chaque $P_{M-1} \in \mathcal{P}_{M-1}$ **faire**

        $C^*_M(P_{M-1}, P_M) := C_M(P_{M-1}, P_M, P_1)$;

      fin

      pour $k = M - 1$ *à* 3 *par pas de* -1 **faire**

        pour chaque $P_{k-1} \in \mathcal{P}_{k-1}$ **faire**

          pour chaque $P_k \in \mathcal{P}_k$ **faire**

            $C^*_k(P_{k-1}, P_k) = min_{P_{k+1} \in \mathcal{P}_{k+1}} \{C_k(P_{k-1}, P_k, P_{k+1}) + C^*_{k+1}(P_k, P_{k+1})\}$;

          fin

        fin

      fin

      pour chaque $P_2 \in \mathcal{P}_2$ **faire**

        $C^*_2(P_1, P_2) = min_{P_3 \in \mathcal{P}_3} \{C_2(P_1, P_2, P_3) + C^*_3(P_2, P_3)\}$;

      fin

      $OPT(P_M, P_1) = \min_{P_2 \in \mathcal{P}_2} \{C_1(P_M, P_1, P_2) + C^*_2(P_1, P_2)\}$;

    fin

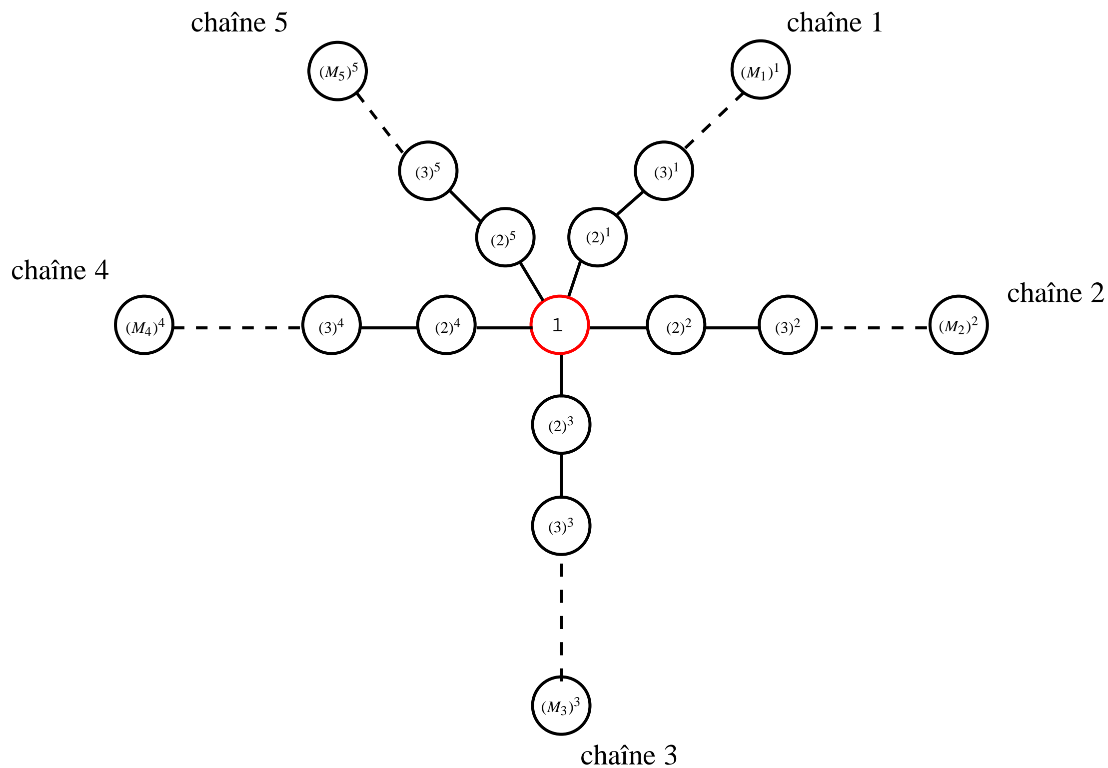
  fin

  Retourner $\min_{P_M \in \mathcal{P}_M, P_1 \in \mathcal{P}_1} OPT(P_M, P_1)$;

fin

5.4.3 Placement de données sur un graphe en étoile généralisée

Dans cette partie, nous nous proposons de résoudre le problème de placement d'objets lorsque la topologie du réseau $\mathcal{N}$ considéré est en étoile généralisée. Dans une telle topologie, les noeuds utilisateurs sont répartis sur un nombre fixe de chaînes qui se coupent en un unique noeud central. Le nombre total de chaînes constituant le réseau est définit par le degré du noeud central. Ainsi, pour identifier ces chaînes, il serait utile de les numéroter en définissant une chaîne de départ et un sens de parcours. Une idée intuitive est de considérer le cercle fictif formé par le voisinage immédiat du noeud central, et de parcourir ce cerle dans le sens des aiguilles d'une montre (figure 5.3).



F . 5.3 – Problème de placement de données sur une topologie en étoile généralisée : exemple d'étoile avec 5 chaînes.

Nous supposons que, sur chaque chaîne, les noeuds sont numérotés en partant du noeud central dont le degré est noté par Δ. Ainsi, le graphe sera constitué de Δ chaînes numérotées de

1 à Δ, et chaque noeud du réseau est identifié par le numéro de la chaîne à laquelle il appartient et son ordre au sein de cette chaîne en partant du noeud central. On note par $(i)^j$ $j \in \{1 \ldots \Delta\}$ le $i^{ème}$ noeud de la chaîne numéro j. On note, aussi, par M_j la taille de la $j^{ème}$ chaîne. L'objectif est de placer les objets sur les noeuds du réseau de façon à minimiser le coût total d'accès des utilisateurs aux objets.

En utilisant le programme dynamique, définie précédement pour calculer un placement optimal sur une chaîne (*Algorithme* 3), l'idée consiste à minimiser la somme des coûts optimaux sur chaque chaîne j, $j \in \{1 \ldots \Delta\}$. Cette minimisation est effectuée par rapport aux différents placements réalisables $P_1 \in \mathcal{P}_1$ sur le noeud central. On définit, ainsi, la fonction $CoûtChaîne(j, M_j, P_{(1)^j})$ qui, étant donnée un placement $P_{(1)^j}$ sur le premier noeud de la chaîne j (qui correspond au noeud central du réseau), retourne le coût total d'accès optimal de placement d'objets sur la chaîne j de taille M_j (*Algorithme* 5). On définit, également, la fonction $TailleChaîne(j)$ qui, pour une chaîne j donnée, retourne la taille correspondante. A l'aide de ces fonctions, nous présentons l'algorithme de minimisation de coût d'accès sur une topologie en étoile généralisée de complexité $O(M \cdot N^{3C})$ (*Algorithme* 6).

5.5 P

Nous nous intéressons dans cette partie au problème de placement de données dans un contexte plus général dans lequel nous considérons un ensemble d'utilisateurs reliés suivant une topologie quelconque et telle que le nombre d'utilisateurs avec un degré supérieur ou égal à 3 est constant.

Une idée naturelle pour résoudre ce problème consiste à décomposer le réseau en un ensemble de sous graphes connexes ne contenant que des chaînes est des anneaux et d'appliquer le

Algorithme 5: CoûtChaîne($h, M_h, P_{(1)^h}$) "Calcul d'un placement optimal sur une chaîne dans un réseau en étoile généralisée"

Données : $h, M_h, P_{(1)^h}$.

Résultat : Coût total d'un placement de données optimal sur la chaîne M_h.

début

$\mathcal{P}_{(0)^h} = \{P_{(0)^h}\}, \mathcal{P}_{(M_h+1)^c} = \{P_{(M_h+1)^h}\}$ avec $P_{(0)^h} = P_{(M_h+1)^h} = \emptyset$;

pour chaque $P_{(M_h-1)^h} \in \mathcal{P}_{(M_h-1)^h}$ **faire**

pour chaque $P_{(M_h)^h} \in \mathcal{P}_{(M_h)^h}$ **faire**

$C^*_{(M_h)^h}(P_{(M_h-1)^h}, P_{(M_h)^h}) = C_{(M_h)^h}(P_{(M_h-1)^h}, P_{(M_h)^h}, P_{(M_h+1)^h})$

fin

fin

pour $k = M_h - 1$ *à* 2 *par pas de* -1 **faire**

pour chaque $P_{(k-1)^h} \in \mathcal{P}_{(k-1)^h}$ **faire**

pour chaque $P_{(k)^h} \in \mathcal{P}_{(k)^h}$ **faire**

$C^*_{(k)^h}(P_{(k-1)^h}, P_{(k)^h}) =$

$min_{P_{(k+1)^h} \in \mathcal{P}_{(k+1)^h}} \{C_{(k)^h}(P_{(k-1)^h}, P_{(k)^h}, P_{(k+1)^h}) + C^*_{(k+1)^h}(P_{(k)^h}, P_{(k+1)^h})\}$;

fin

fin

fin

Retourner $\min_{P_{(2)^h} \in \mathcal{P}_{(2)^h}} C_{(1)^h}(P_{(0)^h}, P_{(1)^h}, P_{(2)^h}) + C^*_{(2)^h}(P_{(1)^h}, P_{(2)^h})$;

fin

Algorithme 6: Algorithme de placement de données optimal sur une étoile généralisée"

Données : Une instance du problème de placement de données sur un réseau d'utilisateurs reliés suivant une topologie d'étoile généralisée. Le neoud central est noté γ.

Résultat : Un placement de données optimal.

début

    pour chaque $P_\gamma \in \mathcal{P}_\gamma$ **faire**

        CoûtTotal(P_γ) = 0 ;

        pour $j = 1$ *à* Δ **faire**

            M_j =TailleChaîne(j) ;

            $CotTotal(P_\gamma)$ =CoûtTotal(P_γ)+CoutChaîne(j, M_j, P_γ) ;

        fin

    fin

    Retourner $\min_{P_\gamma \in \mathcal{P}_\gamma}$CoûtTotal($P_\gamma$) ;

fin

programme dynamique défini précédement pour tous les placements possibles sur les noeuds supprimés.

Le problème de suppression de noeuds

À partir d'une propriété de graphes π, le problème de suppression de noeuds est défini de la manière suivante : il s'agit, dans d'un graphe donné, de supprimer un minimum de noeuds afin de trouver un sous-graphe vérifiant la propriété π.

Dans la suite, nous allons considérer le problème de suppression de noeuds pour des propriétés de graphes vérifiant les hypothèses suivantes :

– hérédité : si π est vraie pour un graphe G, alors elle reste vraie également pour tous les sous-graphes induits par la suppression de certains noeuds.

– non triviale : π est vraie pour un noeud isolé, mais ne l'est pas pour tous les graphes.

– Facile : le problème de décision pour savoir si un graphe donné vérifie π ou non est dans P.

– Intéressante : les taille des graphes vérifiant la propriété π ne sont pas bornées.

Théorème 10 ([82]) *Le problème de suppression de noeuds pour une propriété π héréditaire, non triviale, facile et intéressante, est NP-complet.*

Corollaire 4 *Soit π la propriété sur les graphes contenant uniquement des chaînes et des anneaux. Le problème de suppression de noeuds pour la propriété π est $NP-complet$.*

Preuve Il suffit de remarquer que la propriété π est héréditaire, non triviale, facile et intéressante.

□

5.6 C

Nous nous sommes intéressés dans ce chapitre au problème de placement de données sur un nombre quelconques d'utilisateurs connectés entre eux par un réseau et où les données sont modélisées par un ensemble d'objets de tailles uniformes. Dans un premier temps, nous avons construit un algorithme polynomial basé sur la programmation dynamique pour résoudre d'une façon optimale le problème lorsque les utilisateurs sont reliés par une chaîne.

Ensuite, nous avons adapté cette approche pour résoudre efficacement le cas où la topologie du réseau de connexion est un anneau puis une étoile généralisée. Finalement, nous avons considéré un cas plus général où la topologie du réseau est quelconque et telle que le nombre d'utilisateurs avec un degré au moins égal à 3 est constant. Nous avons prouvé dans ce cas qu'il n'est pas possible d'utiliser l'algorithme de placement de données sur une chaîne pour aboutir à une solution optimale. Nous avons en effet montré que le problème de suppression de noeuds pour aboutir à un graphe ne contenant que des chaînes et des anneaux était NP-complet. Ainsi, le problème de placement de données reste ouvert dans le cas de topologies quelconques.

Il serait, aussi, intéressant de considérer d'autres variantes du problèmes où par exemple on rajoute une contrainte économique qui consiste à allouer à chaque utilisateur un budget limité pour convaincre les autres utilisateurs de répliquer localement des données spécifiques.

Conclusion

Dans cette thèse, nous avons étudié, en deux parties, plusieurs problèmes d'optimisation combinatoire posés dans le contexte des architectures parallèles ainsi que dans le contexte des réseaux informatiques.

Problèmes de minimisation de l'énergie

La première partie de la thèse a été consacrée à l'étude de problèmes d'ordonnancement dans le but de minimiser l'énergie consommée. Dans ce contexte, l'ordonnancement constitue un moyen parmi d'autres pour tenter de maîtriser la consommation d'énergie dans les systèmes informatiques. En effet, l'efficacité énergétique devient de plus en plus un paramètre important dans la conception des protocoles et des mécanismes de calcul et d'échange des données.

Dans un premier temps, nous avons considéré le modèle d'adaptation de la vitesse (*Speed Scaling*). Nous avons proposé un algorithme optimal basé sur une réduction en un problème de maximisation de flot pour calculer un ordonnancement qui minimise l'énergie consommée par un ensemble de machines identiques tout en respectant les dates d'arrivée et les dates d'échéance des tâches. Notons que le passage d'un problème d'ordonnancement à un problème de flot maximum constitue une technique qui a été longtemps utilisée pour résoudre des problèmes d'ordonnancement classiques dans le contexte de machines parallèles. Nous avons donc montré qu'il est possible d'adapter cette approche pour résoudre une nouvelle catégorie de problèmes d'ordonnancement où, en plus des objectifs classiques, on cherche à optimiser le critère de l'énergie. Ceci nous conduit à poser comme perspective l'utilisation des flots pour tenter de

résoudre d'autres variantes du problème de minimisation de l'énergie consommée sous le modèle d'adaptation de la vitesse ou sous d'autres modèles de calcul de l'énergie comme ceux où on introduit des états de veilles.

Dans un deuxième temps, nous nous sommes intéressés au cas où on rajoute un état de veille au modèle d'adaptation de la vitesse. Dans ce cas, le problème devient plus compliqué car il faut fixer à tout instant l'état de chaque machine, *i.e.* en activité ou en veille, en plus de sa vitesse et de la tâche en exécution. Nous avons donc proposé, dans le cas d'une machine unique et d'instances agréables, un programme dynamique permettant de retourner une solution optimale qui minimise la consommation d'énergie. Une extension naturelle de ce travail serait donc de considérer le cas de plusieurs machines identiques ou hétérogènes. Une autre direction à explorer consiste à étudier d'autres modèles où on autorise plusieurs états différents de basse consommation d'énergie.

Finalement, nous avons considéré le cas où les machines sont hétérogènes dans le sens où, la date d'arrivée et la durée d'exécution de chaque tâche ainsi que l'énergie consommée suite à son exécution dépendent de la machine d'affectation. Nous avons aussi supposé qu'il existe un ensemble fini de vitesses pour les machines et que chaque tâche est exécutée sans préemption à une vitesse constante. Dans ce modèle, en plus de la machine d'affectation, la durée d'exécution et l'énergie pour chaque tâche sont aussi fonctions de la vitesse d'exécution.

Notons que par rapport aux problèmes précédents, nous n'avons imposé aucun modèle de calcul de l'énergie consommée dans ce problème. Par ailleurs, nous avons adapté un algorithme d'approximation randomisé basé sur la technique de l'arrondi aléatoire pour résoudre le problème de minimisation de l'énergie plus la somme des temps de complétude pondérés ainsi que le problème de minimisation de la somme des temps de complétude pondérés avec un budget fixe d'énergie qu'on ne souhaite pas dépasser.
Ayant réussi à proposer une version dérandomisée de l'algorithme pour le premier problème, la question reste ouverte quant à la dérandomisation pour le problème avec budget d'énergie. Une

autre question intéressante consiste à étendre ce travail dans le cas où les vitesses des machines forment un spectre continu de valeurs positives. Enfin, en la présence des dates d'arrivée pour les tâches, il semble judicieux d'étudier le cas où on cherche à minimiser une fonction objectif qui, en plus de l'énergie, tient compte des temps de réponse des tâches, *i.e.* le temps total passé par une tâche dans le système entre sa date d'arrivée et sa date de complétude.

Problèmes d'optimisation classiques

Dans la deuxième partie de cette thèse nous avons étudié deux problèmes d'optimisation classiques.

Nous avons considéré, en premier lieu, le problème d'ordonnancement dans un graphe d'ordre d'intervalles sur des machines dédiées et en présence de délais de communication. Nous avons aussi considéré des tâches unitaires avec des dates d'arrivée et d'échéance.

Tout d'abord, nous avons montré que le problème est NP-complet au sens fort dans le cas où les temps de communication sont quelconques. Ensuite, dans le cas des temps de communication monotones, nous avons présenté une extension d'un algorithme de liste de la littérature et nous avons montré qu'elle est capable de retourner une solution réalisable en un temps polynomial. Enfin, nous avons montré à travers un exemple simple qu'il n'existe pas d'algorithme de liste pour résoudre le problème avec des machines typées, *i.e.* plusieurs machines identiques pour chaque type de tâches. On en déduit donc une première perspective qui consiste à étudier le problème dans le cas des machines typées et concevoir des algorithmes efficaces pour le résoudre. Par ailleurs, en essayant d'appliquer notre algorithme dans le cas des machines typées et des temps de communication unitaires, nous nous sommes aperçus que la procédure de modification des dates d'échéance n'améliore pas forcément celles des tâches. Ainsi, l'algorithme risque de retourner des solutions non réalisables. Il serait donc intéressant d'adapter cet algorithme dans le cas des machines typées et des temps de communication unitaires.

Le dernier problème que nous avons étudié est le problème de placement de données dans un

réseau pair-à-pair. Le but étant de minimiser le coût d'accès des utilisateurs aux données, nous avons décrit un algorithme de programmation dynamique pour résoudre ce problème lorsque la topologie du réseau est une chaîne et lorsque les données sont modélisées par des objets de tailles uniformes. Nous avons adapté, par la suite, cet algorithme pour résoudre le problème pour des topologies en anneau et en étoile généralisée. Finalement, nous n'avons pas réussi à étendre cette approche pour des graphes quelconques avec un nombre constant de noeuds de degré supérieur ou égal à 3. En effet, nous avons prouvé que le problème de suppression de noeuds pour décomposer le graphe en des sous graphes connexes en chaînes et en anneaux est NP-complet.

Pour poursuivre ce travail, il serait intéressant d'étudier les cas où le problème de placement de données peut être résolu pour des graphes plus généraux. Une autre perspective consiste à rajouter une dimension économique au problème comme par exemple l'attribution d'un budget à chaque utilisateur pour convaincre les autres noeuds de répliquer des données spécifiques.

Bibliographie

[1] F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, and M. Sviridenko. Approximation schemes for minimizing average weighted completion time with release dates. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 32–44, Washington, DC, USA, 1999. IEEE Computer Society.

[2] F. Afrati, E. Bampis, L. Finta, and I. Milis. Scheduling trees with large communication delays on two identical processors. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 288–295, London, UK, 2000. Springer-Verlag.

[3] S. Albers. Algorithms for dynamic speed scaling. In T. Schwentick and C. Dürr, editors, *28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*, volume 9 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–11, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] S. Albers and A. Antoniadis. Race to idle : new algorithms for speed scaling with a sleep state. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 1266–1285. SIAM, 2012.

[5] S. Albers, A. Antoniadis, and G. Greiner. On multi-processor speed scaling with migration : extended abstract. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 279–288, New York, NY, USA, 2011. ACM.

[6] S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Trans. Algorithms*, 3, November 2007.

[7] S. Albers, F. Müller, and S. Schmelzer. Speed scaling on parallel processors. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 289–298, New York, NY, USA, 2007. ACM.

[8] H. Ali and H. E. Rewini. An optimal algorithm for scheduling interval ordered tasks with communication on n processors. *Journal of Computer and System Sciences*, 51(2) :301 – 306, 1995.

[9] E. Angel, E. Bampis, and V. Chau. Low complexity scheduling algorithm minimizing the energy for tasks with agreeable deadlines. In *The 10th Latin American Symposium on Theoretical Informatics (LATIN2012)*, 2012.

[10] E. Angel, E. Bampis, F. Kacem, and D. Letsios. Speed scaling on parallel processors with migration. *CoRR*, abs/1107.2105, 2011.

[11] E. Angel, E. Bampis, and A. Kononov. A fptas for approximating the unrelated parallel machines scheduling problem with costs. In F. auf der Heide, editor, *Algorithms - ESA 2001*, volume 2161 of *Lecture Notes in Computer Science*, pages 194–205. Springer Berlin / Heidelberg, 2001.

[12] E. Angel, E. Bampis, and A. Kononov. On the approximate tradeoff for bicriteria batching and parallel machine scheduling problems. *Theor. Comput. Sci.*, 306 :319–338, September 2003.

[13] E. Angel, E. Bampis, G. G. Pollatos, and V. Zissimopoulos. Optimal data placement on networks with constant number of clients. *CoRR*, abs/1004.4420, 2010.

[14] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, STOC '92, pages 571–580, New York, NY, USA, 1992. ACM.

[15] C. P. B. Chen and G. Woeginger. *A review of machine scheduling : Complexity, algorighms and approximability*, pages 21–169. Kluwer Academic Publishers, d.-z. du and p. pardalos edition, 1998.

[16] I. Baev, R. Rajaraman, and C. Swamy. Approximation algorithms for data placement problems. *SIAM Journal on Computing*, 38(4) :1411–1429, 2008.

[17] I. D. Baev and R. Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, SODA '01, pages 661–670, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[18] N. Bansal, T. Kimbrel, and K. Pruhs. Speed scaling to manage energy and temperature. *J. ACM*, 54 :1–39, March 2007.

[19] N. Bansal, K. Pruhs, and C. Stein. Speed scaling for weighted flow time. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '07, pages 805–813, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[20] P. Baptiste, M. Chrobak, and C. Dürr. Polynomial time algorithms for minimum energy scheduling. In L. Arge, M. Hoffmann, and E. Welzl, editors, *Algorithms - ESA 2007*, volume 4698 of *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin / Heidelberg, 2007.

[21] B. D. Bingham and M. R. Greenstreet. Energy optimal scheduling on multiprocessors with migration. In *Proceedings of the 2008 International Symposium on Parallel and Distributed Processing with Applications ISPA 2008*, pages 153–161, 2008.

[22] C. E. Bonferroni. *Teoria statistica delle classi e calcolo delle probabilita.* Pubbl. d. R. Ist. Super. di Sci. Econom. e Commerciali di Firenze. 8 Firenze : Libr. Internaz. Seeber. 62 S., 1936.

[23] F. A. Bower, D. J. Sorin, and L. P. Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE Micro*, 28 :17–25, May 2008.

[24] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture : Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6) :26–44, nov 2000.

[25] P. Brucker. *Scheduling Algorithms*. SpringerVerlag, 2004.

[26] P. Brucker and S. Knust. Complexity results for single-machine problems with positive finish-start time-lags. *Computing*, 63 :299–316, 1999.

[27] J. Bruno, E. G. Coffman, Jr., and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Commun. ACM*, 17 :382–387, July 1974.

[28] D. P. Bunde. Power-aware scheduling for makespan and flow. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '06, pages 190–196, New York, NY, USA, 2006. ACM.

[29] R. A. Carrasco, G. Iyengar, and C. Stein. Energy aware scheduling for weighted completion time and weighted tardiness. *CoRR*, abs/1110.0685, 2011.

[30] H.-L. Chan, J. W.-T. Chan, T.-W. Lam, L.-K. Lee, K.-S. Mak, and P. W. H. Wong. Optimizing throughput and energy in online deadline scheduling. *ACM Trans. Algorithms*, 6 :1–22, December 2009.

[31] S.-H. Chan, T.-W. Lam, L.-K. Lee, H.-F. Ting, and P. Zhang. Non-clairvoyant scheduling for weighted flow time and energy on speed bounded processors. In *Proceedings of the Sixteenth Symposium on Computing : the Australasian Theory - Volume 109*, CATS '10, pages 3–10. Australian Computer Society, Inc., 2010.

[32] C. Chekuri and S. Khanna. A ptas for minimizing weighted completion time on uniformly related machines. In F. Orejas, P. Spirakis, and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 848–861. Springer Berlin / Heidelberg, 2001.

[33] C. Chekuri and S. Khanna. Approximation algorithms for minimizing average weighted completion time. *Handbook of Scheduling : Algorithms, Models, and Performance Analysis. CRC Press, Inc., Boca Raton, FL, USA*, 2004.

[34] J.-J. Chen, H.-R. Hsu, K.-H. Chuang, C.-L. Yang, A.-C. Pang, and T.-W. Kuo. Multiprocessor energy-efficient scheduling with task migration considerations. In *Proceedings of the 16th*

Euromicro Conference on Real-Time Systems, pages 101–108, Washington, DC, USA, 2004. IEEE Computer Society.

[35] P. Chrétienne and C. Picouleau. Scheduling with communication delays : A survey. *Scheduling Theory and Its Applications*, 1995.

[36] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[37] B. D. de Dinechin. Scheduling monotone interval orders on typed task systems. In *26th Workshop of the UK Planning and Scheduling Special Interest Group*, pages 25–31, 2007.

[38] X. Deng, H.-N. Liu, and B. Xiao. Deterministic load balancing in computer networks. In *Proceedings of the 1990 IEEE Second Symposium on Parallel and Distributed Processing*, SPDP '90, pages 50–57, Washington, DC, USA, 1990. IEEE Computer Society.

[39] M. J. Flynn, P. Hung, and K. W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, 19(4) :11–22, jul 1999.

[40] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[41] R. Giroudeau, J. König, F. Moulaï, and J. Palaysi. Complexity and approximation for the precedence constrained scheduling problem with large communication delays. In J. Cunha and P. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 622–622. Springer Berlin / Heidelberg, 2005.

[42] R. Graham, E. Lawler, J. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. In E. J. P.L. Hammer and B. Korte, editors, *Discrete Optimization II Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver*, volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979.

[43] G. Greiner, T. Nonner, and A. Souza. The bell is ringing in speed-scaled multiprocessor scheduling. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 11–18, New York, NY, USA, 2009. ACM.

[44] A. Gupta, R. Krishnaswamy, and K. Pruhs. Scalably scheduling power-heterogeneous processors. In *Proceedings of the 37th international colloquium conference on Automata, languages and programming*, ICALP'10, pages 312–323, Berlin, Heidelberg, 2010. Springer-Verlag.

[45] L. A. Hall, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time : off-line and on-line algorithms. In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '96, pages 142–151, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[46] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1) :177–185, 1974.

[47] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18 :244–257, April 1989.

[48] S. Irani and K. R. Pruhs. Algorithmic problems in power management. *SIGACT News*, 36 :63–76, June 2005.

[49] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. *ACM Trans. Algorithms*, 3, November 2007.

[50] J. Jackson. *Scheduling a production line to minimize maximum tardiness*. Research report. University of California, 1955.

[51] K. Jansen. Analysis of scheduling problems with typed task systems. *Discrete Applied Mathematics*, 52(3) :223 – 232, 1994.

[52] K. Jansen and L. Porkolab. Improved approximation schemes for scheduling unrelated parallel machines. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, STOC '99, pages 408–417, New York, NY, USA, 1999. ACM.

[53] R. M. Karp. Reducibility among combinatorial problems. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 219–241. Springer Berlin Heidelberg, 2010.

[54] W. Karush. Minima of functions of several variables with inequalities as side conditions. Master's thesis, Department of Mathematics, University of Chicago, Chicago, IL, USA, 1939.

[55] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. Exercice 23, Chapter 7 and its solution.

[56] J. G. Koomey. Growth in data center electricity use 2005 to 2010. *The New York Times*, 3(3) :24, 2011.

[57] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492. Berkeley, California, 1951.

[58] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *SIGARCH Comput. Archit. News*, 32 :64–75, March 2004.

[59] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59(3) :381–422, dec 1999.

[60] T.-W. Lam, L.-K. Lee, H.-F. Ting, I. K. To, and P. W. Wong. Sleep with guilt and work faster to minimize flow plus energy. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming : Part I*, ICALP '09, pages 665–676, Berlin, Heidelberg, 2009. Springer-Verlag.

[61] T.-W. Lam, L.-K. Lee, I. K. K. To, and P. W. H. Wong. Competitive non-migratory scheduling for flow time and energy. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 256–264, New York, NY, USA, 2008. ACM.

[62] A. Leung, K. V. Palem, and A. Pnueli. Scheduling time-constrained instructions on pipelined processors. *ACM Trans. Program. Lang. Syst.*, 23 :73–103, January 2001.

[63] M. Li, A. C. Yao, and F. F. Yao. Discrete and continuous min-energy schedules for variable voltage processors. *Proceedings of the National Academy of Sciences of the United States of America*, 103(11) :3983–3987, 2006.

[64] J.-H. Lin and J. S. Vitter. epsilon-approximations with minimum packing constraint violation (extended abstract). In *STOC'92*, pages 771–782, 1992.

[65] R. Merritt. Cpu designers debate multi-core future. *EE Times*, February 2008.

[66] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Comput. Archit. Lett.*, 5 :4–17, January 2006.

[67] A. Munier, F. Kacem, B. D. de Dinechin, and L. Finta. Minimizing the makespan for an interval ordered precedence graph on m processors with communication delays and unit execution time tasks. In *9th Workshop on Models and Algorithms for Planning and Scheduling Problems (MAPSP'09)*, pages 8–10, Rolduc, Pays Bas, Juin 2009.

[68] Y. Nesterov and A. Nemirovskii. *Interior-Point Polynomial Algorithms in Convex Programming*. Society for Industrial and Applied Mathematics, Philadephia, PA, 1994.

[69] E. Néron, P. Baptiste, and F. Sourd. Modèles et algorithmes en ordonnancement. *J. ACM*, pages 198–201.

[70] K. V. Palem and B. B. Simons. Scheduling time-critical instructions on risc machines. *ACM Trans. Program. Lang. Syst.*, 15 :632–658, September 1993.

[71] C. H. Papadimitriou and M. Yannakakis. Scheduling interval-ordered tasks. *SIAM Journal on Computing*, 8(3) :405–409, 1979.

[72] K. Pruhs, P. Uthaisombut, and G. Woeginger. Getting the best response for your erg. *ACM Trans. Algorithms*, 4 :38 :1–17, July 2008.

[73] K. Pruhs, R. van Stee, and P. Uthaisombut. Speed scaling of tasks with precedence constraints. *Theor. Comp. Sys.*, 43 :67–80, March 2008.

[74] P. Raghavan and C. Tompson. Randomized rounding : A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7 :365–374, 1987.

[75] A. S. Schulz and M. Skutella. Scheduling unrelated machines by randomized rounding. *SIAM Journal on Discrete Mathematics*, 15(4) :450–469, 2002.

[76] D. B. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Math. Program.*, 62 :461–474, December 1993.

[77] M. Skutella. Convex quadratic and semidefinite programming relaxations in scheduling. *J. ACM*, 48 :206–242, March 2001.

[78] M. A. Trick. Scheduling multiple variable-speed machines. *Operations Research*, 42(2) :234–248, 1994.

[79] B. Veltman, B. Lageweg, and J. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16(2-3) :173 – 182, 1990.

[80] J. Verriet. Scheduling interval-ordered tasks with non-uniform deadlines subject to non-zero communication delays. *Parallel Computing*, 25(1) :3 – 21, 1999.

[81] W. Wu, M. Li, and E. Chen. Min-energy scheduling for aligned jobs in accelerate model. In Y. Dong, D.-Z. Du, and O. Ibarra, editors, *Algorithms and Computation*, volume 5878 of *Lecture Notes in Computer Science*, pages 462–472. Springer Berlin / Heidelberg, 2009.

[82] M. Yannakakis. Node-and edge-deletion np-complete problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC ’78, pages 253–264, New York, NY, USA, 1978. ACM.

[83] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS ’95, pages 374–382, Washington, DC, USA, 1995. IEEE Computer Society.

[84] W. Yu, H. Hoogeveen, and J. K. Lenstra. Minimizing makespan in a two-machine flow shop with delays and unit-time operations is np-hard. *Journal of Scheduling*, 7 :333–348, 2004.