HAL
open science

Ordonnancement de tâches parallèles dans les environnements fortement perturbés

Adel Safi

To cite this version:

Adel Safi. Ordonnancement de tâches parallèles dans les environnements fortement perturbés. Autre [cs.OH]. Université de Grenoble; 351 Université de la Manouba, 2012. Français. NNT : 2012GRENM093 . tel-00846477

HAL Id: tel-00846477

https://theses.hal.science/tel-00846477

Submitted on 19 Jul 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

préparée dans le cadre d'une cotutelle entre l'*Université de Grenoble* et l'Université de la Mannouba

Spécialité : **Informatique**

Arrêté ministériel : le 6 janvier 2005 -7 août 2006

Présentée par

Adel SAFI

Thèse dirigée par **Denis TRYSTRAM** et **Mohamed JEMNI**

préparée au sein des **Laboratoires LIG (Grenoble, France) et (LaTICE, Tunisie)**

Ordonnancement de tâches parallèles dans des environnements fortement perturbés

Thèse soutenue publiquement le **15 Octobre 2012**
devant le jury composé de :

M. Frédéric Guinand Professeur, Université du Havre, France,	Président
M. Imed KACEM Professeur, Université de Paul Verlaine – Metz, Metz, France	Rapporteur
M. Wahid NASRI MdC, Université de Tunis, Tunis, Tunisie	Rapporteur
M. Christophe CERIN Professeur, université de Paris XIII, Paris, France	Rapporteur
M. Denis TRYSTAM Professeur, Université de Grenoble, Grenoble, France	Directeur
M. Mohamed JEMNI Professeur, Université de Tunis, Tunis, Tunisie	Directeur
M. Gregory MOUNIE MdC, Université de Grenoble, Grenoble, France	Examinateur

Remerciements

Au début, je voudrais adresser les remerciements à Monsieur Frédéric Guinand, Professeur à l'université du Havre pour avoir accepter de présider le jury de ma soutenance de thèse. Mes remerciement s'adressent aussi à messieurs Wahid Nasir, MdC à l'université de Tunis, Imed Kacem, Professeur à l'université de Paul Verlaine – Metz et Christophe CERIN, Professeur à Université de Paris XIII pour avoir accepté de rapporter et de juger mon travail. Je leur suis très reconnaissant de l'intérêt qu'ils avaient porté à mes travaux de recherche, tout en ayant un regard critique et constructif. Je tiens donc à leur exprimer ma profonde gratitude.

Par ailleurs, je tiens aussi à remercier Monsieur Grégory Mounié, MdC à l'université de Grenoble et Monsieur Louis-claude Canon, MdC à l'université de Franche-Comté pour m'avoir accompagné dans les moments les plus critiques de la thèse. Leurs contributions et directives m'ont été, certes, de la plus grande utilité.

Ma plus grande reconnaissance s'adresse aussi à mes directeurs de thèse Mohamed JEMNI, Professeur à l'université de Tunis, et Denis TRYSTRAM, Professeur à l'université de Grenoble qui ont accepté de m'encadrer dans ce travail de thèse. En effet, durant toute la période des travaux, j'ai toujours senti qu'ils ont mis tous les moyens humains et logistiques pour que la thèse aboutisse. Je n'oublierais pas d'adresser mes sincères respects à Monsieur Zaher Mahjoub, Professeur à l'université de Tunis-el Manar, pour m'avoir intégré dans le domaine de recherche et avoir conduit mes travaux pour l'obtenions du diplôme des études approfondies. Je tiens aussi à remercier les membres des équipes «calcul parallèle» du laboratoire LaTICE et «ordo»du Laboratoire LIG pour leurs interactions et leurs idées constructives au long de la période de la thèse. De plus, les membres de ces laboratoires ont toujours été attentifs et serviables aussi bien au niveau professionnel que humain.

Finalement, j'adresse ma reconnaissance et mon amour aux membres de ma famille et à mes amis. Je pense particulièrement à ma femme Hana qui n'a pas cessé de m'encourager ainsi qu'à ma grande mère, Fatma, en lui espérant longue et heureuse vie.

À ma femme, oh combien patiente
À mes enfants
À ma famille
À mes amis et à tous ceux pour qui je compte

Table des matières

1 Introduction **1**
1.1 Les plateformes du calcul parallèle 1
1.2 Motivation et contexte 3
1.3 Contributions 4

2 État de l'Art **7**
2.1 Introduction 7
2.2 ordo. dans les nouvelles plateformes 8
2.2.1 Nouvelles plateformes d'exécution de calcul distribué . 8
2.2.2 Étude de disponibilités de ressources dans les plateformes de calcul distribué 11
2.2.3 Notation du problème d'ordonnancement sous contraintes d'indisponibilités 14
2.2.4 Ordonnancement avec contraintes d'indisponibilité de ressources 15
2.2.5 Problèmes connexes 19
2.2.6 Sources d'indisponibilité et d'incertitude 20
2.3 Vivre avec les incertitudes 22
2.3.1 Motivation 22
2.3.2 Les Sources d'incertitude dans les problèmes d'ordonnancement 22
2.3.3 Modèles de l'incertitude 24
2.3.4 Résolution du problème d'ordonnancement avec incertitude 25
2.3.5 Construction d'algorithmes d'ordonnancement flexible . 26
2.3.6 Mesure de performance 27
2.4 Stabilisation des ordonnancements grâce aux tampons 29
2.5 Définition du problème étudié 31

2.5.1 Définition du workload 31
2.5.2 Définition de la plateforme d'exécution 32
2.5.3 Définition de la solution (ordonnancement) et des mesures 32
2.5.4 Modèle de perturbation 33
2.6 Tableau récapitulatif 34
2.7 Conclusion 34

3 Étude préliminaire 37
3.1 Introduction 37
3.2 Modèle d'applications parallèles 38
3.3 Définition du problème 39
3.4 Analyse des algorithmes basées sur LPT 42
3.4.1 Analyse de l'algorithme LPT classique 42
3.4.2 Stable-LPT 45
3.5 Nouvelle famille d'algorithmes flexibles 46
3.5.1 Principe 46
3.5.2 Description 46
3.5.3 Analyse de performance de fLPT 47
3.5.4 Borne inférieure pour le pire des cas 49
3.6 Résumé des résultats 50
3.7 Conclusion 51

4 Algorithme bi-objectif 53
4.1 Introduction 53
4.2 Modèle de perturbation 54
4.3 Définition du problème 54
4.4 Analyse de stabilité 55
4.5 Algorithme bi-objectif 57
4.5.1 Description 57
4.5.2 Analyse théorique 59
4.6 Conclusion 61

5 Conception d'algorithmes 63
5.1 Introduction 63
5.2 Définitions et modèles 64
5.3 Stabilisation par approche de tampon 65
5.3.1 règle de tampon 65

5.3.2 Règle de tampon paramétré 69
5.4 Analyse de la stabilité 70
5.4.1 Algorithme d'exécution dynamique 70
5.4.2 Etude du cas où l'indisponibilité avance 71
5.4.3 Analyse du cas général 75
5.5 Algorithmes bi-objectifs pour l'ordonnancement avec contraintes d'indisponibilité 77
5.5.1 Algorithme Glouton 78
5.5.2 Heuristiques basées sur les intervalles 79
5.5.3 Résumés des heuristiques conçues 81
5.6 Conclusion 84

6 Simulations 85
6.1 Introduction 85
6.2 Caractéristiques des instances 85
6.2.1 Le workload 86
6.2.2 La plateforme cible 86
6.3 Simulation de GAPS 88
6.3.1 Plan de l'expérience 88
6.3.2 Interprétation 88
6.4 Expérimentation 90
6.4.1 Performances des variantes proposées 91
6.4.2 Stabilité 91
6.4.3 Impact du paramètre β 95
6.4.4 Caractérisation de la sensibilité des processeurs 97
6.4.5 Effet de la loi de probabilité 99
6.4.6 Étude expérimentale du regret sur des scénarios clairvoyants 104
6.4.7 Complexité temporelle 105
6.5 Conclusion 106

7 Conclusion 113

Table des figures

2.1 Loi de Moore : Correspondance entre les valeurs théoriques et les valeurs réelles 8
2.2 Le calcul global : Collection de ressources individuelles connectées à travers Internet 9
2.3 Une grille de calcul : un ensemble de clusters liés à travers Internet 9
2.4 Architectures possibles des grilles de calcul 10
2.5 Fenêtre de préférence de BOINC 12
2.6 La stabilité mesure la distance entre la performance d'un algorithme pour deux scénarios 29
2.7 Représentation de l'intervalle k sur la machine i, qui contient une période de disponibilité suivie d'une période d'indisponibilité qui commencent aux dates e_i^{k-1} (date de fin de l'intervalle $k-1$) et s_i^k. Les machines entièrement disponibles ne contiennent aucune indisponibilité 32

3.1 Illustration des variables 40
3.2 Partitionnement de l'espace disponible en deux zones 46
3.3 Borne inférieure pour le pire des cas 50
3.4 Ratio d'approximation et ratio de stabilité de la famille d'algorithmes fLPT 51

4.1 L'indisponibilité k peut commencer plus tôt à cause de la perturbation δ_i^k. 54
4.2 Forme d'un ordonnancement qui respecte la règle de tampon . 57

5.1 Récurence sur les intervalles de la preuve de la proposition 11 66

5.2 Cas initial pour la proposition 11 : L'ordonnancement initial est sur la première ligne et les trois autres cas sur les lignes qui suivent 67
5.3 Première ligne : ordonnancement avec deux tâches. Seconde ligne : le pire cas d'exécution de cet ordonnancement. 73
5.4 Première ligne : Le pire cas pour l'ordonnancement avec un intervalle unique quand l'indisponibilité avance. Seconde ligne : l'ordonnancement produit au pire cas 75
5.5 Schéma récapitulatif des combinaisons d'heuristiques possibles 83

6.1 Courbe de la fonction de distribution cumulative des durées de tâches 87
6.2 Effet du paramètre β sur les ratios de stabilité et de makespan de GAPS. Chacune des 1100 mesures représentent une simulation avec 300 processeurs et 3000 tâches. Pour chacune des 11 valeurs de β, GAPS est exécuté 100 fois avec différentes instances. 89
6.3 Horizons des variantes conçues 92
6.4 Ratio de performance des variantes 93
6.5 Occurrences avec makespan perturbé infini 94
6.6 Stabilité en fonction de β 95
6.7 Nombre d'occurrences infinies en fonction de β 96
6.8 Stabilité des ordonnancements en fonction de β 98
6.9 Durée des disponibilités des processeurs sensibles et insensibles 100
6.10 Nombre d'intervalles des processeurs sensibles et stables 101
6.11 ECDF des moyennes des temps d'exécution 102
6.12 Nombre d'instances non-réalisable 107
6.13 Moyenne des makespan non perturbé 108
6.14 Performances des algorithmes clairvoyants 109
6.15 Rapport entre l'horizon clairvoyant et l'horizon initial 110
6.16 Durée d'exécution des variantes conçues 111
6.17 Durée d'exécution des variantes conçues (sans D/D) 112

Liste des tableaux

2.1 Résumé des notations 35

3.1 Tableau comparatif des différents résultats 51

Chapitre 1

Introduction

1.1 Les plateformes du calcul parallèle

Le calcul parallèle a été développé au milieu des années 70 avec la naissance des machines vectorielles conçues spécialement pour exécuter des applications numériques qui manipulent des données régulières (comme des vecteurs et des matrices)[21]. Le principe de ces applications est de faire un découpage fin du code source adapté à l'architecture de la machine cible. Cette décomposition prend en considération les propriétés les plus techniques des calculateurs (comme la taille du registre, la vitesse des unités de calcul, la bande passante de la mémoire).

Avec le progrès très rapide qu'avait connu l'électronique au milieu des années 60, Gordon Moore, avait énoncé une loi qui s'est avérée correcte au cours du temps. La loi de Moore annonce que le nombre de transistors dans un microprocesseur double environ tous les 18 mois, ce qui implique qu'il en va de même pour la puissance de calcul des supercalculateurs. Malgré l'avantage évident des supercalculateurs présentés en terme de performance (et aussi en image de marque de constructeur), leur coût reste généralement prohibitif et inaccessible pour la plupart.

Avec l'explosion que connait Internet depuis les années 2000, il est devenu possible de faire collaborer un nombre, parfois très grand, de machines éparpillées partout dans le monde afin d'exécuter un calcul. Cette technique permet de construire un supercalculateur virtuel de dimension planétaire. On parle alors de Grille de calcul [33]. Le coût de la mise en place d'une telle plateforme reste peu couteux vis-à-vis des clusters et des machines parallèles

dédiées. Cependant les grilles posent plusieurs problèmes de disponibilité, de fiabilité et de sécurité (du calcul et des données).

La motivation à la construction de telles plateformes provient du fait que la moyenne mondiale du nombre de cycles processeurs, d'une machine de bureau, qui ne sont pas utilisés est estimé à 75%. Cette moyenne est estimée à 47% en entreprise [1].

Actuellement, il existe deux approches pour la construction des grilles de calcul. La première est connue sous le nom de calcul global (appelé aussi Internet Computing ou desktop grid) qui consiste à collecter un ensemble de ressources le plus souvent données par des volontaires[35]. Le projet le plus connu dans la communauté est certainement SETI@HOME [9]. La deuxième approche est plutôt institutionnelle et consiste à rassembler un ensemble de clusters reliés à travers Internet pour former une seule entité logique, offrant une capacité de calcul (ou de service) énorme. Les projets français grid5000 [2] et européen EGEE [3] adoptent cette architecture.

De point de vue architecture, il existe trois organisations possibles de grille :

- grille à architecture client/serveur : dans ce type de grille, les communications se font uniquement entre les ressources et un serveur centralisé. Aucune communication/collaboration ne se fait directement entre les ressources. Les projets Boinc [8], SETI@Home [9], United Devices [2], Distributed.net [1], XtremWeb [35] adoptent cette architecture.
- grille à architecture centralisée : dans ce type de grille, les unités de calcul se connectent à un seul serveur qui se charge de leur transmettre les données et de collecter le résultat du calcul. Cependant, les ressources peuvent communiquer entre elles pour échanger des données. Ce type de grille est adapté aux applications distribuées où les tâches font de l'échange de données.
- grille de calcul décentralisée : dans ce genre de grille, il n'existe aucun élément central. Tous les noeuds ont la même importance et peuvent être producteurs ou consommateurs de données. Organic Grid [10], Messor [68], Paradropper [64] et Vigne [47] sont des exemples de projets avec architectures décentralisées.

1. Source : Omni Consulting Group.
2. www.grid5000.fr
3. http ://www.eu-egee.org/

1.2 Motivation et contexte

Dans les systèmes de grilles de calcul, chaque ressource prend deux états en terme de disponibilité : disponible et indisponible. Une ressource est dite disponible quand on peut l'utiliser pour faire du calcul, sinon, elle est considérée comme indisponible.

Cette disponibilité est gérée d'une manière différente selon la nature du projet. Dans un système de calcul global, tel que BOINC ou XtremWeb, il existe un élément central auquel les utilisateurs se connectent et soumettent leurs travaux (calcul). D'autre part, les volontaires connectent leurs ressources à travers un logiciel qu'ils installent sur leur PC et qui se charge de demander du travail au serveur s'il s'aperçoit que le processeur est libre. Le serveur va donc envoyer du travail aux clients. C'est la stratégie d'ordonnancement qui décide quel travail envoyer à quel PC. Après avoir effectué le calcul, le PC délivre le résultat au serveur.

D'autre part, dans les grilles de calcul, l'administration des ressources se fait par des stratégies élaborées selon des protocoles moins simple. Cette administration se fait par un nombre très réduit de personnes et qui obéissent à des protocoles définis et généralement connus par les sites partenaires. Avec ces projets, la soumission de l'application se fait à travers une machine frontale. Les tâches de l'application sont ensuite transférées à la file d'attente de l'ordonnanceur qui se charge de les exécuter. Cette exécution dépend de la disponibilité des ressources. L'ordonnanceur va donc attendre qu'un ensemble de ressources soit disponible pour y placer les tâches.

Les tâches de l'application peuvent être indépendantes ou peuvent échanger des données. On parle alors de graphe de tâches[61]. Le résultat est ensuite sauvegardé dans des serveurs de données afin que l'utilisateur vienne les récupérer ultérieurement. Les grilles EGEE et Grid'5000 (en mode best effort) adoptent une telle stratégie.

Notons aussi que la grille française Grid'5000 offre un mécanisme sophistiqué pour la réservation et l'exécution d'applications parallèles. Un utilisateur soumet une requête pour réserver un ensemble de ressources. Si cette requête est acceptée par l'ordonnanceur OAR de grid'5000, alors l'application de cet utilisateur va être exécutée pendant cet intervalle de temps. Sur les machines concernées aucune interférence n'est possible avec les applications d'autres utilisateurs. Cependant, quand le slot temporel arrive à sa fin, les tâches encore actives vont être tuées.

Nous venons ainsi de voir que la stratégie d'exécution des tâches dépend

fortement de la disponibilité des ressources. Leur administration constitue donc un élément clé dans la performance de ces grilles. Nous dirons que une grille est administrativement homogène quand elle est administrée par une seule (ou très peu de personnes), appartenant à une même institution.

Une plateforme est dite administrativement hétérogène quand un système est maintenu par des personnes appartenant à des domaines administratifs distincts et ayant des politiques d'administration différentes.

Il existe une catégorie intermédiaire où chaque cluster possède son propre domaine d'administration mais que les domaines soient suffisamment proches (ex. ayant la même direction de tutelle) pour collaborer et se mettre d'accord sur un protocole commun de fédération de ressources. Ce type de grille est connu comme étant les grilles légères [79] telle que la grille française grid'5000.

1.3 Contributions

Nous abordons dans ce travail, le problème d'ordonnancement de tâches avec contraintes d'indisponibilité et d'incertitude. Ce problème étant NP-Complet au sens fort, nous proposons dans ce travail des schémas algorithmiques approchés qui permettent de produire des ordonnancements qui résistent aux aléas et qui ont une complexité raisonnable. Nous commençons dans le premier chapitre par étudier le cas préliminaire de l'ordonnancement d'un ensemble de tâches avec au plus une seule contrainte de disponibilité par machine. Nous analysons, dans un premier temps, l'impact des perturbations sur la performance de l'algorithme de liste LPT (longest Processing Time) et nous proposons un algorithme simple qui a la meilleure stabilité possible. Nous proposons ensuite une famille d'algorithmes d'ordonnancement paramétrés par un facteur qui est vu comme un compromis entre la stabilité est la performance des algorithmes présentés.

Dans le chapitre qui suit, nous nous attaquons au problème général de l'ordonnancement des tâches parallèles sur des processeurs avec contraintes d'indisponibilité. Un processeur dispose ainsi de plusieurs périodes d'indisponibilité qui ne se superposent pas et qui sont sujette à des perturbations. Nous proposons une famille d'algorithmes basée sur les mécanismes de tampon. Le principe est d'anticiper l'effet des perturbations sur l'éventuel retard occasionné sur les dates de fin des tâches, en laissant un espace libre devant chaque indisponibilité. La taille des tampons est paramétrée en fonction de la taille des tâches allouées à la disponibilité.

Dans le chapitre 4, nous généralisons l'étude faite au chapitre 3 pour aborder le problème général où les indisponibilité peuvent aussi bien se produire avant ou après la date initialement prévue. Nous concevons pour cette configuration un mécanisme d'anticipation et de réaction aux perturbations. Nous proposons ensuite un ensemble d'algorithmes d'ordonnancement qui respectent la contrainte de tampon définie.

Enfin, dans le dernier chapitre, nous faisons une campagne de tests pour évaluer le comportement des différents algorithmes qui ont été conçue dans les trois premiers chapitres. Ces expériences ont la plupart été réalisées en utilisant des données réalistes collectées à partir de la grille BOINC. Les résultats expérimentaux montrent que les algorithmes que nous avons conçus accomplissent une performance satisfaisante.

Chapitre 2

État de l'Art

2.1 Introduction

Nous assistons aujourd'hui à une diversité sans précédent des plateformes de calculs distribués. Cette diversité est due en grande partie à l'évolution des réseaux de communication, et en particulier Internet. Comme nous l'avons mentionné dans le chapitre précédent, ces plateformes peuvent être montées sur un espace réduit (administrativement et spacialement) ou peuvent avoir une dimension planétaire où les ressources sont dispersées au tour du monde et connectées à travers la toile.

Ces plateformes délivrent une puissance de calcul (de stockage et de services) gigantesques mais, apportent un ensemble de problèmes intéressants théoriquement (et pratiquement) pour optimiser leur utilisation comme la sécurité et la fiabilité. La disponibilité des ressources et l'incertitude sur les dates des évènements sont parmi les problèmes les plus critiques dans de telles plateformes. Dans le contexte du calcul volontaire, les ordinateurs (PC) se connectent aux grilles de calcul et offrent leurs ressources disponibles aux communités de calculs haute performance. Cette disponibilité peut cependant être altérée si, par exemple, certaines ressources tombent en pannes ou que des utilisateurs font des réservations à l'avance. Bien que plusieurs travaux se sont concentrés sur les problèmes d'incertitude des données dans les problèmes d'ordonnancement, il n'existe pas de travaux, à notre connaissance, qui traitent le problème d'incertitude sur les dates d'occurrence des périodes d'indisponibilité. C'est ce problème que nous attaquons dans le cadre de ce travail.

Dans ce chapitre, nous présentons les principaux travaux qui concernent aussi bien l'ordonnancement avec contraintes d'indisponibilité et l'ordonnancement avec incertitude sur les données. Ainsi, nous exposons dans une première section, les problèmes engendrés par les nouvelles plateformes de calcul distribué, en particulier les sources d'incertitude. Nous passons ensuite à la présentation des travaux relatifs à l'ordonnancement avec contrainte d'indisponibilité de ressources. Enfin, dans la section 2.2.4, nous passons en revue les techniques adoptées pour gérer l'incertitude dans les plateformes étudiées et nous donnons les mesures adoptées pour caractériser les performances dans ce type de plateformes.

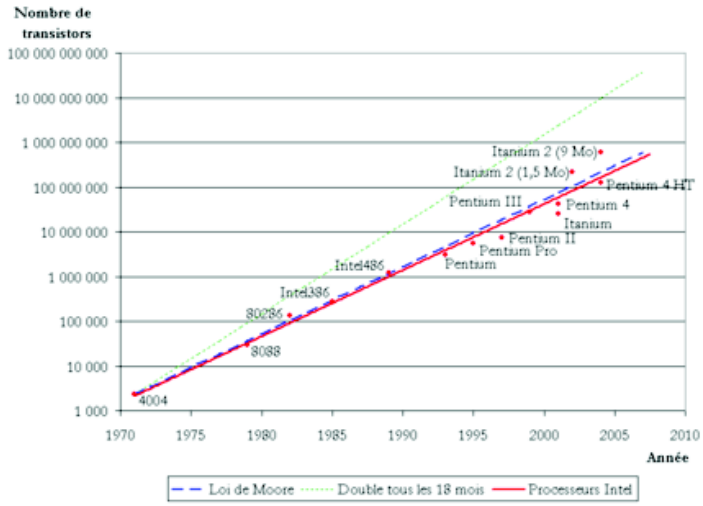


FIGURE 2.1 – Loi de Moore : Correspondance entre les valeurs théoriques et les valeurs réelles

2.2 Ordonnancement dans les nouvelles plateformes de calcul distribué

2.2.1 Nouvelles plateformes d'exécution de calcul distribué

Avec l'explosion que connait Internet depuis les années 2000, il est devenu possible de faire collaborer un nombre, parfois très grand, de machines

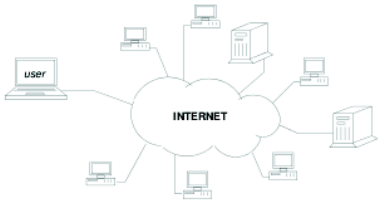


FIGURE 2.2 – Le calcul global : Collection de ressources individuelles connectées à travers Internet

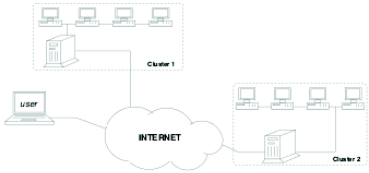


FIGURE 2.3 – Une grille de calcul : un ensemble de clusters liés à travers Internet

éparpillées partout dans le monde afin d'exécuter un calcul. Cette technique permet de construire un supercalculateur virtuel de dimension planétaire. On parle alors de Grille de calcul. Le coût de la mise en place d'une telle plateforme est peu couteu vis-à-vis des clusters et des machines parallèles. Cependant les grilles posent plusieurs problème de disponibilité, de fiabilité et de sécurité (du calcul et des données).

La motivation provient du fait que la moyenne mondiale estime que le nombre de cycles processeurs d'une machine de bureau qui ne sont utilisés est evalué à 75%. Cette moyenne est estimée à 47% en entreprise [1].

Actuellement, il existe deux approches pour la construction des grilles de calcul. La première est connue sous le nom de calcul Global (appelé aussi calcul volontaire) qui consiste à collecter un ensemble de ressources le plus souvent données par des volontaires (figure 2.2). Le projet de desktop grid le plus connu dans la communauté des grilles de calcul est SETI@HOME. La deuxième approche est plutôt institutionnelle et consiste à rassembler un ensemble de clusters reliés à travers Internet pour former une seule entité logique, offrant une capacité de calcul (ou de service) énorme (figure 2.3). Le projet français grid5000 [2] et européen EGEE [3] sont issus de cette méthode.

Du coté de l'architecture architecture, comme le montre la figure 2.4 il existe trois organisations possibles de grille :

- grille à architecture client/serveur : dans ce type de grille, les communications se font uniquement entre les ressources et un serveur centralisé. Aucune communication/collaboration ne se fait directement entre les

1. Source : Omni Consulting Group
2. www.grid5000.fr
3. www.eu-egee.org

ressources. C'est par exemple le cas dans le projet XtremWeb [35].

- grille à architecture centralisée : Dans ce type de grille, les unités de calcul se connectent à un seul serveur qui se charge de leur transmettre les données et de collecter le résultat du calcul. Cependant, au cours de l'exécution d'une application distribuée les ressources peuvent communiquer entre elles pour échanger des données. Ce type de grilles constituent ainsi une plateforme pour exécuter les modèles d'application avec échange de données. XtremewebCH [3], qui est une extension de XtremWeb avec prise en compte des communications, se classe dans cette catégorie.
- grille de calcul décentralisée : dans ce genre de grille, il n'existe aucun élément central. Les projets comme pastryGrid [40], ourGrid[20] et Vigne [46] sont des exemples de cette architecture. Le plus souvent, lors de l'exécution d'une application distribuée, les ressources (PCs) prennent des fonctions différentes (ordonnanceur, worker ....) mais, dès la fin de l'exécution tous les PC reprennent le même rôle au sein de la plateforme.

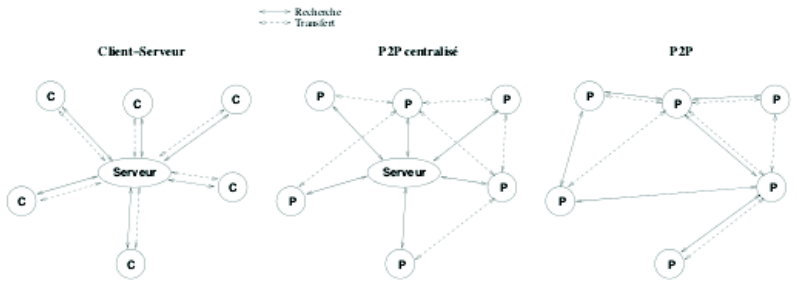


FIGURE 2.4 – Architectures possibles des grilles de calcul

Récemment, un nouveau type de plateforme est né : le cloud computing. La définition de cette plateforme est toujours une source de polémique dans la communauté du calcul distribué puisque aucun consensus n'a encore fait l'unanimité. Selon Boss et al [15], le cloud est un ensemble de ressources virtuelle. C'est donc un complément au grille de calcul dans lequel les ressources peuvent être gérées interactivement. Deux avantages principaux émanent de la virtualisation : la possibilité de faire passer une application à l'échelle en allouant de nouvelles ressources et la possibilité de faire un équilibrage de charge et une réallocation des tâches aux ressources d'une façon dynamique.

Des compagnies comme Google, Amazon, SUN, IBM et Oracle proposent des solutions pour utiliser le cloud pour pouvoir fournir aux utilisateurs encore plus services et d'applications (mail, application de traitement de texte, son, video, ...).

2.2.2 Étude de disponibilités de ressources dans les plateformes de calcul distribué

Les projets de calcul global reposent sur la fédération de ressources par des volontaires ou par des organisations via Internet. Le propriétaire de la ressource doit installer un composant logiciel sur sa machine cliente et faire une première configuration pour que sa machine puisse être utilisée par la plateforme de calcul parallèle. Ce logiciel va mesurer le taux d'utilisation de la ressource. Quand il détecte que l'activité du CPU est faible, il se charge de demander des travaux au serveur, d'exécuter les tâches et rendre le résultat.

Par exemple, le projet XtremWeb(CH) [3] comporte deux éléments essentiels : *le coordinateur* et *le worker*. Le coordinateur est le serveur qui va accepter les applications à exécuter, les dispatcher sur les workers et collecter les résultats. Le worker dans XtremeWeb(CH) est un logiciel qui s'installe sur la machine cliente et qui permet d'émettre un signal (appelé *work request* c.a.d demande de travail) toutes les 30 secondes au coordinateur quand la charge de la machine cliente est basse. Dans certaines autres plateformes, comme BOINC [8], il est possible de paramétrer les dates de début et de fin de la disponibilité d'une ressource. La ressource va donc se connecter et se déconnecter automatiquement du système aux dates indiquées.

Lorsqu'un volontaire installe le client BOINC, il accède au gestionnaire de configuration où il peut indiquer ces éventuelles préférences et/ou restriction de la ressource. En particulier, un onglet propre à l'utilisation du CPU est disponible et dans lequel, le volontaire peut indiquer les préférences requises. La figure 2.5 montre l'onglet préférence du CPU où on peut indiquer les restrictions d'utilisation du CPU. Ainsi, il est possible, entre autre, de définir une fenêtre temporelle dans laquelle le client peut exécuter du calcul fournit par l'un des projets de BOINC.

Une bonne utilisation des ressources dans de telles plateformes constitue un élément clé pour tirer le meilleur parti de la puissance de calcul offerte. Plusieurs travaux ont par conséquent été réalisés afin de caractériser au mieux le comportement de ressources volontaires et identifier des groupes de res-

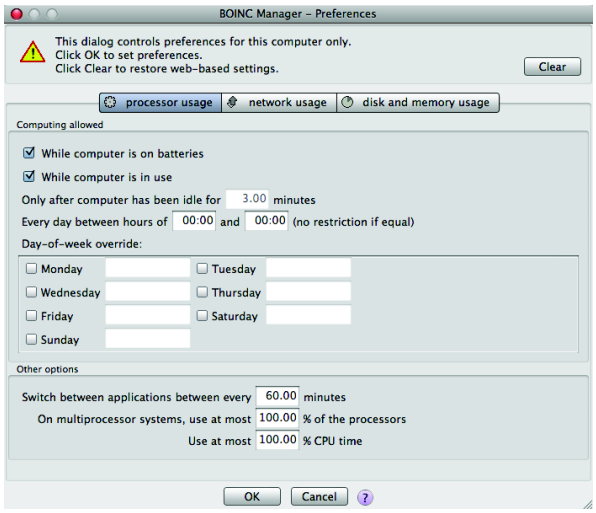


FIGURE 2.5 – Fenêtre de préférence de BOINC

sources selon des propriétés communes. Dans le reste de ce paragraphe, nous reportons les principaux travaux liés à caractérisation des ressources selon la disponibilité de leur CPU.

Les premiers travaux qui ont tenté de caractériser la disponibilité des ressources dans les systèmes pair-à-pair se sont focalisés sur la disponibilité des machines (hôtes), i.e, une valeur binaire qui indique si la machine est disponible (joignable) ou pas [11, 72]. Cette définition trouve rapidement ces limites avec le calcul global puisque une machine peut bien être disponible bien que son processeur soit occupé à 100%. Il n'est donc pas possible de l'utiliser pour le calcul global.

Plusieurs travaux ont essayé, par la suite, de caractériser la disponibilité de la plateforme. Cependant, les modèles produits caractérisent le système dans sa globalité et ne produisent pas une modélisation individuelle de la disponibilité des machines. Par exemple, dans [69] les auteurs ont montré que la disponibilité d'un système distribué peut être modélisée avec la loi de Weibull ou avec la loi hyper-exponentielle. Cependant, ils affirment que cette distribution ne représente pas convenablement la disponibilité individuelle des machines.

Récemment, Kondo et al. [53] ont produit les modèles qui caractérisent le mieux la disponibilité individuelle des processeurs dans les systèmes distribués à large échelle. Cette étude a été effectuée sur 230000 machines et a

aboutit aux conclusions suivantes : La disponibilité de 34% des processeurs qui se connectent au projet SETI@HOME suit une loi aléatoire et qu'il est possible de modéliser sa distribution d'une façon précise avec un nombre réduit de familles de distribution. Plus précisément, la disponibilité de ces processeurs a été modélisée avec 6 distributions et en particulier avec les lois Gamma , Weibull et log-normal. De plus, 51% de la disponibilité totale provient des processeurs qui suivent une distribution Gamma.

Dans le même contexte, kondo et al. [52] ont étudié un autre aspect important de la disponibilité qui consiste à chercher la corrélation entre la disponibilité des processeurs. Ce type d'information peut être d'une importance capitale lors de la prise de décision dans la phase d'ordonnancement. Savoir si un nombre assez important de machines est simultanément disponible peut aider à ordonnancer une application distribuée complexe. L'information sur la non corrélation entre la disponibilité peut par exemple être utile pour prendre une bonne décision quand à la duplication des tâches.

Les auteurs ont aussi fait leur étude en partant des traces collectées à partir de 110000 postes connectées à travers SETI@HOME. La conclusion du travail est qu'il est possible de grouper les machines qui manifestent un comportement identique vis-à-vis de la disponibilité. En utilisant la technique de clustering (groupement) les auteurs ont défini 5 groupes de motifs de disponibilité. Parmi ces groupes, il existe un groupe où les processeurs sont disponible de 90-100% du temps et un groupe où les processeurs ne sont presque pas disponibles.

Cette classification peut avoir des implications intéressantes sur la performance et la qualité des applications exécutée sur la grille de calcul (ou calcul global). A titre d'exemple, quand une application ne gère pas la migration (comme dans [30]) des tâches interrompues, elle peut favoriser le placement sur le groupe d'ordinateurs avec haute disponibilité.

Dans [71], une classification théorique qui identifie les motifs de disponibilités suivants :

- constant : toutes les machines sont disponibles
- zigzag : le nombre de processeurs disponibles est soit k soit $k-1$.
- croissant (décroissant) : le nombre de machines disponibles croit (décroit) au cours du temps.
- zigzag croissant : le nombre de machines disponibles croit au fils du temps mais ne décroit jamais de plus qu'une machine. Le motif inverse est appelé zigzag décroissant.
- escalier : dans ces motifs, si une machine i est disponible, alors, la

machine $i-1$ est aussi disponible.
- arbitraire : Dans le cas où le motif ne correspond à aucun des motifs cités ci-dessus, il est noté arbitraire.

2.2.3 Notation du problème d'ordonnancement sous contraintes d'indisponibilités

Le problème d'ordonnancement est d'une importance capitale dans le monde du calcul parallèle. Informellement, l'ordonnancement va spécifier la ressource sur laquelle une tâche va s'exécuter ainsi que la date de début d'exécution. De ce fait, la performance de tout système passe par une bonne utilisation des ressources disponibles, i.e, par la construction d'un *bon* ordonnancement.

Lorsque nous construisons un ordonnancement, nous devons respecter les contraintes imposées par l'environnement et nous cherchons à optimiser un critère de performance. Souvent, nous chercherons à minimiser la date de fin de la dernière tâche ou *le makespan* et noté C_{max}.

Dans [38], une notation standard pour le problème d'ordonnancement a été définit et qui se compose de trois champs $\alpha|\beta|\gamma$. Le champs α définit l'environnement d'exécution et est composé de trois sous-champs qui définissent respectivement, le type de l'environnement d'exécution (une machine (1), machines parallèles (P), machines uniformes (Q) ...), leurs nombres et le modèle de leurs disponibilités. Dans [71], huit modèles d'indisponibilité de machines ont été définie et qui permettent de renseigner sur les indisponibilités des machines.

Le champ β nous renseigne sur les tâches à exécuter (préemption, structure, durée, ...).

Finalement le champ γ indique le critère de performance à optimiser. Comme nous l'avons déjà signalé, souvent nous cherchons à optimiser le makespan noté C_{max}, mais, il est possible de spécifier d'autres critères comme la moyenne des temps de complétion $\sum C_j$, le retard maximal L_{max} ou encore le nombre de tâches en retard $\sum U_j$ où U_j vaut 1 si la tâche est en retard 0 sinon [16].

2.2.4 Ordonnancement avec contraintes d'indisponibilité de ressources

Contrairement aux configurations hors-lignes, où tout est connu à l'avance et que nous allons reporter ultérieurement, la littérature n'est pas très riche pour le cas d'une configuration en-ligne. Pour le cas d'une machine unique, kasap et al. ont étudié le cas où l'indisponibilité est due à une panne. Ils ont montré que quand la distribution des temps entre les différentes pannes est convexe, alors l'heuristique LPT est optimale pour le critère de minimisation du makespan [50]. Adiri et al. [5] ont étudié le problème de minimisation de la somme des temps de complétion des tâches et a proposé des algorithmes asymptotiquement optimaux quand les pannes suivent une distribution exponentielle. Notons que les travaux cités ci-dessus concernent une configuration non-préemptive puisque si une tâche est interrompue, elle devra être reprise (redémarrée) depuis le début.

Li et Cao [62] ont étudié le problème avec une configuration non-préemptive et ont proposé des solutions optimales pour minimiser les fonctions objectifs suivantes (1) la somme pondérée des temps de complétion des tâches (2) le nombre pondéré des tâches en retard ayant une date d'échéance constante (3) le nombre pondéré des tâches en retard ayant une date d'échéance aléatoire.

Pour le cas des machines parallèles, Lee et Yu [60] considèrent le cas où toutes les machines disparaissent simultanément et où la durée de l'indisponibilité est inconnue. Ils proposent des algorithmes pseudo-polynomiaux pour minimiser le problème de la somme pondérée des temps de complétion et ce aussi bien pour les cas préemptifs que non préemptifs.

Dans une politique avec lookahead, Albers et Schmidt [6] ont montré que le makespan optimal peut être obtenu par l'algorithme "le plus long reste d'abord". De plus, il est possible de ne pas utiliser le lookahead puisqu'il est possible d'approcher le makespan optimal aussi près que l'on veut et obtenir des algorithmes qui renvoient des makespan de valeur $OPT + \varepsilon$ avec $\varepsilon > 0$. Ces résultats ont été étendu dans [73] pour le cas des machines uniformes.

Récemment, des nouveaux résultats pour le problème de la somme des temps de complétion a été établie dans [26]. Les auteurs montrent que l'algorithme SRPT qui a chaque instant exécute les tâches qui ont le plus petit reste à exécuter minimise le makespan dans un motif de disponibilité zigzag croissant.

Finalement, le résultat suivant a été établit [74] :

Proposition 1. *aucun algorithme en ligne ne peut avoir un ratio de compétitivité meilleur que* $(2-\varepsilon)$ *pour* $\sum C_j$ *avec* $\varepsilon > 0$ *même si la longueur des tâches est* $\{1, 2\}$.

Passons maintenant à l'étude du problème dans une configuration hors-ligne. Nous commençons par présenter les résultats de complexité du problème. Nous présentons ensuite les principaux travaux qui traitent ces problèmes et qui peuvent être répartis en deux classes : Des algorithmes rapides (généralement basés sur LPT) avec des garanties de performance et des algorithmes moins rapides (schéma d'approximation polynomiaux) mais plus performant (vis-à-vis de la fonction objectif).

Les problèmes d'ordonnancement avec contraintes d'indisponibilité de machines est un problème NP-difficile puisque sa généralisation, qui est le problème classique d'ordonnancement est NP-difficile. Selon la notation à 3 champs introduite par [14], ce problème est noté $Pm|nr-a|C_{max}$. De plus, ce problème ne peut pas être approximé comme il a été prouvé dans [26]. La preuve de ce résultat est faite par réduction à partir du problème de partition [34]. De plus, dans [26], les auteurs énoncent le résultat suivant : Les problèmes $Pm|nr-a|C_{max}$ et $P|nr-a|C_{max}$ n'admettent pas d'algorithmes d'approximation polynomiaux avec ratio constant sauf si $P = NP$.

Ce résultat découle du fait de l'existence d'intervalles où aucune machine ne peut être disponible. Il est par conséquent possible de construire des contre-exemples où l'ordonnancement optimal tient avant le début des indisponibilité mais tout ordonnancement non optimal les place derrière les indisponibilités. Le makespan peut par conséquent être aussi mauvais que l'on souhaite.

De ce fait, il est important de trouver un ensemble de restrictions raisonnable qui vont nous permettre de concevoir des algorithmes avec ratio d'approximation constant. La restriction la plus utilisée dans la littérature [59, 26] est de considérer qu'il existe une machine qui soit toujours disponible, i.e, une machine sur laquelle aucune période d'indisponibilité ne peut être programmée. Dans le cadre du calcul distribué, cela correspond à un ensemble de machines fiables qui sont mises exclusivement à la disposition du projet et ne peuvent aucun cas être utilisée à d'autres fins. Avec cette restriction, le problème devient polynomial pour $m = 1$ mais reste NP-difficile au sens fort pour le cas où $m \geq 2$.

Proposition 2. *Le problème d'ordonnancement* $Pm, 1up|nr-a|C_{max}$ *n'admet pas de FPTAS quand* $m \geq 2$ *sauf si* $P = NP$.

La relaxation de contraintes dans ce problème ne le rend pas plus facile. En effet, ce problème demeure toujours NP-difficile au sens fort même quand le nombre maximal d'indisponibilités est limitée à 1. Il en découle par conséquence le résultat suivant :

Proposition 3. *le problème* $Pm, 1up|nr - a|C_{max}$ *n'admet pas de FPTAS même quand il y a au plus une seule réservation par machine pour* $m > 3$ *sauf si* $P = NP$.

Le cas le plus général pour ce problème a été étudié dans [25] et où, contrairement aux travaux cités précédemment, des schémas d'approximation polynomiaux ont été proposés qui utilisent des techniques issues du domaine de la recherche opérationnelle. En particuliers, les auteurs proposent des algorithmes basés sur la technique d'approximation duale et le problème du sac-à-dos multiple. Le problème avec un seul sac-à-dos a été étudié dans [45] et un FPTAS a été proposé et ensuite amélioré dans [57]. Dans la formulation du problème, les auteurs utilisent une famille particulière du problème où la taille des items est égale à leur profil. Ce problème est connu dans la littérature sous le nom subset set problem (SSP). La version avec plusieurs sac-à-dos, qui est utilisée ici, est désignée par MSSP. Un résultat d'inaproximabilité a d'abord été prouvé et qui conclut que le problème n'admet pas de FPTAS.

Un PTAS a ensuite été proposé et qui constitue le meilleur algorithme d'approximation possible. L'idée consiste à fixer un horizon pour l'ordonnancement et d'utiliser un MSSP pour sélectionner un ensemble de tâches à ordonnancer avant cet horizon. Ensuite, cet horizon est ajusté par le principe de la recherche dichotomique. Une version plus rapide a aussi été proposée qui consiste à utiliser un algorithme glouton à la place du MSSP, ce qui produit un algorithme d'approximation avec un ratio d'approximation de 1+m/2.

Notons que la plupart des travaux d'ordonnancement avec contraintes d'indisponibilité des ressources pour des tâches indépendantes sont basés sur l'analyse et l'adaptation de l'algorithme LPT (qui est un algorithme de liste qui trie les tâches par ordre décroissant des durée d'exécution), connu déjà pour être efficace.

Dans [58], Lee a étudié le problème d'ordonnancement avec date de début d'indisponibilité non-simultanées des ressources. Dans la pratique, ce problème peut correspondre à une situation ou nous ordonnançons par phase. A un instant t, on commence à calculer l'ordonnancement pour la phase actuelle tout en sachant que les tâches de la phase précédente vont terminer l'exécution

après l'instant t. La contribution principale de ce travail est de montrer que la performance de LPT est majorée par $\frac{3}{2}$ et de proposer un algorithme nommé mLPT (Modified LPT) dont la performance est majorée par $\frac{4}{3}$.

La généralisation de ce problème a été étudiée dans [59] et qui consiste à étudier l'ordonnancement des tâches indépendantes sous n'importe quel motif d'indisponibilité. L'auteur commence par montrer que ce problème est étudié NP-difficile au sens fort et qu'il ne peut pas être approché dans le cas général. Il est en effet possible de construire des instances pour lesquels seul un ordonnancement optimal donne un résultat acceptable. La performance de tout autre algorithme sera arbitrairement mauvaise. Lee a donc justifié l'adoption de l'hypothèse suggérant qu'une machine soit toujours disponible. Avec l'adoption de cette hypothèse et quand au plus une seule indisponibilité est autorisée sur chaque machine, la performance de l'algorithme LPT est majorée par $\frac{m+1}{2}$.

Dans [43], les auteurs ont attaqué le problème quand au plus la moitié des machines est autorisée à être indisponible simultanément et ont montré que la performance de LPT est majorée par 2 sous ces conditions. Ce résultat a été généralisé dans [44] comme suit : si au plus λ (compris entre 1 et $m-1$) sont autorisé à être indisponible simultanément, alors la performance de LPT est majorée par $1 + \frac{1}{2}\lceil\frac{m}{m-\lambda}\rceil$.

Liao et al. [63] ont abordé le problème avec $m = 2$ (deux processeurs/machines) et ou chaque machine possède une seule période d'indisponibilité. Ils ont proposé des algorithmes exponentiels qui donnent des solutions optimales et ont montré, à travers des expérimentations, que ces algorithme se comportent efficacement dans la pratique.

Le cas général du problème a été abordé dans [25] et où une machine est toujours disponible et un nombre quelconque de périodes d'indisponibilités (appelé réservations selon les auteurs) peut être programmé sur le reste des machines. Les auteurs montrent, d'abord, que ce problème n'admet pas de FPTAS et proposent ensuite un schéma d'approximation polynomial (PTAS) qui permet d'approcher la solution optimale.

Enfin, notons que tous les travaux cités ci-dessus sont relatifs aux tâches séquentielles. Eyraud et al. [31] ont attaqué le problème des tâches parallèles rigides. Ils ont montré que ce problème est inaproximable en un temps polynomial dans le cas général et ont procédé à l'étude de ce problème pour deux

cas restreints. Ainsi, ils ont proposé un algorithme d'approximation pour le motif des indisponibilités décroissantes (i.e, quand le nombre de machines disponibles augmente dans le temps). De plus, pour les cas des indisponibilité restreintes deux bornes inférieures et supérieures ont été établies pour borner la performance des algorithmes de listes.

2.2.5 Problèmes connexes

Bien que le problème de minimisation du makespan soit un critère très étudié dans la communauté du calcul parallèle vue qu'il représente la date à la quelle l'application parallèle termine l'exécution, certains autres critères, comme la somme des temps de complétion des tâches, ont intéressé les chercheurs. Ce critère prend son intérêt du fait qu'il représente le temps de complétion moyen pour l'exécution des tâches. Dans ce paragraphe, nous présentons certains travaux liés à ce critère. Les difficultés posées par ces problèmes et les techniques de résolution sont d'une utilité certaine pour résoudre les problèmes d'ordonnancement avec indisponibilité.

Dans [49], Kacem a étudie le problème de minimisation du makespan pour les tâches ayant une queue sur une machine avec une seule contrainte d'indisponibilité. Pour ce problème, les auteurs proposent un algorithme avec une borne atteinte de $\frac{3}{2}$. Cet algorithme construit itérativement un ensemble d'ordonnancement en sélectionnant adéquatement l'ensemble de tâches à ordonnancer avant le début de l'indisponibilité. Ces tâches sont triées selon la règle de Jackson(qui ordonne les tâches par ordre de queue). Cet algorithme est ensuite utilisé pour concevoir une FTPAS (schéma d'approximation complètement polynomial) au problème. Ce schéma est obtenu en réduisant l'espace de recherche en elliminant certains états de l'algorithme initial.

Le problème de minimisation de la somme des temps de complétion pour des machines parallèles avec contraintes de d'indisponibilité a été traité dans [67]. Dans ce travail, Mellouli et al. proposent plusieurs approches exactes pour résoudre ce problème dans un contexte non-préemtif. Notons d'abord que ce problème peut être résolu en un temps polynomial quand aucune contrainte d'indisponibilité n'est définie, et ce en utilisant la règle très connu SPT. Rappelons que SPT est une règle qui trie les tâches selon l'ordre croissant de durée [76]. Le problème a déjà été montré NP-difficile pour le cas d'une seule machine avec une contrainte d'indisponibilité unique et que toutes

les tâches ne peuvent pas être ordonnancées avant le début de l'indisponibilité.

Afin de résoudre ce problème, noté $Pm, h_{j1}|| \sum C_i$, Mellouli et al. proposent trois méthodes exactes : La première méthode consiste à utiliser la Programmation Linéaire en Nombre Entier (PLNE). Ainsi, ils considèrent que le problème revient à allouer un ensemble de tâches à $2m$ intervalles (qui sont les intervalles avant et après l'indisponibilité sur chaque machine). Ils conçoivent quatre variantes du programme linéaire dont la différence réside dans la façon avec laquelle le séquencement entre les tâches va être représentées au niveau des variables d'incidence.

La deuxième méthode repose sur l'utilisation de la programmation dynamique : l'horizon de l'ordonnancement est divisé en deux intervalles (le premier représente l'espace disponible avant l'indisponibilité alors que le deuxième intervalle représente l'espace disponible après l'indisponibilité). En se basant sur ce partitionnement, un programme dynamique est conçu et qui construit l'ordonnacement partiel pour les $k+1$ premières tâches en partant de l'ordonnancement partiel pour les k premières tâches.

Enfin, la troisième méthode que proposent les auteurs consiste à utiliser le paradigme (branch & bound) pour explorer toutes les solutions valides et garder la solution optimale. Pour cela, ils développement deux variantes. La première variante consiste à construire tous les ordonnancement partiels en affectant une tâche donnée à tous les intervalles de disponibilité existants. La solution optimale est ensuite gardée. La deuxième variante consiste à générer toutes les séquences possibles et de calculer l'ordonnancement conséquent en utilisant un ordre de numérotation de tâches et de machines.

2.2.6 Sources d'indisponibilité et d'incertitude

Les sources d'indisponibilité de ressources proviennent du propriétaire de la ressource et consistent généralement à la planification de la période de disponibilité ou de l'indisponibilité. Nous avons vu dans le paragraphe 2.2.2, les clients BOINC offrent une interface d'administration qui permet de spécifier les dates entre lesquelles l'utilisateur autorise l'utilisation de sa ressource. Par ailleurs, dans la grille française grid'5000, l'ordonnanceur permet de faire des réservations de ressources qui deviennent alors indisponibles vis-à-vis des autres utilisateurs de la grille. Les tâches soumises par ces derniers devront alors être exécutées hors ces périodes d'indisponibilités.

La planification de maintenance des ressources est aussi une source d'in-

disponibilités des ressources. Comme pour les deux cas précédents, ces périodes sont connues à l'avance et il est possible de les prendre en considération lors de la phase de l'ordonnancement.

De plus, dans certaines grilles de calcul, comme grid'5000, il est possible d'effectuer des réservations de ressources. Ces ressources sont par conséquent vues comme étant indisponibles de point de vu de l'ordonnanceur (OAR). Si d'autres tâches sont soumises à la grille, elles devront être exécutées hors les dates de réservation, i.e, quand les ressources sont disponibles.

Malgré la planification de ces évènements lors de la phase d'exécution des évènements externes peuvent provoquer des perturbations aux dates prévues. Les indisponibilités vont par conséquence commencer/se terminer avant/après les dates initialement prévues. Une telle situation risque d'avoir un impact sur l'exécution des tâches et plus généralement sur la qualité de la solution (ordonnancement).

Par exemple, des pannes peuvent affecter la ressource elle même ou l'un des composants intermédiaires entre la ressource et le serveur (élément central), typiquement le réseau de communication. Ces pannes ont pour effet de rendre la ressource indisponible éventuellement plus longtemps que les dates annoncées a priori. Quand les dates des pannes sont inconnues, le problème fait partie du domaine de la tolérance aux pannes. D'autre part, comme les grilles de calculs sont basées sur la fédération de ressources, le comportement du propriétaire peut contribuer à la déstabilisation de la disponibilité des ressources. A cet effet, un chercheur qui prévoit de donner sa machine à un projet, tous les jours entre 20h et 9h du matin, peut dans une période chargée être indisponible plutôt que prévu, à 7h par exemple, pour travailler. Dès qu'il reprend sa machine, il peut interrompre une tâche qui a été affectée à sa machine et dont la date de fin d'exécution est prévue pour 8h40. Cette tâche va devoir donc être redémarrée ultérieurement.

Il est évident que si de telles situations se produisent régulièrement, elles vont affecter la performance du système, vu que le temps disponible pour le calcul sera alloué à d'autres tâches et ainsi, l'exécution ne va jamais se terminer. Il est donc indispensable de concevoir des algorithmes qui prennent en compte ces aléas et qui permettent d'optimiser l'utilisation des ressources.

2.3 Vivre avec les incertitudes

2.3.1 Motivation

Les environnements d'exécution parallèles se sont beaucoup diversifiés pendant la dernière décennie et sont devenus de plus en plus populaire et versatile. Les plateformes de calcul global (PCG) ont été conçues pour être une alternative aux plateformes de calculs dédiées comme les grappes et les machines parallèles. L'avantage premier de ces plateformes est qu'elles peuvent être mise en œuvre à des coûts très intéressants. L'idée est de profiter des cycles d'inactivité des ressources disponibles pour exécuter du calcul.

Les ressources sont mises à la disposition du système par des personnes volontaires se situant à des endroits distants géographiquement et reliés par réseau, et souvent par Internet. Cette solution, bien que peu coûteuse, engendre un certain nombre de problèmes qu'il faut résoudre théoriquement. Ainsi, la sécurité, la volatilité des ressources, la certification des résultats, la fiabilité des connexions et le placement des tâches sur les ressources sont des problèmes auxquels il faut trouver des solutions aussi bien théoriques que pratiques.

L'incertitude sur les données des paramètres de la grille est l'une des caractéristiques intrinsèques à ces plateformes. Particulièrement, les incertitudes sur les durées de communications et sur les dates de début et de fin des indisponibilités des ressources sont des caractéristiques intrinsèques aux PCG. Ce phénomène aura un impact direct sur la performance de tout le système puisque sans la prise en compte de ces aléas, nous risquons de gaspiller les ressources disponibles. Du point de vue de l'utilisateur, ces phénomènes peuvent dégrader la qualité du service attendu.

Dans cette section, nous présentons les sources qui peuvent causer les incertitudes, nous détaillons les méthodes de résolution des problèmes d'ordonnancement avec incertitudes et nous définissions les mesures de performances qui qualifient la qualité d'une solution dans ce domaine.

2.3.2 Les Sources d'incertitude dans les problèmes d'ordonnancement

Les données associées à un problème d'ordonnancement sont les durées des tâches, les dates d'occurrence de certains évènements, certaines caractéristiques structurelles, et les coûts de l'exécution des tâches (dans certains modèles

économiques). Aucune de ces données n'est exempte de facteurs d'incertitudes. Les durées d'exécution des tâches dépendent des conditions de leurs exécutions, et en particulier des ressources humaines et matérielles nécessaires. Elles sont donc incertaines par nature, indépendamment des aléas pouvant perturber leur exécution. Les éventuelles durées de communications entre deux tâches informatiques dépendant de l'état du réseau de communication, du taux d'utilisation, de la disponibilité des liens, etc.

Les évènements qui structurent un ordonnancement surviennent à des dates qui pour certaines sont des données initiales. C'est le cas de l'arrivée d'une tâche (date de disponibilité) : elle dépend souvent d'évènements extérieurs au système étudié. Il en est de même de la date due d'une tâche. Les périodes de disponibilité des ressources (humaines, machines) sont parfois difficiles à prévoir exactement, pour cause de retard ou tout autre évènement critique.

Plus radicalement certains évènements peuvent survenir sans avoir été prévus et modifier la structure du problème, donc l'ordonnancement en cours. Une tâche peut être rajoutée sans préavis, ou à l'inverse supprimée. Les caractéristiques d'une tâches peuvent être modifiées, comme son mode d'exécution (gamme de fabrication, opérateur ou ordinateur imposé) ou ses liens avec les autres tâches, comme les précédences, disjonctions, etc. une machine peut tomber en panne ou du moins être une inutilisable subitement pour des raisons imprévues.

Dans le contexte de notre étude, les ressources qui constituent la PCG sont formés en majorité par des ordinateurs anonymes sur lesquels l'administrateur de la grille n'a pas des droits d'administration. De plus, même quand les ressources appartiennent à des personnes de confiance, les propriétaires des ressources peuvent les connecter ou déconnecter quand ils veulent. En effet, quand une ressources est connectée à une PCG, elle reçoit des tâches à exécuter et qui occupent le processeur. Le propriétaire, ayant besoin de sa machine, peut décider de déconnecter sa machine même si celle-ci est en train d'exécuter une tâche. Celles-ci va devoir donc être reprise, éventuellement sur une autre ressource (en fonction de la politique de l'ordonnanceur).

Ainsi, aucune donnée ne peut être considérée comme immuable, même si la possibilité d'un changement dépend du contexte. Nous pouvons considérer deux cas pour chaque donnée de type durée, date ou coût : soit sa valeur est incertaine, et la donnée peut prendre un grand nombre de valeurs sans que l'une d'elles soit privilégiée ; soit sa valeur peut être sujette à un aléa, c'est -à-dire qu'elle est fixée pour un fonctionnement normal du système mais peut

être modifiée par un évènement imprévu.

2.3.3 Modèles de l'incertitude

Les modèles non déterministes sont indispensables pour résoudre des problèmes d'ordonnancement avec incertitude sur les données. L'hypothèse aléatoire conduit à l'ordonnancement stochastique. Dans cette hypothèse, toutes les données sont modélisées grâce à des variables aléatoires, éventuellement constantes : les durées, mais aussi les dates d'occurrences des évènements, y compris de certains aléas. La probabilité d'occurrence de ces aléas est supposée connue. A partir de ce modèle stochastique, il est en théorie possible de calculer a priori les ordonnancements les plus performants, ou plutôt (en cas d'aléas possibles) les politiques, c'est-à-dire les décisions les plus performantes [18, 70].

Souvent, cette hypothèse aléatoire n'est pas retenue, pour plusieurs raisons dont nous citons : (1) D'abord la connaissance a priori sur les données n'est pas toujours suffisante pour déduire les lois de probabilité associées, surtout si le problème posé est traité pour la première fois (2) les hypothèses d'indépendance sont rarement justifiées, une source principale de perturbations entraînant souvent des incertitudes sur plusieurs données (3) Le modèle stochastique est souvent trop complexe pour être utilisable.

La valeur d'une donnée est alors considérée comme incertaine. Cependant, il est généralement possible de maintenir cette valeur dans des limites, i.e, elle est *presque sûrement* à l'intérieur d'un ensemble, discret ou continu (un intervalle). Dans le cas d'ensembles discrets, on obtient un nombre fini, mais potentiellement important de scénarios, où une valeur est affectée à chaque donnée. Il peut arriver que les données sortent malgré tout de l'ensemble proposé. Aussi, une solution simple consiste-t-elle à ne proposer aucun ensemble. Malgré tout, une hypothèse fréquente est d'attribuer à chaque donnée une valeur centrale : son estimation. Nous travaillons donc sur ces valeurs moyennes tout en anticipant sur les différences possibles à l'exécution. Il reste enfin la dernière possibilité, quand aucune estimation n'est disponible.

En résumé, les données peuvent être représentées par des variables aléatoires (modèle stochastique), des intervalles réels (modèle par intervalles), des ensembles discrets (modèle des scénarios). De plus, on peut leur associer ou non une estimation initiale.

2.3.4 Résolution du problème d'ordonnancement avec incertitude

Approches de résolutions

Plusieurs approches [81, 13, 66, 23, 39, 80] ont été proposées pour traiter le problème d'ordonnancement avec incertitudes sur les données. La différence principale entre les différentes approches réside dans l'instant pendant lequel l'ordonnancement est totalement calculé. La résolution complète se fait selon trois étapes :

Étape 0 (modélisation) : Définition du problème statique. Cette définition comprend, en plus des spécifications classiques en ordonnancement déterministe, la spécification des incertitudes et leur modélisation. La notion de qualité d'un ordonnancement doit être également précisé à cette étape.

Étape 1 : Calcul d'un ensemble de solutions, d'une famille d'ordonnancements réalisables, par un algorithme statique (phase statique).

Étape 2 : Lors de l'exécution, calcul d'une solution unique, c.a.d de l'ordonnancement réalisé, issu de cet ensemble par un algorithme dynamique (phase dynamique).

Les approches de résolution diffèrent suivant les choix effectués durant les étapes 1 et 2, donc suivant les algorithmes statiques et dynamiques choisis. Ces choix dépendent bien sûr des modèles de *l'étape 0*.

Approche proactive

Dans cette approche, *l'étape 1* est privilégiée : la connaissance de l'incertitude est utilisée par l'algorithme statique pour construire un seul ordonnancement noté ordonnancement de référence ou une famille d'ordonnancements qui est décrite de manière explicite ou implicite. Cette approche est nommée aussi approche prédictive dans la littérature [24].

Durant la seconde phase de cette approche l'exécution ne demande aucun calcul : suivant la valeur réelle des données, un des ordonnancements calculés est utilisé, ou bien l'ordonnancement de référence est ajusté pour rester réalisable. Ces choix ou ajustements se font grâce à des règles simples, comme par exemple attendre la tâche prévue si elle est en retard, ou prendre telle décision si tel événement survient.

Approche proactive/réactive

Quand, dans l'approche proactive, nous utilisons un algorithme dynamique sophistiqué pour le choix de l'algorithme à utiliser dans la phase 2, nous appelons cette approche proactive/réactive. De plus, il est souvent impossible de prendre en compte toutes les incertitudes et en particulier les aléas, durant la phase statique. L'exemple des pannes des machines est le plus évidant, mais ce n'est pas le seul. L'algorithme dynamique est alors une nécessité, y compris lorsqu'un ordonnancement de référence est le seul proposé en statique [66, 7].

L'algorithme dynamique peut, soit faire des réparations sur l'algorithme qui a déjà été pré-calculé pour que ce dernier reste réalisable (i.e puisse s'exécuter), soit faire une ré-optimisation au fur et à mesure que les valeurs des données du problème définitivement fixés.

Approche réactive

Dans l'approche réactive pure, les choix spécifiant un ordonnancement sont effectués durant la phase dynamique[80]. Le temps de réaction demandé peut aller de plusieurs jours pour certains projets à moins d'une seconde pour des applications informatiques ou des systèmes embarqués. Si nous disposons d'informations assez précises sur la valeur des données concernant la suite de l'exécution, l'idéal est de calculer à chaque instant de décision la décision optimale. Mais le plus souvent des règles de décisions simples sont appliquées.

2.3.5 Construction d'algorithmes d'ordonnancement flexible

Dans une approche d'ordonnancement flexible, des degrés de libertés sont offerts a priori pour construire un ordonnancement. Lors de la phase 2, nous pouvons alors tirer partie des libertés offertes. La flexibilité porte peut porter sur :

- *Flexibilité sur les temps, ou flexibilité temporaire, c'est à dire sur les dates de début d'exécution des opérations :* cette flexibilité est assez implicite en ordonnancement, puisqu'elle consiste à autoriser certaines opérations à dériver dans le temps, si les conditions l'imposent. Cette flexibilité est donc la base incontournable de la flexibilité en ordonnancement.

- *Flexibilité sur les ordres ou séquentielles :* nous autorisons dans ce cadre, à modifier l'ordre dans lequel les opérations doivent s'exécuter sur des machines. Implicitement, cela suppose une flexibilité temporelle. Elle peut être proposée lors de l'exploitation de la séquence, en autorisant certaines opérations à en doubler d'autres, si les conditions l'exigent.
- *Flexibilité sur les affectations :* dans le cas où les ressources existent en plusieurs exemplaires, nous autorisons l'exécution d'une tâche à s'effectuer sur une autre ressource que celle qui était prévue initialement. Cette flexibilité est d'un grand secours lorsque par exemple une machine devient inutilisable. Elle suppose implicitement une flexibilité séquentielle et une flexibilité temporelle.
- *Flexibilité sur les modes d'exécution :* Le mode d'exécution comprend l'autorisation ou non de la préemption, du chevauchement, le changement de gamme, la prise en compte ou non de temps de préparation, la modification de nombre de ressources nécessaires pour exécuter une opération, etc. Cette flexibilité peut être proposée suivant le contexte, pour pallier une situation difficile.

La flexibilité, qui est un degré de liberté offert durant la phase d'exploitation, peut être exploitée lors de l'étape 1, i.e, dans la phase statique.

2.3.6 Mesure de performance

Mesure de stabilité d'un ordonnancement

La mesure de stabilité consiste à mesurer les écarts entre les solutions obtenues suivant les différents scénarios. Dans certains ouvrages, cette mesure est appelée de robustesse de la solution (solution robustness)[65, 12]. Un ordonnancement est dit stable si la solution qu'il produit reste inchangée dans les différents scénarios. Nous pouvons adopter plusieurs mesures pour caractériser l'écart entre deux ordonnancements comme le nombre de permutations entre tâches ou entre machines, ou toute autre mesure pertinente dans le contexte considéré. Le but est soit de minimiser le plus grand écart entre deux solutions, soit le plus grand écart par rapport à un ordonnancement de référence σ. Dans le second cas, nous parlons donc de la stabilité de cet ordonnancement de référence. Notons d la distance entre deux ordonnancement. Analytiquement, la stabilité est définie comme suit :

$$max_I \quad d(S(\sigma_I), S(\sigma))$$

où I est une instance du problème, σ_I est l'ordonnancement calculé pour l'instance I et S et une mesure de performance.

En d'autres termes, nous cherchons à construire le plus petit ensemble possible d'ordonnancement compatible avec l'incertitude prise en compte. C'est donc la recherche d'un ensemble de flexibilité minimum, mais suffisante pour garantir l'exécution. En pratique bien sûr, nous partons d'un ordonnancement de référence dont les performances sont acceptables, voire très bonne, pour les estimations disponibles.

Stabilité relativement au critère de performance

Dans la pratique, lors de la construction d'un ordonnancement nous souhaitons optimiser un critère de performance. Généralement, l'étude de la stabilité porte à minimiser l'écart des valeurs entre les solutions obtenues pour les différents scénarios. Si l'on considère la valeur du critère comme une caractéristique parmi d'autre d'un ordonnancement, nous voyons qu'il s'agit d'une stabilité particulière : nous cherchons à construire un ensemble d'ordonnancements aux valeurs voisines. On parle alors de robustesse de la qualité d'un ordonnancement (quality robustness). Le plus souvent cette approche est utilisée conjointement avec un modèle basé sur des estimations des données. Il existe un ordonnancement de référence σ, et la mesure de robustesse est donné par le plus grand écart entre la valeur de cet ordonnancement pour le scénario initial $S(\sigma)$ et la valeur obtenue par la méthode pour tout autre scénario potentiel (noté $\tilde{I}$). Le rapport de stabilité est alors définit comme suit :

$$max_{\tilde{I}}$$

I et $\tilde{I}$ sont des instances possible du problème. Chacune a une solution optimale et une solution produite par un algorithme d'ordonnancement particulier. La stabilité est mesurée comme le rapport entre ces deux dernières valeurs (voir figure 2.6).

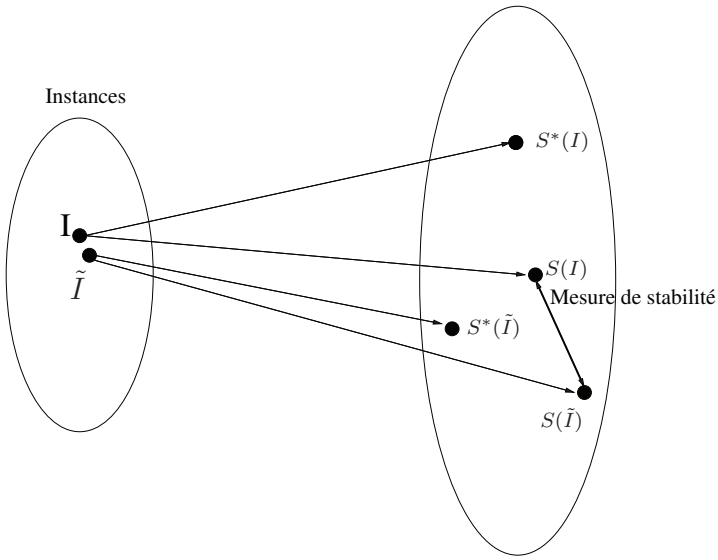


FIGURE 2.6 – La stabilité mesure la distance entre la performance d'un algorithme pour deux scénarios

2.4 Stabilisation des ordonnancements grâce aux tampons

Après avoir présenté les différentes techniques de construction d'algorithmes dans les environnements incertains, nous nous intéressons dans cette section à certains travaux sur l'ordonnancement avec ressources parallèles [55, 56, 32].

Dans [55] les auteurs étudient le problème d'ordonnancement d'un workload représenté par un graphe de tâches sur des processeurs parallèles. La durée des tâches et les précédences entre elles sont connues à l'avance.

La plateforme d'exécution est formée par un ensemble de machines (processeurs) ayant chacune une disponibilité, mais aussi susceptibles à avoir des pannes dont la date d'occurrence et la durée de réparation sont aléatoires.

Afin de résoudre ce problème, les auteurs proposent deux approches : proactive et réactive.

La solution proactive consiste à construire un ordonnancement de référence puis d'anticiper l'effet des pannes en adoptant un mécanisme de tampon sur les machines ou sur les tâches.

Pour atteindre cet objectif, il est possible de modéliser la durée totale des pannes sous formes de loi de poissons, et ce, en la modélisant comme la somme des pannes issues de tous les facteurs possibles. Les auteurs supposent que si les facteurs des pannes suivent une loi exponentielle et que ces lois sont indépendantes, alors la somme de ces durées suit une loi de poisson [41]. Comme le processus de poisson correspond à une distribution exponentielle des temps d'arrivée, les temps entre les pannes seront distribués exponentiellement [36].

Cependant, cette hypothèse ne peut pas être réaliste pour les durée de réparation (durée des pannes), mais, pour leurs travaux, les auteurs jugent acceptables sur les plans théoriques et pratiques de considérer que ces dates suivent aussi une loi exponentielle.

Au lieu/en plus du mécanisme du tampon sur les ressources, les auteurs définissent le tampon sur les tâches. Ce mécanisme consiste à ajouter un tampon devant le début de chaque tâche afin de retarder son début d'exécution aussi tard que possible. Ce retard va permettre d'absorber les perturbations dues aux pannes qui ont lieu plutôt et ont causé un décalage d'exécution de tâches. Le processus consiste à calculer un ordonnancement réalisable et de décaler les dates de début d'exécution des tâches à droite (c.a.d retarder) itérativement la date de début d'exécution afin de les protéger le mieux possible et ce sans dépasser la date due (d'échéance).

Bien que les deux algorithmes présentés ci-dessus représentent une approche proactive pour construire un ordonnancement peu sensible aux aléas produits lors de la phase d'exécution, ils peuvent rendre l'ordonnancement non faisable malgré les mécanismes de tampon envisagées. Des stratégies réactives ont alors été conçues par les auteurs et qui permettent de corriger le mieux possible la déviation due aux pannes qui se produisent. La première stratégie proposée est basée sur la technique de liste. Ainsi, quand une panne se produit une liste de tâches non encore exécutée est construite. Les tâches de cette liste sont triées par ordre de dates de début d'exécution de l'ordonnancement de référence. Cette liste est ensuite considérée pour continuer l'exécution après la reprise des machines défaillantes.

Cette solution est ensuite améliorée par l'utilisation d'un mécanisme basé sur la recherche tabou [37]. Les auteurs proposent en effet de faire une série de permutations entre les tâches adjacentes, qui ne suivent pas un ordre tabou. Le but est de trouver un ordonnancement faisable qui dévie le moins possible par rapport à l'ordonnancement de référence. La déviation est mesurée comme la somme pondérée de la déviation absolue entre la date de

début des tâches dans l'ordonnancement de référence et dans l'ordonnancement perturbé.

Dans [32], la solution proactive décrite ci-dessus a été revisitée et les auteurs ont proposé plus de choix pour la construction d'une solution proactive. Afin d'absorber les effets des perturbations dues aux incertitudes, les auteurs proposent pour la construction de l'ordonnancement de référence, en plus du coût d'instabilité cumulatif, l'utilisation de deux autres méthodes à savoir le temps-ressource et la méthode de la ressource critique. La première méthode consiste à calculer la priorité en combinant l'utilisation des ressources, la durée des tâches et le facteur d'instabilité de la tâche. Dans la deuxième approche, c'est la disponibilité de la ressource qui est considérée et les tâches seront triées de façon qu'il n'y ait pas de situation critique.

Ces trois algorithmes ont par la suite été utilisés pour la conception d'un algorithme proactif qui vise à absorber les effets des perturbations si celles-ci se produisent au cours de la phase d'exécution. L'ordre initial de considération des tâches est calculé en utilisant l'une des trois stratégies. Par la suite, un tampon sera ajouté aux ressources et/ou aux tâches afin de limiter les effets de pannes des ressources dans un contexte préemptif (contrairement à [55]).

Les expérimentations reportées la fin de ce travail montrent que la double protection de l'ordonnancement à travers des tampons sur les ressources et sur les tâches permet d'obtenir les meilleurs résultats. En plus, l'utilisation de la méthode de ressources critiques pour construire l'ordonnancement initial donne un meilleur résultat qu'avec les deux autres méthodes.

2.5 Définition du problème étudié

2.5.1 Définition du workload

Nous étudions dans le cadre de ce travail, l'ordonnancement d'un ensemble T de n tâches qui seront indexées par j. Chaque tâche d'indice j a un coût p_j exprimé en nombre de FLOP. La durée de la tâche dépend par conséquent du processeur sur lequel elle sera exécutée.

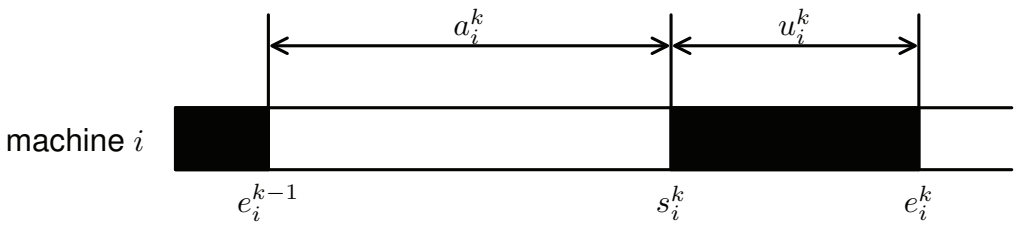


FIGURE 2.7 – Représentation de l'intervalle k sur la machine i, qui contient une période de disponibilité suivie d'une période d'indisponibilité qui commencent aux dates e_i^{k-1} (date de fin de l'intervalle $k-1$) et s_i^k. Les machines entièrement disponibles ne contiennent aucune indisponibilité

2.5.2 Définition de la plateforme d'exécution

L'ensemble de tâches définies ci-dessus sera exécuté sur un ensemble P de m processeurs (appelés aussi machines) indexées par i. Le cycle d'une machine i est noté b_i. Quand une tâche j est placée sur une machine i, sa durée d'exécution sera $p_j b_i$.

Chaque processeur présente de plus un ensemble d'indisponibilités, qui sont des intervalles de temps, connus à l'avance, pendant lesquels il n'est pas possible d'utiliser le processeur. Nous considérerons dans la suite qu'un intervalle est formé par une disponibilité suivie par une indisponibilité. Les intervalles sont indexé par k. La date de début de la k^{ieme} indisponibilité est s_i^k et la durée est u_i^k.

La durée de la plus petite disponibilité est notée $a_{\min}$ alors que la plus grande indisponibilité est notée $u_{\max}$. Le plus grand rapport entre la durée d'une indisponibilité et la disponibilité qui la précède définit le ratio d'indisponibilité que nous notons γ.

La figure 2.7 résume l'ensemble de ces notations.

2.5.3 Définition de la solution (ordonnancement) et des mesures

Pour résoudre le problème d'ordonnancement avec contraintes d'indisponibilités et d'incertitudes, nous proposons une solution à deux phases. La première phase consiste à calculer un ordonnancement avec les données déterministes (sans prendre en compte les perturbations) et la deuxième

consiste à adapter cet ordonnancement aux éventuelles perturbations.

Un ordonnancement ρ est une application de l'ensemble des tâches T vers $PX\mathbb{R}$ qui définit pour chaque tâche le processeur sur lequel elle va être exécutée et la date de début d'exécution.

$$\rho : T \longrightarrow PX\mathbb{R}$$

Pour chaque tâche j, $\rho(j) = (i, t)$ signifie que la tâche j est ordonnancée sur le processeur i et commence à être exécutée à la date t. La date de fin de la tâche est notée $C_j = t + p_j$.

Pour les besoins du chapitre 5, nous définissons la fonction $\pi(i, k)$ qui renvoie toutes les tâches affectées à la k^{ieme} disponibilité du processeur i. D'autre part, M_i^k et S_i^k dénotent respectivement la plus longue durée, et la somme des durées des tâches allouées à l'intervalle k du processeur i.

Finalement, nous notons $C_{max}^{alg} = max_{j \in T} C_j$ le makespan d'un ordonnancement produit avec un algorithme *alg*. Le makespan optimal d'une instance est noté C_{max}^*. Dans le cas d'ambiguïté, nous indiquons l'instance en question entre parenthèses (exemple, $C_{max}^*(I)$ désigne le makespan optimal d'une instance I).

2.5.4 Modèle de perturbation

Au cours de l'étude des instances perturbée, nous convenons de noter les paramètres sujets aux perturbation par le même nom surmonté d'une *tilde*. Si par exemple x dénote un paramètre, $\tilde{x}$ dénote sa valeur perturbée.

Nous allons dans le cadre de ce travail étudier plusieurs modèles de perturbations. Dans le chapitre 3, L'étude est restreinte à un seul intervalle d'indisponibilité. La perturbation touche alors la date de début de l'intervalle et sa durée qui seront perturbées. L'amplitude de la perturbation est notée δ_i (Il n'est pas nécessaire d'indexer les intervalles dans ce chapitre puisqu'il en existe un seul). La date de début perturbée est par conséquence $\tilde{s_i} = s_i + \delta_i$ avec $-s_i \leq \delta_i \leq 0$.

Dans le chapitre 4, nous généralisons l'étude effectuée pour un nombre quelconque d'indisponibilités. La perturbation portera sur la date de début de l'indisponibilité, mais la durée restera constante. Nous notons $\tilde{s_i^k} = s_i^k + \delta_i^k$ avec $-a_i^k \leq \delta_i \leq 0$. Dans ces deux chapitres, les indisponibilités ne peuvent qu'avancer (se produire avant la date prévue).

Enfin, dans le chapitre5 l'étude est étendue au cas où les perturbations peuvent avoir lieu aussi bien avant où après la date prévue. L'amplitude de

la perturbation δ_i^k porte sur la date de début d'indisponibilité mais sa durée reste fixe. Ainsi, nous notons $\tilde{s_i^k} = s_i^k + \delta_i^k$ avec $-a_i^k \leq \delta_i \leq -a_i^{k+1}$ et $\tilde{u}_i^k = u_i^k$. Les indisponibilités peuvent donc avancer ou reculer.

2.6 Tableau récapitulatif

Nous récapitulons dans le tableau qui suit, les différentes notations introduites ci-dessus.

2.7 Conclusion

L'émergence des nouvelles plateformes de calcul distribué a apporté une puissance de calcul sans précédent mais aussi un ensemble de problèmes lié à la nature distribuée de ces plateformes. En particulier, la non-disponibilité et l'incertitude sur certains paramètres des ressources posent un défi important pour la construction d'ordonnancements robustes. Ce besoin émerge du fait que les évènements imprévus qui se produisent durant l'exécution des tâches peuvent avoir un effet néfaste sur la qualité de l'ordonnancement qui peut être au pire des cas, arbitrairement mauvaise.

Afin de s'adapter à ces nouvelles plateformes, plusieurs études ont été réalisées et qui traitent les aspects que nous avions indiqués. Dans ce chapitre nous avons reporté plusieurs travaux qui se sont intéressés à l'ordonnancement avec contraintes d'indisponibilité de ressources.

Nous avons ensuite présenté un état de l'art sur l'ordonnancement dans des environnements incertains. Ce type de problèmes nécessite de nouvelles approches de résolution et aussi de nouveaux critères pour l'évaluation de la qualité de la solution générée (ordonnancement). Le but n'étant plus la génération d'ordonnancement performant par rapport aux fonctions objectifs classiques (makespan, somme de temps de complétion des tâches, retard maximal .....), mais, la génération d'ordonnancement robustes, i.e, qui résistent aux aléas.

Dans la dernière partie de ce chapitre, nous avons étudié certains travaux liés à l'utilisation des tampons, qui sont des intervalles de temps pendant lesquels aucune tâche n'est exécutée. La suite de notre travail est basée sur une idée similaire. Les travaux ont conclu que l'adoption de telles techniques conduit à des performances satisfaisantes.

Symbole	Définition
i	index des processeurs
j	index des tâches
k	index des intervalles
m	nombre de processeurs
n	nombre de tâches
b_i	cycle de la machine i
p_j	coût de la tâche j
a_i^k	durée de la disponibilité k sur le processeur i
u_i^k	durée de l'indisponibilité k sur le processeur i
s_i^k	date de début de l'indisponibilité k sur le processeur i
e_i^k	date de fin de l'indisponibilité k sur le processeur i
$p_{\max}$	durée maximale d'exécution des tâches ($p_{\max} = \max_{i\in[1..m],j\in[1..n]} b_i p_j$)
$a_{\min}$	durée de la plus petite disponibilité
$u_{\max}$	durée de la plus longue indisponibilité
γ	ratio d'indisponibilité, $\gamma = \max_i \left(\max_k \frac{u_i^k}{a_i^k}\right)$
δ_i^k	perturbation sur la date de début de l'indisponibilité s_i^k
$\widetilde{s}_i^k$	date de début perturbée de l'indisponibilité k sur le processeur i ($\widetilde{s}_i^k = s_i^k + \delta_i^k$)
$\pi(i,k)$	ensemble de tâches allouées à l'indisponibilité k sur le processeur i
$C_{\max}$	makespan
λ	horizon
σ	stabilité
d_i^k	tampon qui précède l'indisponibilité k sur le processeur i
β	paramètre de la règle de tampon ($\beta \in [0,1]$)
M_i^k	plus grande taille de tâche allouée à la disponibilité k sur le processeur i ($M_i^k = \max_{j\in\pi(i,k)} b_i p_j$)
S_i^k	somme des tailles des tâches allouées à la disponibilité k sur le processeur i ($S_i^k = \sum_{j\in\pi(i,k)} b_i p_j$)

Table 2.1 – Résumé des notations

Bien que plusieurs travaux ont mis l'accent sur l'incertitude de la disponibilité des ressources, la plupart d'entre elles considère que ces périodes sont dues à des maintenances prévues (ou à des réservations) ou à des pannes. La première branche d'étude est classée généralement sous la rubrique RCPSP (Ressouce Constrained Project Scheduling Problem). La deuxième classe de travaux considère aussi que les indisponibilités se produisent généralement aléatoirement et que leur durée (réparation) est aussi aléatoire. A notre connaissance, il n'y a pas eu de travaux qui ont abordé l'incertitude sur les dates d'occurrence des disponibilités et sur leurs durées.

Dans le chapitre qui suit, nous commençons la présentation de notre contribution par l'étude préliminaire de l'incertitude sur les dates de début des indisponibilités et sur leurs durées.

Chapitre 3

Étude préliminaire de l'ordonnancement avec une seule contrainte d'indisponibilité par processeur

3.1 Introduction

L'incertitude sur les valeurs des données du problème est une propriété intrinsèque aux plateformes de calcul distribué. En particulier, les dates de début et de fin de ces périodes ne peuvent pas être figées et sont sujettes à des perturbations malgrè qu'il existe des mécanismes pour gérer les indisponibilités dans ces plateformes. La prise en compte de ces contraintes lors de la conception de l'ordonnancement s'avère une étape cruciale afin d'obtenir un ordonnancement qui soit à la fois performant, et aussi qui résiste aux éventuelles perturbations.

Dans ce chapitre, nous commençons par étudier le cas où au plus une seule indisponibilité est programmée par machine. Nous considérons aussi qu'une machine est toujours disponible. Dans les environnements distribués que nous étudions, nous considérons que les dates de début et/ou de fin d'indisponibilité sont sujettes à des perturbations. Nous commençons dans la section 3.2 par donner les définitions du problème. En particulier, nous détaillons les différentes mesures que nous allons adopter. Nous passons dans la section 3.3 à l'étude de la complexité du problème. La section 3.4 est dédiée

à l'étude des algorithmes basés sur LPT l'heuristique (Longest Processing Time). Nous présentons ensuite une nouvelle famille d'algorithmes flexibles qui représentent un compromis entre la performance et la stabilité. Nous montrons à la fin que cette famille accomplie un résultat très proche de la borne inférieure.

3.2 Modèle d'applications parallèles

Les tâches parallèles sont des tâches qui s'exécutent sur plusieurs processeurs simultanément [29]. La prise de décision sur le nombre de processeurs utilisés ainsi que l'évolution de ce nombre au cours de l'exécution est le facteur principal de différentiation entre le type de tâches. Dans la littérature, les types suivants ont été définis :

Les tâches Rigides Le nombre de processeurs utilisés ici est donné et fixé à l'avance. Par conséquence, la taille (durée) de ce type de tâche est connu a priori. Dans la littérature [79], le type des tâches qui utilise i-processeurs est appelé i-tâches (i-tasks).

Les tâches moldable Dans ce modèle, le nombre de processeurs sur lequel la tâche va s'exécuter est défini par l'ordonnanceur avant le début de l'exécution. Une fois ce dernier est fixé, il restera figé jusqu'à la fin de l'exécution de la tâche. Il est possible d'exprimer la durée de cette classe de tâches en fonction du nombre de processeurs définis[28]. Il est aussi possible de limiter les valeur des tailles des tâches à un ensemble fini et dont la valeur adéquate dépend du nombre de processeurs.

Les tâches malléables Ce modèle peut être vu comme une extension du modèle précédent où le nombre de processeurs choisi par l'ordonnanceur varie au cours de l'exécution. Une tâche peut utiliser de nouveaux processeurs et peut aussi en libérer si le besoin est justifié. L'infrastructure cible doit être capable d'installer l'application sur un nouveau processeur et doit offrir les mécanismes pour libérer une tâche allouée à un processeur. Ces mécanismes peuvent être réalisés de différentes manières comme la migration des tâches dans un système d'exploitation à temps partagé ou par la migration de tout l'environnement de calcul dans un système qui gère la virtualisation. Les mécanismes de préemption et de checkpoint sont aussi nécessaires pour les plateformes cibles afin de pouvoir gérer l'aspect dynamique.

Une tâche parallèle possède une quantité totale de travail à effectuer. La durée de l'exécution de la tâche varie selon certaines fonctions dont nous citons :

- idéalement parallélisable (préservation de la quantité de travail) : dans ce modèle, la durée de la tâche est égale au rapport entre la durée de la tâche exécutée sur un seul processeur et i. Ce type de tâches n'existe presque pas dans la pratique.
- durée des tâches non-croissantes en fonction de i : avec ce type de tâches, plus le nombre de processeurs que la tâche utilise est grand, plus la durée de la tâche est petite.
- durée de tâche est une fonction discrète de i : les valeurs que peuvent prendre la durée des tâches appartiennent à un ensemble discret de valeurs possibles.

Par ailleurs, il est aussi possible de définir certains autres types de tâches parallèles en fonction d'une éventuelle topologie particulière des processeurs requis où en fonction de certains types de processeurs exigés par certaines tâches. Une littérature étendue a été établie dans [78].

Dans ce chapitre, nous nous intéressons aux applications du type *sac de tâches*. Ces applications sont en fait composées d'un ensemble de tâches monoprocesseurs indépendantes [19]. Ce type de d'application, malgré sa simplicité apparente, est utilisé dans plusieurs domaines dans les grilles de calcul comme le balayage des paramètres [4], la fouille des données [22], la bioinformatique [42] et le domaine traitement d'images (comme la tomographie [75]). L'indépendance des tâches facilite leur exécution sur des grilles potentiellement large comme l'avait montré l'expérience du projet SETI@HOME.

3.3 Définition du problème

Dans le contexte des Desktop Grid, les utilisateurs soumettent un ensemble de tâches, le plus souvent indépendantes. Ces tâches vont s'exécuter sur des machines (PCs) connectés à travers le Desktop Grid. Dans le contexte de ce chapitre, nous considérons que les machines ont sensiblement la même vitesse d'exécution. Pour plus de simplicité, nous considérons que les vitesses des machines est 1 et que le coût d'exécution de la tâche (C_j) est par conséquent égale à sa durée (p_j).

Nous disposons aussi de m machines parallèles indexées par i. Sur l'ensemble des machines, nous considérons qu'il existe au moins $\tau > 1$ proces-

seurs disponibles. Cette hypothèse est indispensable à l'obtention de résultats théoriques bornés. De plus, dans [53], Kondo et al. caractérisent la disponibilité des processeurs et montrent que 20% des machines ont été disponibles pendant 95% du temps. Chaque machine i , $i = 1..m - \tau$, possède une indisponibilité qui commence à la date s_i et se termine à la date e_i. La durée de l'indisponibilité est $u_i = e_i - s_i$. Comme il existe une seule indisponibilité par processeur, il n'est pas nécessaire d'indexer les paramètres par le numéro de l'intervalle.

Dans le contexte des grilles de calcul, les données sont sujettes à des perturbations. Ainsi, les dates d'occurrence des évènements, la durée des tâches sont données à titre prévisionnel. Lors de la phase d'exécution des facteurs extérieurs peuvent causer une perturbation de ces données et on obtient ainsi des valeurs perturbées. Si s dénote la valeur prévue d'une variable, $\tilde{s}$ dénote la valeur perturbée de cette variable.

Dans le cadre de ce chapitre, nous nous intéressons aux perturbations sur la durée des tâches. Lors de la phase d'exécution nous considérons que la date de début de l'indisponibilité sera perturbée. Soit δ_i l'ampleur de la perturbation. La date de début de l'indisponibilité commence à l'instant : $\tilde{s}_i = s_i - \delta_i$, $0 \leq s_i \leq \delta_i$, $i = 1..\tau$. L'ensemble de ces notations est illustré dans la figure 3.1.

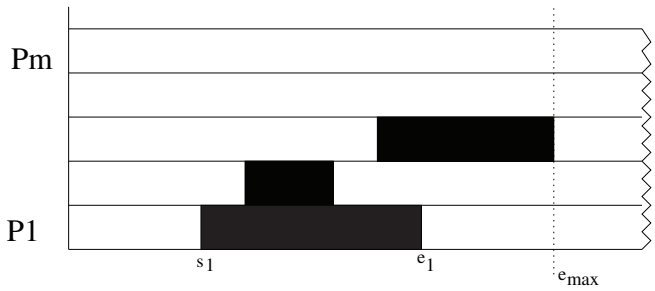


FIGURE 3.1 – Illustration des variables

Le modèle d'exécution que nous adoptons est un modèle non-préemptif et sans migration. Lors de la phase d'exécution, si la perturbation cause l'interruption d'une tâche, celle-ci sera entièrement reprise sur le même processeur juste après la fin de l'exécution prévue.

Notre but est de concevoir des méthodes qui produisent des algorithmes performants et qui résistent le mieux aux perturbations. Le critère de performance que nous adoptons est le critère classique de la minimisation du

makespan. Ce critère caractérise la date de fin de la dernière tâche exécutée de l'ensemble de tâches J. Si C_j est la date de fin d'exécution de la tâche J_j alors le makespan de l'algorithme A est définit analytiquement comme suit :

$$C_{max}^A = max_{j \in \{1..n\}} C_j$$

Comme nous l'avons abordé dans le chapitre 2, dans les environnements perturbés il existe de nouvelles mesures qui permettent de qualifier la solution réellement obtenue. Dans le cadre de ce travail, nous mesurons la stabilité qui est définie comme la distance entre le makespan de l'ordonnancement actuel et le makespan de la solution prévue qui a été calculé en utilisant les valeurs initiales (non perturbées) des variables de l'entrée du problème.

Définition 1. *Soit A un algorithme d'ordonnancement et de makespan C_{max}^A pour une instance I. Soit $\tilde{C}_{max}^A$ le makespan perturbé obtenu après l'exécution de l'instance perturbée $\tilde{I}$. La stabilité de A est le plus grand rapport entre $\tilde{C}_{max}^A$ et C_{max}^A pour tous les scénarios perturbées possibles.*

$$\sigma = max_{\tau \in [0,1]} \frac{\tilde{C}_{max}^A}{C_{max}^A}$$

Cette mesure permet de caractériser la perte que le makespan a subi dans l'instance perturbée par rapport a celui qui a été prévu à cause des perturbations.

Proposition 4. *Le problème $Pm, 1up|nr - a|C_{max}$ est un problème NP-difficile au sens fort même pour $m \geq 2$.*

La démonstration de ce résultat se fait par réduction à partir du problème 3-partition qui est lui même NP-difficile au sens fort. Commençons d'abord par rappeler le problème 3-partition [34].

Problème : 3-partition

Instance : un ensemble de $3k$ entiers a_i , et une borne B telle que $\sum_i a_i = kB$.

Question : déterminer s'il existe une partition de $[1..n]$ en k groupes G_l, tous de cardinal 3, telle que $\forall l$, $\sum_{i \in G_l} a_i = B$.

Démonstration. Notons d'abord qu'il est facile de vérifier que ce problème appartient à NP. En effet, la vérification d'une instance correcte se fait en un temps polynomial en fonction de m et de n.

Considérons une instance I de 3-partitions $\sum_{i \in G_l} a_i = B$. A partir de I, nous construisions une instance I' de notre problème comme suit :

- 1 machine n'ayant aucune indisponibilité et k machines ayant des indisponibilités qui commencent à B et se termine à $B+1$
- $3k+1$ tâches : $3k$ tâches de taille $p_i = a_i$ et une tâche (virtuelle) de taille $B+1$.

Si I' est une instance avec un makespan optimal dont la valeur du makespan est $B+1$ alors l'instance I est forcement une instance optimale. Cela s'obtient en considérant les tailles des tâches ordonnancées sur les processeurs ayant une indisponibilité comme des entiers des partitions de taille B. Inversement, si I est une instance ayant une solution optimale pour le problème 3-partition, alors I' est aussi une instance ayant un makespan optimal qui vaut $B+1$. Pour cela, il faut placer les tâches qui correspondent à chacune des partitions de I sur les processeurs ayant une indisponibilité (entre les instants 0 et B) et placer la tâche virtuelle sur le processeur totalement libre.

Ainsi, nous avons montré que notre problème est NP-difficile au sens fort. □

3.4 Analyse des algorithmes basés sur LPT

3.4.1 Analyse de l'algorithme LPT classique

Comme nous l'avons mentionné dans le chapitre précédent, Lee [59] a étudié le problème d'ordonnancement avec une seule contrainte d'indisponibilité par processeur et où au moins un processeur est toujours disponible. Il montre que la ratio d'approximation de LPT est $\frac{m+1}{2}$. Il est aussi possible d'exprimer cette borne en fonction de τ comme $1 + \frac{m-\tau}{2}$.

Dans le contexte d'un environnement sujets aux perturbations, il est possible de concevoir des algorithmes ayant une stabilité aussi mauvaise que l'ont veut. Il est facile de concevoir des instances ayant une stabilité infinie.

Soit par exemple l'instance suivante :

- $m = 2$: deux machines disponibles
- $\tau = 1$: une machine est toujours disponible
- $s_1 = 1 - \varepsilon$ et $t_1 = \infty$
- $p_1 = 1$ et $p_2 = 1 - \varepsilon$

Pour cette instance LPT génère un ordonnancement optimal dont le makespan est 1. Cependant, une perturbation sur la date de début de l'indisponibilité fait que la tâche de durée $1 - \varepsilon$ soit interrompue et devra être redémarree.

Dans notre modèle la migration des tâches n'est pas autorisée. Par conséquent, il faudra attendre la fin de l'indisponibilité pour ré-exécuter la tâche. Comme cette durée est infinie, la stabilité de l'ordonnancement généré est arbitrairement mauvaise.

Cette instance correspond dans la réalité à une machine volontaire dont le propriétaire la reprend pour une très longue durée avant de la rendre de nouveau disponible. Afin d'obtenir des résultats avec des bornes finies, nous imposons que toutes les indisponibilités doivent se terminer avant l'horizon du makespan optimal. Analytiquement, cette contrainte s'écrit $e_{max} < C^*_{max}$ où $e_{max} = max_{i \in \{1..m-\tau\}} e_i$.

Cette hypothèse est interprétée comme étant nous souhaitons faire plus de calcul que l'horizon qui nous est disponible après la fin de toutes les indisponibilités. En considérant ces hypothèses, le makespan de LPT sera majorée comme suit :

Proposition 5. *Le makespan de LPT est majoré par* $C^{LPT}_{max} \leq \frac{5}{3} C^*_{max}$

Démonstration. La preuve de ce résultat est faite par une analyse de la date de complétion des tâches en fonction de leurs durées.

Les tâches dont la durée est supérieure à $\frac{1}{3} C^*_{max}$ se terminent avant $\frac{3}{2} C^*_{max}$. En effet, il existe au plus $2m$ telles tâches. Dans un ordonnancement optimal, ces tâches se terminent avant C^*_{max}. Il existe un intervalle suffisamment long pour contenir deux telles tâches. LPT va ordonnancer au moins l'une de ces tâches dans l'intervalle en question. Comme LPT considère les plus grande tâches d'abord, les tâches dont la durée est supérieure à $\frac{1}{2} C^*_{max}$ vont se terminer avant C^*_{max}. Il reste donc au plus m tâches dont la durée est comprise entre $\frac{1}{3} C^*_{max}$ et $\frac{1}{2} C^*_{max}$ qui vont commencer avant C^*_{max} et vont par conséquence se terminer avant $\frac{3}{2} C^*_{max}$.

Nous montrons dans la suite que cette borne sera dominée par la $\frac{5}{3} C^*_{max}$ qui est la pire date de fin d'exécution des tâches dont la durée est inférieure à $\frac{1}{3} C^*_{max}$. Une telle tâches va soit être placée devant une indisponibilité, et va donc se terminer avant C^*_{max}, soit après. Dans le cas échéant, l'espace qui n'a pas été utilisé avant C^*_{max} est majorée par $\frac{1}{3} C^*_{max}$. La moyenne des dates de fin d'exécution des tâches sur les machines est $C^*_{max} + \frac{1}{3} C^*_{max} = \frac{4}{3} C^*_{max}$. Comme la dernière tâche n'a pas encore été exécutée, il existe forcément un

processeur libre avant $\frac{4}{3}C^*_{max}$. La tâche va donc être placée sur ce processeur et va donc se terminer avant $\frac{5}{3}C^*_{max}$. □

Considérons maintenant une perturbation qui affecte la date de début d'une ou de plusieurs indisponibilités. Selon notre modèle, si une tâche est interrompue alors elle sera re-éxécutée après la fin de l'indisponibilité sur le même processeur. Avec ces hypothèses, on obtient le résultat suivant :

Proposition 6. *La stabilité de LPT est bornée par 2 et cette borne est atteinte.*

Démonstration. Pour les besoins de la démonstration, nous définissons C^i, $i = 1..m - \tau$ comme la date de fin d'exécution de la dernière tâche exécutée sur le processeur i.

Comme les processeurs d'indices allant de $m-\tau$ à τ ne possèdent pas d'indisponibilités, les tâches qu'ils exécutent ne risquent pas d'être perturbées. Pour le reste des processeurs, cette valeur peut être dégradée comme suit :

$$\tilde{C}^i \leq C^i + \sum_{j \in S_i} p_j$$

où S_i est l'ensemble des tâches ayant été ordonnancé avant le début de l'indisponibilité sur les processeurs d'indice i, i.e, telles que $\sigma(J_j) = (i, t)$ avec $t < s_i$.

Comme le makespan de l'instance perturbé est $\tilde{C}^{LPT}_{max} = max_{i=1..m}(\tilde{C}^i)$ nous obtenons,

$$C^{LPT}_{max} \leq max_{i=1..m}(C^i) + \sum_{j \in S_i} p_j$$

Comme $\sum_{j \in S_i} p_j < C^{LPT}_{max}$, il en résulte :

$$C^{LPT}_{max} \leq 2C^{LPT}_{max}$$

Pour montrer que cette borne est atteinte, considérons une instance avec m tâches ; une tâche de durée $1 + \varepsilon$ et $m - 1$ tâches de durée 1. Soit $m - 1$ indisponibilités de durée ε et qui commencent à la date 1. LPT génère un ordonnancement dont le makespan est $1 + \varepsilon$. Le makespan de l'instance perturbée a un makespan $2 + \varepsilon$. Le rapport $\dfrac{2+\varepsilon}{1+\varepsilon}$ converge vers 2 quand ε tend vers 0. □

3.4.2 Stable-LPT

Dans cette section, nous concevons un algorithme avec la meilleure stabilité possible que nous désignons par stable-LPT que nous notons $sLPT$. Le principe de l'algorithme est d'ordonnancer toutes les tâches après la date de fin de la dernière indisponibilité, i.e, à partir de la date e_{max} et ce utilisant l'algorithme LPT. Cet algorithme a un ratio d'approximation de $\frac{7}{3}$.

Proposition 7. $C_{max}^{SLPT} \leq \frac{7}{3} C_{max}^{SLPT}$ *et cette borne est atteinte*

Démonstration. Soit I' l'instance du problème sans la prise en compte des indisponibilités et soit $C_{max}^{*}(I')$ le makespan correspondant. $C_{max}^{LPT}(I')$, qui est le makespan de l'algorithme LPT pour l'instance I' est majorée comme suit :

$$C_{max}^{LPT}(I') \leq \frac{4}{3} C_{max}^{*}(I')$$

Par ailleurs, le makespan de sLPT peut être écrit comme la somme de e_{max} et de $C_{max}^{sLPT}(I')$. Comme $e_{max} \leq C_{max}^{*}$ on a :

$$\begin{aligned} C_{max}^{sLPT} &\leq e_{max} + C_{max}^{LPT}(I') \\ &\leq C_{max}^{*} + 4/3 C_{max}^{*} \end{aligned}$$

donc

$$C_{max}^{sLPT} \leq \frac{7}{3} C_{max}^{*}$$

□

Il est évident que quelles que soit la magnitude de la perturbation aucune tâche ne sera perturbée. Le ratio de stabilité est par conséquent égal à 1. Cet algorithme est certes peu pratique puisqu'il est contre intuitif. Il est en effet difficile de justifier le non usage des ressources pourtant disponibles avant les débuts des indisponibilité. Cet algorithme prend son intérêt du fait qu'il représente le meilleur algorithme possible en terme de stabilité (stabilité optimal). Tout autre algorithme proposé devra donc avoir une performance meilleure que $\frac{7}{3}$ vis-à-vis du makespan. Sur la figure 3.4, cet algorithme est représenté par le point de coordonnées $(\frac{7}{3})$ qui domine tout les autres points vis-à-vis de la stabilité.

3.5 Nouvelle famille d'algorithmes flexibles

3.5.1 Principe

Considérons une instance du problème avec m processeurs et où chaque processeur possède au plus une seule indisponibilité. Dans cette section, nous conçevons une nouvelle famille d'algorithmes, appelée $fLPT$, basée sur une technique de remplissage où l'espace disponible est partitionné en deux zones (figure 3.2) :

PARTIE I composée de l'espace disponible avant les indisponibilités

PARTIE II qui est composée de l'espace disponible après les indisponibilités et par les τ processeur n'ayant aucune indisponibilité.

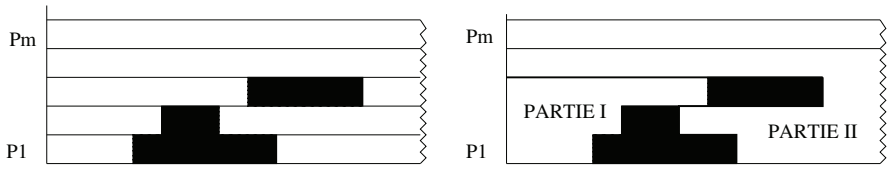


FIGURE 3.2 – Partitionnement de l'espace disponible en deux zones

3.5.2 Description

Le principe de l'algorithme consiste à utiliser un algorithme de remplissage pour sélectionner un sous ensemble de tâches à ordonnancer dans PARTIE I. L'effet des perturbations est anticipé en utilisant la technique de tampon, et qui consiste à laisser un espace avant les indisponibilités. Le reste des tâches est ordonnancé en utilisant l'algorithme LPT ou tout autre algorithme performant dans PARTIE II

Le principe de l'algorithme est de concevoir un ordonnancement qui absorbe l'effet des perturbations sur les dates de début d'indisponibilité. L'espace disponible dans PARTIE I est considéré comme un ensemble de $m - \tau$ sac-à-dos et un espace vide (tampon) est laissé systématiquement avant le début de chaque indisponibilité.

Nous utilisons un schéma d'approximation polynomial pour le problème de sac-à-dos multiple avec taille de sacs différentes (MSSP) comme algorithme de remplissage [51]. Il est cependant possible d'utiliser un autre al-

gorithme de remplissage (éventuellement moins performant) pendant cette phrase. L'algorithme est composé de deux phases décrites ci-dessous :

Algorithm 1 Algorithme flexible LPT

Input: $\beta \in [0, 1]$: paramètre de la taille du tampon
Output: un ordonnancement avec tampon

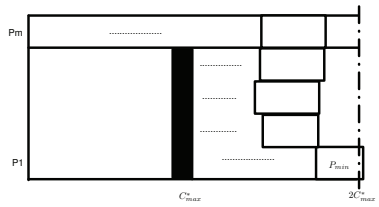
1: chaque processeur $i \in [1, m - \tau]$ nous considérons l'espace disponible $[0, (1 - \beta)s_i)$ comme étant un sac-à-dos . Nous obtenons ainsi, $m - \tau$ sac-à-dos de taille $\beta s_i, i = 1..m - \tau$. Pour tout $\varepsilon > 0$, nous utilisons un schéma d'approximation de ratio $1-\varepsilon$ pour sélectionner un sous ensemble des tâches qui seront ordonnancées avant les indisponibilités.
2: ordonnancer les tâches restantes en utilisant LPT.

3.5.3 Analyse de performance de fLPT

La famille de fLPT est une famille d'algorithmes paramétrée par β. Ce facteur représente la taille du tampon. Ainsi, quand $\beta = 1$ tout l'espace disponible avant l'indisponibilité sera utilisé et aucun espace libre ne sera réservé. Cependant, quand $\beta = 0$, tout cet espace est utilisé et aucun tampon ne sera considéré. Nous obtenons ainsi des ordonnancements dont le makespan est paramétré avec β comme suit :

Proposition 8. *La performance de fLPT est majorée par* 2

Démonstration. Nous considérons l'ordonnancement généré et nous élliminons toutes les tâches dont l'indice est plus petit que la dernière tâche (d'indice j) tel que $C_j = C_{max}$. Cette tâche sera notée P_{min}. Nous montrons le résultat par analyse de cas et à travers un raisonnement par l'absurde.



Supposons que $C_{max} > 2C^*_{max}$. Selon la taille de p_{min} on distingue les cas suivants :

- $p_{min} \geq \frac{1}{3}C^*_{max}$: dans l'ordonnancement initial, sur chacun des $m-1$ processeurs ayant une (eventuelle) indisponibilité, au plus, 3 tâches peuvent être placées devant l'indisponibilité.
  Au pire des cas, aucune parmi ces tâches ne peut être exécutée à cause de l'indisponibilité. Elles seront donc exécutées avec LPT juste après la fin des indisponibilités. Si une tâche de durée supérieure ou égale à $\frac{1}{3}C^*_{max}$ se termine après $2C^*_{max}$ alors chaque machine contient soit une seule tâche dont la taille est supérieure à $\frac{1}{2}C^*_{max}$ soit deux tâches dont la taille est supérieure à $\frac{1}{3}C^*_{max}$. PARTIE I est uniquement composée de $m-1$ machines ayant une disponibilité majorée par C^*_{max}. Un ordonnancement optimal va devoir placer les tâches de la machine m (une ou deux tâches) avec le reste des tâches dans PARTIE I. Dans tous les cas, ces tâches vont être placées avec une seule autre tâche de durée supérieure à $\frac{1}{3}C^*_{max}$. La tâche de durée p_{min} sera donc placée forcément avec des tâches la somme des durées dépasse $\frac{2}{3}C_{max}$ ce qui est absurde (on dépasse C^*_{max}).
- $p_{min} < \frac{1}{3}C^*_{max}$ de même, supposons que la taille de la plus petite tâche se termine après $2C^*_{max}$.
  - Si $\frac{1}{3}C^*_{max} > p_{min} > \frac{1}{m-1}C^*_{max}$ alors nous montrons que l'ordonnancement optimal ne peut pas placer toutes les tâches dans PARTIE I.
    Si on considère l'ensemble des tâches qui ont été affectées aux machines de 2 aux m. La borne inférieure d'ordonnancement optimal qui va ordonnancer ces tâches sur $m-1$ machines est supérieure à $1-\frac{1}{m}C^*_{max}$. Si nous ajoutons maintenant n'importe quelle tâche ordonnancée sur la machine 1 (qui ont toutes une durée supérieure à p_{min}) alors le placement de cette tâche va automatiquement engendrer le dépassement de la date C^*_{max}.
  - Si $p_{min} < \frac{1}{m}C^*_{max}$ alors la somme de la taille de toutes les tâches dépasse $(m-1)C^*_{max}$ puisque de C^*_{max} à $(2-\frac{1}{m})C^*_{max}$ les m processeurs disponibles sont occupées.

Remarque :
La preuve est indépendantes des tâches qui ont été ignorées. Donc, elle reste valide même si celles-ci n'auraient pas été ignorées.

□

Notons que si $\beta = 0$, i.e, nous ne laissons aucun espace libre avant les indisponibilité pendant la phase 1, alors la performance de fLPT est majorée par $\frac{3}{2}+\varepsilon$. Cependant, quand tout l'espace disponible avant les indisponibilités

n'est pas utilisé, cette performance est majorée par $\frac{5}{2}$.

Comme pour le cas précédent, considérons un ensemble de perturbations qui peuvent affecter les dates de début des indisponibilités. Si $s_i - \tilde{s}_i < \beta s_i$, $\forall i = 1..m - \tau$ alors les perturbations n'auront aucun effet sur l'exécution des tâches et par conséquent sur le makespan. Le vecteur $< \beta s_1, ...\beta s_{m-\tau} >$ peut alors être vu comme étant un rayon de stabilité.

Si cette relation n'est pas vérifiée au moins pour un indice i, alors la perturbation peut causer l'interruption d'une ou de plusieurs tâches. Celles-ci devront être exécutées après la fin de l'indisponibilité. La stabilité de fLPT est alors majorée comme suit :

Proposition 9. *La stabilité de fLPT est majorée par* $2 - \beta$

Démonstration. Le pire cas se produit quand la perturbation cause l'interruption de toutes les tâches qui ont été ordonnancées avant l'indisponibilité. La somme de toutes ces tâches est inférieure à βs_i. Donc, pour une machine d'indice i, la fin de de l'exécution C^i va au plus être dégradé de $(1 - \beta)s_i$. Nous avons par conséquence :

$$\tilde{C}^i \leq C^i + (1 - \beta)s_i \leq (2 - \beta)C^i$$

Il en résulte que

$$\tilde{C}_{max}^{fLPT} \leq (2 - \beta)C_{max}^{fLPT}$$

□

3.5.4 Borne inférieure pour le pire des cas

Dans cette section, nous construisons une classe d'instances qui représente la borne inférieure pour le pire des cas. Rappelons que nous avons toujours un processeur qui est libre de toute indisponibilité.

Soit K un entier positif très grand. Dans les équations qui suivent, nous arrondissons les valeurs vue que la valeur de K est très grande. Nous construisons une instance ayant $(m - 1)K$ tâches ayant chacune une taille de durée $(1 - \varepsilon)/K$ et une tâche de durée 1. L'instance se compose aussi d'un ensemble d'indisponibilité de durée ε et qui commencent à l'instant $1 - \varepsilon$ et se terminent à l'instant 1.

Pour tout $\beta \in [0, 1]$, nous pouvons construire un ordonnancement dont la valeur du makespan est $1 + \beta$ en plaçant la tâche de durée 1 sur la machine

P_m, $(1-\beta)(m-1)/K$ petites tâches sur chacun des processeurs P_i, $i = 1..\tau$ avant les indisponibilités et en plaçant $\beta(m-1)/K)$ après les indisponibilités.

Le pire cas se produit si lors de la phase d'exécution, la magnitude de perturbation la plus grande possible a lieu, i.e, une perturbation de $1-\varepsilon$. Toutes les tâches placées avant les indisponibilités vont alors s'exécuter derrière celles-ci et l'ordonnancement produit a par conséquent un makespan égal à 2. La stabilité de cette classe d'instance est $2/(1+\beta)$.

Tout autre ordonnancement ayant une stabilité de $\dfrac{2}{1+\beta}$ va avoir un makespan supérieur à $1+\beta$ et un makespan perturbé inférieur à 2. L'instance décrite ci-dessus, et illustrée dans 3.3, constitue alors la borne inférieure pour le pire des cas.

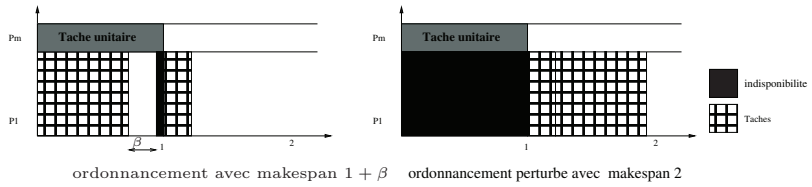


FIGURE 3.3 – Borne inférieure pour le pire des cas

3.6 Résumé des résultats

Nous récapitulons dans cette section les différents algorithmes proposés. Nous avons commencé par étudier le comportement de LPT pour le problème d'ordonnancement avec contraintes d'indisponibilité. Nous fixons le nombre maximal d'indisponibilités par processeur à 1. Un processeur au moins est toujours disponible. Nous avons montré que les ratios d'approximation et de stabilité sont respectivement 2 et 2.

Nous avons aussi proposé un algorithme $sLPT$ qui a la meilleure borne possible pour la stabilité. Le ratio d'approximation de cet algorithme constitue une borne supérieure pour tout algorithme proposé.

Nous avons enfin conçu une nouvelle famille d'algorithmes paramétrés par un facteur β qui varie entre 0 et 1. Ce facteur constitue en fait le compromis entre les ratios de stabilité et d'approximation. Ces deux objectifs étant contradictoires, notre famille permet de choisir entre ces deux critères

en fonction du besoin en jouant sur β. Dans un environnement de calcul distribué, ce paramètre sera fixé par l'administrateur du système en fonction de la qualité de l'environnement cible. Il est ainsi préférable dans un environnement relativement sûr i.e où les pannes et les évènements imprévus sont rares, de lancer l'application avec une valeur de β proche de zéro. Inversement, quand l'incertitude est grande par rapport aux évènements prévus, il est plus adéquat de paramétrer une valeur proche de 1 ce qui va permettre d'anticiper le mieux possible les effets des perturbations.

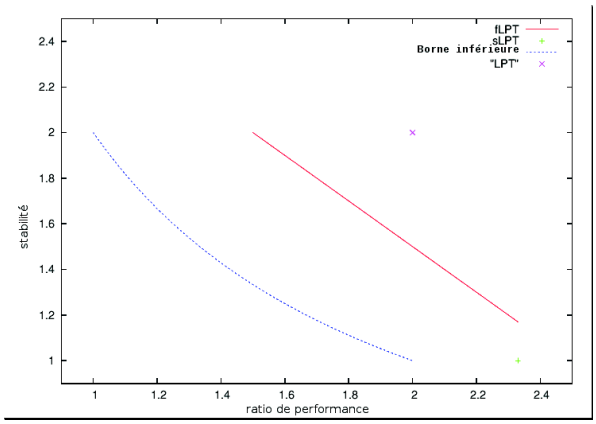


FIGURE 3.4 – Ratio d'approximation et ratio de stabilité de la famille d'algorithmes fLPT

Algorithme	ratio d'approximation	ratio de stabilité
LPT	2	2
sLPT	$\frac{7}{3}$	1
fLPT	$\begin{cases} \frac{5}{3} & \text{if } \beta = 0 \\ \min(\frac{7}{3}, \max(2, \frac{5}{3} + \beta)) & \beta > 0 \end{cases}$	$(2 - \beta)$

TABLE 3.1 – Tableau comparatif des différents résultats

3.7 Conclusion

L'incertitude liée à la nature distribuée des nouvelles plateformes d'exécution de calcul intensif est, certes, l'une des contraintes à aquelle une attention

particulière doit être accordée. Dans le contexte proactif, lors de la phase de construction de l'algorithme, l'effet nuisible des perturbations qui touchent les dates de début et de fin des indisponibilités des processeurs est anticipé par plusieurs techniques.

Dans ce chapitre, nous avons commencé par l'étude de l'impact des perturbations sur un ordonnancement construit avec LPT. Une seule indisponibilité est considérée par processeur. Les perturbations étudiées dans ce cadre portent sur la date de début de l'indisponibilité et sa durée. Aucune perturbation ne touche la date de fin de l'indisponibilité. Nous avons ensuite conçu un algorithme, qui a la meilleure stabilité possible. Nous sommes persuadé que cet algorithme est plutôt d'intérêt théorique mais qu'en contre partie, il représente le pire algorithme en terme de performance.

Nous avons par la suite, conçu une famille d'algorithmes flexibles qui offre un compromis paramétrable en la performance et la stabilité. Cette fin est atteinte grâce à l'adoption de la technique de tampon sur les ressources et qui consiste à anticiper les effets des perturbations par un espace vide paramétré qui est laissé systématiquement devant les indisponibilités.

Dans le chapitre qui suit, nous généralisons l'étude faite pour une seule indisponibilité par processeur à un nombre quelconque d'indisponibilités.

Chapitre 4

Algorithme bi-objectif pour l'ordonnancement des tâches avec contraintes d'indisponibilité

4.1 Introduction

La solution que nous proposons dans ce chapitre, dédié à l'étude de l'ordonnancement des tâches avec contraintes d'indisponibilité, suit une approche flexible : nous concevons un algorithme en nous basant sur les estimations existantes, et nous effectuons des corrections simples lors de l'exécution pour réagir aux perturbations éventuelles. Plusieurs publications ont abordé le problème étudié ici mais en adoptant des approches pro-actives basées sur les tampons.

Dans [32], Fallah et al. ont étudié la version avec préemption. Dans [56, 55], les Herroelen et al. ont exploré le problème où les pannes des ressources (arrivé des indisponibilités) suivent des lois stochastiques.

Dans ce chapitre, nous nous intéressons au problème où ni la préemption ni la migration n'est autorisée. Ainsi, si une tâche est interrompue à cause des perturbations, elle sera entièrement reprise ultérieurement sur le même processeur. La réaction aux incertitudes est par conséquence limitée. Nous commençons par préciser le modèle auquel on s'attaque, puis nous définissons théoriquement le problème. Nous passons dans la section 4.4 à l'analyse de la

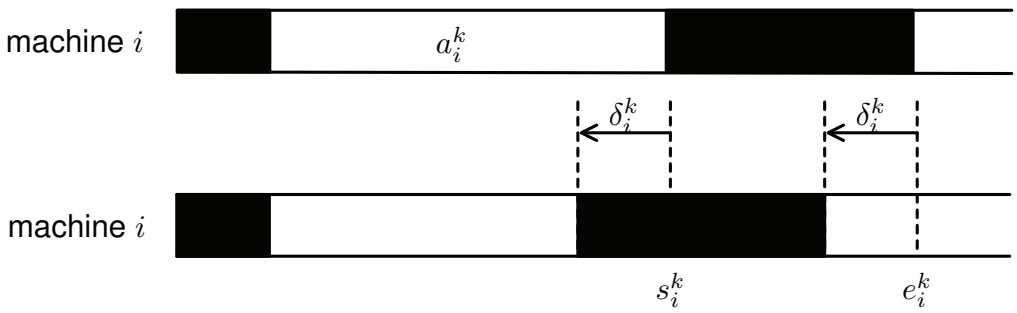


FIGURE 4.1 – L'indisponibilité k peut commencer plus tôt à cause de la perturbation δ_i^k.

stabilité et nous proposons, dans la section 4.5, un algorithme bi-objectif qui offre un compromis entre deux critères, à savoir, la stabilité et la performance.

4.2 Modèle de perturbation

Nous suivons le modèle introduit dans la section 2.5 sauf pour les perturbations. Comme les processeurs sont identiques, nous pouvons considérer que leur vitesse est unitaire. Par la suite le coût des tâches représente aussi leur durée.

Soit δ_i^k la perturbation qui affecte l'indisponibilité k sur le processeur i. Comme nous considérons que la perturbation affecte uniquement la date de début d'indisponibilité, nous définissons la date de début d'indisponibilité perturbée $\tilde{s}_i^k = s_i^k + \delta_i^k$. Dans ce chapitre, nous traitons le cas où les indisponibilité avancent (se produisent avant la date prévue). Les indisponibilités correspondent à des périodes où la ressource est occupée par son propriétaire. Elles ne peuvent donc pas se superposer. La valeur maximale de δ_i^k est donc a_i^k : $-a_i^k \leq \delta_i^k \leq 0$. L'incertitude porte uniquement sur la date de début de l'indisponibilité. Sa durée reste donc fixe lors de la phase de l'exécution.

4.3 Définition du problème

Étant données les valeurs estimées du problème (durée des tâches, date de début et durée d'indisponibilité), l'objectif est de générer un ordonnancement

réalisable qui satisfait toutes les contraintes spécifiées a priori.

Dans le scénario perturbé, les indisponibilités peuvent commencer plus tôt que les dates prévues (selon le modèle de perturbation spécifié ci-dessus). L'adaptation de l'ordonnancement se fait dynamiquement en utilisant les deux règles suivantes :

1. Chaque tâche interrompue s'exécute le plus tôt possible tant qu'elle ne cause pas de retard sur les dates de début d'exécution des tâches prévues sur le même processeur.
2. Quand un processeur devient libre, il commence à exécuter les tâches qui lui ont déjà été affectées.

Notre objectif est de générer un ordonnancement de bonne qualité dont les critères sont : la performance (rapport entre le makespan obtenu et le makespan optimal) et l'aptitude à résister aux perturbations.

Le premier critère est évalué en mesurant le makespan de l'ordonnancement obtenu (de référence), i.e., en utilisant l'instance initiale du problème où aucune perturbation n'est prise en compte.

Le second objectif est la stabilité définie par $\sigma = \tilde{C}_{max}/C_{max}$. Ce facteur représente l'insensibilité de l'ordonnancement aux perturbations.

Le problème consiste ainsi à trouver un ordonnancement qui, à la fois, minimise le makespan et maximise la stabilité.

4.4 Analyse de stabilité

Dans cette section, nous présentons le mécanisme principal de stabilité utilisé pour faire face à l'incertitude au niveau des dates d'occurrence des indisponibilités. Il consiste à réserver de l'espace libre, i.e., dans lequel on interdit l'exécution des tâches. Cet espace libre ou tampon sera utilisé pour re-exécuter les tâches interrompues de façon à retarder le makespan de l'ordonnancement de référence le minimum possible.

Définition 2. *Le temps libre d_i^k qui précède l'indisponibilité k sur le processeur i est appelé tampon*

Définition 3 (règle de tampon)**.** *Chaque tampon doit être plus grand où égal à la taille maximale des tâches qui ont été allouées à l'intervalle en question, i.e., pour chaque intervalle k sur une machine i on a*

$$d_i^k \geq \max_{j \in \pi(i,k)} p_j$$

Proposition 10. *Tout ordonnancement basé sur la règle de tampon est stable s'il n'existe aucune tâche ordonnancée dans le dernier intervalle de disponibilité qui précède l'indisponibilité qui se termine après le makespan, et ce, pour tous les processeurs, i.e., la dernière disponibilité k pour laquelle e_i^k est inférieur ou égal au makespan.*

Démonstration. Notons qu'un ordonnancement ne peut pas être stable s'il n'existe pas des machines (processeurs) sans indisponibilité : quand $\tau = 0$, la tâche qui se termine en même temps que le makespan peut être interrompue est re-exécutée après le makespan de référence. Dans ce cas, la proposition 10 sera violée car cette tâche commence nécessairement sur une machine pendant le dernier intervalle de disponibilité qui commence avant le makespan.

□

Nous avons montré que le problème d'ordonnancement avec contraintes d'indisponibilité est NP-difficile au sens fort par réduction du problème de 3-Partition (voir 3.3).

Théoreme 1. *Trouver un ordonnancement qui minimise le makespan et qui respecte la contrainte de tampon est un problème NP-difficile au sens fort.*

Démonstration. Tout d'abord, remarquons que le problème d'ordonnancement avec la règle de tampon est dans NP.

La preuve est basée sur une réduction à partir du problème de 3-partition [34]. Nous gardons les mêmes notations que celles du paragraphe 3.3.

Nous construisons une instance du problème à partir d'une instance de 3-partition de la façon suivante. $4m$ tâches sont ordonnancées sur m processeurs. $3m$ parmi ces tâches ont une durée d'exécution égale aux éléments du problème de 3-partition et m autres tâches ont une durée de $B + 1$. Afin de respecter la contrainte de tampon, nous ciblons une valeur de makespan égale à $3B + 2$.

Chaque processeur contient exactement une seule tâche de taille $B + 1$, puisque mettre deux telles tâches ensemble sur le même processeur va violer la contrainte de tampon, vu que $2(B+1)+\max(B+1, B+1) = 3B+3 > 3B+2$. La forme de l'ordonnancement est illustrée dans la figure 4.2.

Ainsi, résoudre le problème d'ordonnancement revient exactement à résoudre 3-partition. Il en résulte que le problème d'ordonnancement avec contraintes d'indisponibilité est NP-difficile au sens fort. □

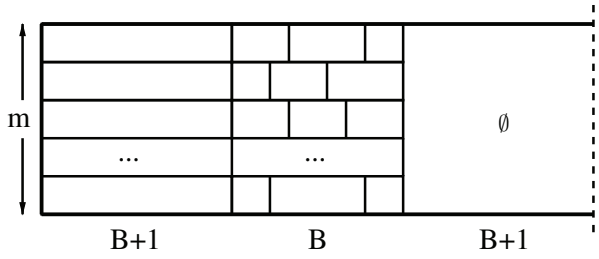


FIGURE 4.2 – Forme d'un ordonnancement qui respecte la règle de tampon

4.5 Algorithme bi-objectif

Dans cette section nous décrivons un algorithme bi-objectif, noté GAPS, et nous analysons sa stabilité.

4.5.1 Description

Notre algorithme bi-objectif utilise un paramètre de compromis β pour générer des ordonnancement résistant aux perturbations. D'une façon informelle, ce paramètre indique le degré de la contrainte de tampon préfixée. Ainsi, l'espace minimal non utilisé dans chaque intervalle est proportionnel à β, i.e., $d_i^k \geq \beta \times \max_{j \in \pi(i,k)} p_j$. De plus, les tâches ne sont jamais ordonnancées dans les intervalles qui débutent avant le makespan. Quand $\beta = 1$, la contrainte de marge est respectée, l'ordonnancement généré est par conséquent stable. Quand $\beta = 0$, la règle de tampon est ignorée.

La première étape de l'algorithme GAPS consiste à remplir les intervalles de disponibilité sans violer la contrainte de la marge. Cette étape peut être vue comme une version modifiée de l'algorithme First Fit Decreasing du problème de bin-packing[48]. Dans la seconde étape, les tâches n'ayant pas été sélectionnées sont ordonnancées sur les τ processeurs libres. La transition se fait à la seconde étape quand la condition de la ligne 8 n'est plus vérifiée. Celle-ci consiste à calculer une borne inférieure pour le temps nécessaire à l'exécution de ces tâches sur les τ processeurs libres.

Comme aucune tâche ne doit être exécutée dans un intervalle de disponibilité qui commence après le makespan, cette condition garantie que la dernière tâche exécutée (telle que $C_j = C_{max}$) sera exécutée sur l'un des processeurs libres.

Algorithm 2 Algorithme Glouton d'allocation avec tampon paramétré (GAPS)

Input: Un ensemble de tâches J
Output: La fonction d'allocation π

1: Créer une liste L triée des les intervalles par ordre croissant de e_i^k (date de la fin de l'indisponibilité)
2: Trier les tâches par durée décroissante de p_j (durée)
3: $S = J$ {L'ensemble des tâches non-ordonnancées}
4: **for all** intervalle(i, k) **do** {Considérer le premier intervalle de la liste L (intervalle k sur le processeur i)}
5: **for all** $j \in S$ **do** {Considérer les tâches dans l'ordre de tri}
6: $M = \sum_{j' \in \pi(i,k)} p_{j'}$ {Somme des temps d'exécution de l'intervalle courant}
7: **if** $e_i^k \leq \frac{\sum_{j' \in S} p'_j - p_j}{\lambda}$ **then**
8: **if** $a_i^k - M - p_j \geq \beta \max_{j \in \pi(i,k)} p_j$ **then** {règle de tampon relaxée}
9: $\pi(i, k) = \pi(i, k) \cup \{j\}$ {Ordonnancer la tâche j dans la disponibilité courante}
10: $S = S \setminus \{j\}$ {mise à jour de l'ensemble S}
11: **end if**
12: **end if**
13: **end for**
14: **end for**
15: Ordonnancer le reste des tâches en utilisant LPT sur les τ processeurs entièrement disponibles

Coût de GAPS : Les étapes principales de GAPS consistent à trier les tâches et les intervalles de disponibilité. Son coût est donc majoré par $nlog(n)$ si le nombre de tâches est supérieur au nombre d'intervalles, ou par $n_i log(n_i)$ sinon (où n_i est le nombre d'intervalles).

4.5.2 Analyse théorique

La contrainte de marge que nous avons imposée dans les paragraphes précédents peut avoir un effet de seuil. Il est en effet possible de construire des instances où les tâches sont ordonnancées efficacement dans les intervalles de disponibilité quand $\beta = 0$, mais, si $\beta > 0$, aucune tâche ne sera sélectionnée. Pour cela, il suffit de considérer un ensemble de tâches dont la taille est égale à la durée des intervalles de disponibilité et un autre ensemble de tâches de grande taille.

Afin de pouvoir placer au moins une tâche dans chaque intervalle de disponibilité, nous adoptons une nouvelle supposition sur la durée des tâches par rapport à la durée des disponibilités. Ces durées devront être supérieures au moins deux fois à la durée maximale des tâches. Analytiquement, cette supposition s'écrit comme suit : $2p_{\max} \leq a_{\min}$ où $p_{\max} = \max_j p_j$ et $a_{\min} = \min_{i,k} a_i^k$

Nous définissons aussi un ratio d'indisponibilité γ pour empêcher l'obtention de ratio de stabilité arbitrairement grand. Ce ratio caractérise le plus grand ratio entre la durée d'une indisponibilité et la durée de la disponibilité qui la précède.

Définition 4. *Soit* $u_{\max} = \max_{i,k} u_i^k$ *le ratio d'indisponibilité* γ *est*

$$\gamma = \frac{u_{\max}}{a_{\min}}$$

Intuitivement, plus γ est grand, plus une tâche devra attendre pour sa prochaine exécution.

Théoreme 2. *En considérant les hypothèses définies ci-dessus on a :*

$$\rho = \begin{cases} \frac{5}{2} - \beta + \gamma & si\ \beta \neq 1 \\ 1 & sinon \end{cases}$$

Démonstration. Pour chaque ordonnancement construit avec GAPS, nous mesurons la durée du travail qui peut être interrompue et qui devra être

re-exécutée après le makespan dans le pire cas. L'analyse est faite pour un seul processeur, mais, elle est facilement extensible au cas des processeurs parallèles. Cela est dû au fait que le phénomène de perturbation est indépendant du processeur et le mécanisme de correction reste local à chaque processeur (pas de migration). Aucune corrélation n'existe entre les processeurs pendant la phase d'exécution.

Soit K le nombre d'intervalles qui se terminent avant le makespan sur un processeur i. La somme de toutes les marges est par conséquent $\sum_{k=1}^{K} \beta \times \max_{j \in \pi(i,k)} p_j$. Dans le pire scénario, la k-ième indisponibilité se termine en même temps que le makespan (*i.e.*, $e_K^i = C_{\max}$) et les périodes d'indisponibilités sont arbitrairement petites (*i.e.*, $\forall k \in [1..K], u_i^k = \epsilon$). Cette instance maximise le nombre de tâches ordonnancées et leur taille. Par conséquent, elle maximise aussi la somme des durées des tâches interrompues. En ignorant la durée des petites indisponibilités, la somme des durées des tâches ordonnancées sur la machine i est majorée par

$$C_{\max} - \sum_{k=1}^{K} \beta \times \max_{j \in \pi(i,k)} p_j$$

Dans le pire scénario, toutes les indisponibilités vont commencer avant la date prévue (tout en gardant de même durée). De plus, celle-ci vont interrompre une tâche de durée maximale placée dans l'intervalle de disponibilité correspondant juste avant que l'exécution de celle-ci ne s'achève. La somme des tâches qui devront être ré-exécutées après le makespan est alors $\sum_{k=1}^{K} \max_{j \in \pi(i,k)} p_j$.

Comme nous l'avons indiqué dans le modèle, l'exécution des tâches est compacte, donc, les tâches sont exécutées le plus tôt possible selon la stratégie de l'ordonnancement. Les indisponibilités peuvent interrompre au plus une seule tâche. Une partie de la quantité de travail peut être re-exécutée avant le makespan grâce au mécanisme de marge que nous avons conçu. Le reste devra être re-exécuté après le makespan. Cette quantité est majorée par $(1-\beta) \sum_{k=1}^{K} \max_{j \in \pi(i,k)} p_j$. Cette quantité est maximale quand la plus petite période de disponibilité est maximale. Cela se produit quand toutes les périodes de disponibilité sont de même taille (*i.e.*, $a_{\min} = \frac{C_{\max}}{K}$). De plus, la durée de la plus grande tâche est la moitié de cette quantité (*i.e.*, $p_{\max} = \frac{C_{\max}}{2K}$) et chaque intervalle contient au moins une telle tâche. Par conséquence la taille maximale du travail qui reste à exécuter après le makespan est $\frac{(1-\beta)}{2} \times C_{\max}$.

Cette quantité est divisée en deux parties $D1$ et $D2$. $D1$ correspond à la quantité de travail formée par les tâches de durée maximale et qui ont été exécutées après le makespan alors que $D2$ représente la quantité de travail nécessaire à la terminaisons d'une tâche ayant commencée son exécution avant le makespan et qui se termine après cet horizon.

Il existe $\left\lfloor \frac{(1-\beta)}{2} \times \frac{C_{\max}}{p_{\max}} \right\rfloor = \lfloor (1-\beta)K \rfloor$ tâches de taille maximale qui vont mettre un intervalle entier de disponibilité minimale et d'indisponibilité maximale pour être re-exécutés (puisque la tâche peut être interrompue une autre fois dans le même intervalle). Ainsi,

$$D_1 = \lfloor (1-\beta)K \rfloor (a_{\min} + u_{\max})$$

Le second terme de cette expression (*i.e.*, $\frac{(1-\beta)}{2} C_{\max} \mod p_{\max}$) correspond au temps nécessaire pour exécuter la tâche qui commence avant le makespan et qui se termine après celui-ci. Par conséquent, nous avons :

$$D_2 = (((1-\beta) \times K \mod 1) + \frac{1}{2})a_{\min} + \lceil (1-\beta)K \mod 1 \rceil u_{\max}$$

En résumé, le makespan perturbé est exprimé comme suit :

$$\begin{aligned}
\tilde{C}_{\max} &= C_{\max} + D_1 + D_2 \\
&\leq C_{\max} + (1-\beta)Ka_{\min} + Ku_{\max} + \frac{a_{\min}}{2} \\
&\leq C_{\max} + (1-\beta)C_{\max} + \gamma C_{\max} + \frac{C_{\max}}{2}
\end{aligned}$$

La stabilité de GAPS est directement dérivée de cette équation.

□

4.6 Conclusion

Dans ce chapitre, nous avons étudié le problème d'ordonnancement sous contrainte d'indisponibilité et d'incertitude sur les dates de début des indisponibilités. Tous les paramètres du problème sont connus a priori (durée des tâches, dates d'occurrences des indisponibilités ...).

Les incertitudes portent uniquement sur les dates de début des indisponibilités qui peuvent avoir lieu plus tôt que prévu. Leurs durées restent cependant inchangées. La réaction aux perturbations se fait en re-exécutant

la tâche, éventuellement interrompue, immédiatement après la fin de l'indisponibilité.

Notre objectif est de concevoir des algorithmes qui résistent aux perturbations et qui minimisent le makespan. La qualité de la solution perturbée est mesurée en calculant le rapport entre le makespan de l'ordonnacement initial et celui de l'ordonnancement perturbé. Nous avons montré que ce problème est NP-difficile au sens fort par réduction du problème de 3-partition.

Notre contribution principale fut la conception d'un algorithme qui offre un compromis entre la stabilité et la performance et qui est basée sur un ordre de considération des intervalles et sur la contrainte du tampon paramétré. Nous avons établie la borne théorique de la stabilité.

Dans le chapitre 6, nous avons fait une campagne d'expérimentation pour évaluer le comportement de l'approche conçue et pour déterminer la bonne valeur du paramètre du tampon.

Dans le prochain chapitre, nous généralisons le modèle de perturbation et nous proposons plus d'algorithmes pour la construction d'algorithmes de référence.

Chapitre 5

Conception d'algorithmes d'ordonnancement pour les environnements fortement pertubés

5.1 Introduction

Après avoir étudié dans le chapitre précédent l'ordonnancement avec plusieurs contraintes d'indisponibilité par processeur, et où les indisponibilités peuvent uniquement se produire à l'avance, nous passons dans ce chapitre à l'étude du cas général.

Nous définissons le problème, qui consiste à produire un ordonnancement, qui fait le meilleur compromis possible entre stabilité et performance, dans un environnement où les indisponibilités sont perturbées et peuvent aussi bien survenir plutôt que plus tard.

Au début, nous proposons une technique basée, comme au chapitre précédent, sur les tampons pour atteindre ce compromis. Couplé avec un algorithme d'exécution dynamique que nous avons conçu, nous obtenons des garanties raisonnables pour la stabilité.

Finalement, nous concevons plusieurs variantes d'algorithmes, basées sur la technique de tampon et qui font des compromis entre l'horizon et la stabilité.

5.2 Définitions et modèles

Nous avons dans les chapitres qui précèdent étudié le problème dans une configuration où les indisponibilités peuvent uniquement être perturbés vers l'avant (i.e, avancer). Dans ce chapitre, nous généralisons cette étude pour les deux cas, à savoir, où la perturbation sur la date de début de l'indisponibilité peut aussi bien la faire produire plus tôt où plus tard.

Nous nous limitons aux instances réalisables, dans lesquelles les disponibilités peuvent ordonnancer toutes les tâches et dont le coût est connu à l'avance.

Soit δ_i^k la perturbation qui touche l'indisponibilité k sur le processeur i. Comme l'incertitude porte sur la date de début de l'indisponibilité, celle-ci est notée $\widetilde{s}_i^k = s_i^k + \delta_i^k$. La durée de l'indisponibilité reste néanmoins non perturbée, nous avons donc $\widetilde{u}_i^k = u_i^k$. La date de fin de l'indisponibilité perturbée est par conséquence $\tilde{e}_i^k = e_i^k + \delta_i^k$.

D'un autre coté, l'amplitude de la perturbation est limitée à la taille des intervalles qui précèdent et succèdent l'indisponibilité (*i.e.* $-a_i^k \leq \delta_i^k \leq a_i^{k+1}$). Cette limitation permet de minimiser l'erreur dûe à la prédiction et $-a_i^k \leq \delta_i^k \leq a_i^{k+1}$. Avec ce modèle de perturbation, il n'y aura pas plus que deux interruptions par intervalle (et un seul dans le cas où les indisponibilités avancent).

Comme auparavant, la qualité de l'ordonnancement est évaluée à l'aide d'un double critère : la performance et la stabilité, qui est l'aptitude de l'ordonnancement à résister aux perturbations.

L'horizon est défini comme la plus grande date de fin des indisponibilités qui succèdent au makespan. La première indisponibilité qui se termine après l'horizon est décalée de façon qu'elle se termine en même temps que l'horizon. Comme toutes les tâches doivent pouvoir rentrer dans toutes les indisponibilités, l'horizon est supérieur ou égal au makespan

$$C_{\max} \triangleq \max_j C_j$$

. où $\max_j C_j$ est la date de fin de la tâche j.

Dans le chapitre précédent, nous avons mesuré le makespan au lieu de l'horizon. Par conséquence, nous avons adopté une hypothèse supplémentaire qui consiste à supposer que le makespan optimal est supérieur ou égal à la date de la dernière indisponibilité. Considérer l'horizon donne par conséquence plus de flexibilité et plus de généralité aux résultats.

La mesure de stabilité adoptée est le rapport entre le plus grand makespan perturbé possible et l'horizon.

$$\sigma \triangleq \frac{\widetilde{C}_{\max}}{\lambda}$$

Ce rapport représente l'insensibilité (ou la résistance) de l'ordonnancement aux perturbations. Un ordonnancement est dit *stable* si sa stabilité est égale à 1 (i.e, $\sigma = 1$.)

Le problème consiste, finalement, à générer un ordonnancement avec la plus petite valeur possible du makespan et de stabilité.

5.3 Stabilisation par approche de tampon

L'approche de stabilisation que nous proposons repose sur l'utilisation des tampons qui sont des espaces dans lesquels on interdit l'exécution des tâches. Nous allons dans un premier temps annoncer une règle non flexible dans laquelle le tampon est soit respecté totalement soit il ne l'est pas du tout. Par la suite, nous présentons une règle plus flexible qui permet plus de souplesse dans le choix de la taille du tampon.

5.3.1 Règle de tampon

Nous adoptons ici un mécanisme d'ajustement d'exécution. Il est donc indispensable de concevoir une stratégie d'exécution dynamique. Dans cette section, nous considérons que les tâches exécutées dans un intervalle k peuvent être exécutées dans un intervalles $k' \leq k$.

Définition 5 (tampon)**.** *Le temps non utilisé d_i^k qui précède l'indisponibilité k sur le processeur i est appelé* tampon

Soit $M_i^k \triangleq \max_{j \in \pi(i,k)} b_i p_j$ la taille maximale des tâches allouées à l'intervalle k du processeur i. ($M_i^k \triangleq 0$ si $\pi(i,k) = \emptyset$)

Définition 6 (Règle de tampon)**.** *Un ordonnancement respecte la règle de tampon si et seulement si $d_i^k \geq \max(M_i^k, M_i^{k+1}, M_i^{k+2})$ pour tout intervalle k et toute machine i. Dans le cas où les indisponibilités se produisent plutôt, la contrainte se réduit à $d_i^k \geq M_i^k$.*

La technique de tampon permet d'obtenir le résultat suivant :

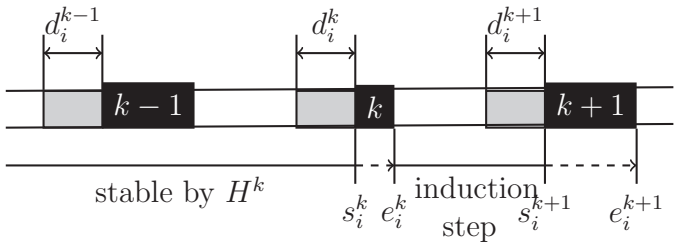


FIGURE 5.1 – Récurence sur les intervalles de la preuve de la proposition 11

Proposition 11. *Quand* $2p_{\max} \leq a_{\min}$*, tout ordonnancement qui respecte la règle de tampon est stable quand toute tâche ordonnancée dans un intervalle* k *est exécutée dans tout intrevalle* $k' \leq k$*.*

Démonstration. La démonstration est faite par récurrence sur les indices des intervalles (figure 5.1). L'hypothèse H^k est que toutes les tâches exécutées se terminent avant les dates suivantes :

1. e_i^k si l'indisponibilité k se produit avant la date prévue .
2. $s_i^k - d_i^k$ si l'indisponibilité k est en retard et que l'indisponibilité $k-1$ se termine avant $s_{k-d_i^k}$.
3. $s_i^k - d_i^k - u_i^k - d_i^{k-1}$ sinon (c.a.d, quand l'indisponibilité k est en retard et que l'indisponibilité $k-1$ se termine après $s_i^k - d_i^k$.

La figure 5.1 illustre la preuve qui se construit par récurrence sur les intervalles d'une seule machine. L'hypothèse de récurrence H^k est que toutes les tâches ordonnancées dans l'intervalle k sont exécutées avant e_i^k si l'indisponibilité k avance, et avant $s_i^k - d_i^k$ sinon.

Nous commençons par le cas élémentaire $k = 2$ puis nous généralisons pour le cas général.

Cas $k = 2$ **:** L'indisponibilité est en avance. Quand l'indisponibilité 1 se produit avant $s_i^1 - d_i^1$, le pire des cas se produit quand l'indisponibilité interrompe la plus grande tâche sur cet intervalle dont la taille est M_i^1. La règle de tampon ($d_{i\,i}^{11}$) assure par conséquence que la tâche interrompue sera re-exécutée avant e_i^1 avec toutes les tâches qui ont été initialement allouées à cet intervalle. Le même raisonnement reste valide pour l'indisponibilité 2 qui interrompe la plus grande tâche de durée M_i^2 et qui sera re-exécutée avant e_i^2.

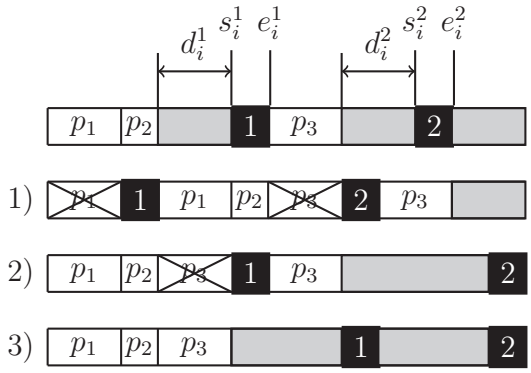


FIGURE 5.2 – Cas initial pour la proposition 11 : L'ordonnancement initial est sur la première ligne et les trois autres cas sur les lignes qui suivent

Le même raisonnement reste valide pour l'indisponibilité 2 qui interrompe la plus grande tâche de durée M_i^2 et qui sera re-exécutée avant e_i^2.

Cas 2 : L'indisponibilité 2 est en retard et l'indisponibilité 1 se terminent avant $s_i^2 - d_i^2$. Dans le pire des cas, l'indisponibilité 1 va interrompre la plus grande tâche parmis celles allouées aux deux intervalles. La règle de tampon assure que $d_i^1 \geq max(M_i^1, M_i^2)$, donc, toutes les tâches allouées à ces deux intervalles se terminent avant $s_i^2 - d_i^2$.

Cas 3 : Quand l'indisponibilité 2 est en retard et que l'indisponibilité 1 se produit après $s_i^2 - d_i^2$, toutes les tâches allouées à ces deux intervalles s'exécutent sans être interrompues. Donc, elles terminent leur exécution avant $s_i^2 - d_i^2 - u_i^1 - d_i^1$. Finalement, H^2 est prouvée.

récurrence H^k Nous supposons dans la suite que H^k est vrai pour $k > 2$ intervalles et nous montrons que cette hypothèse reste vraie pour $k+1$ intervalles. Nous considérons les 3 cas selon la date d'occurence de l'indisponibilité k (comme pour le cas $k = 2$).

Dans le premier cas 1), l'indisponibilité k se produit plutôt que prévu et toutes les tâches des k remiers intervalles sont exécutées avant e_i^k par l'hypothèse de reccurence. D'après l'algorithme d'exécution dynamique, certaines tâches de l'intervalle $k+1$ peuvent s'exécuter dans l'intervalle k (i.e, avant e_i^k). Malgrès que ces tâches peuvent être interrompues par les indisponibilités précédentes, elle vont être reexécutée dans l'intervalles $k+1$ (i.e, après e_i^k) durant lequel elle ne peuvent être interrompues que par l'indis-

ponibilité $k+1$. Nous montrons que H^{k+1} est vraie quand l'indisponibilité k est en avance en considérant que les cas où l'indisponibilité $k+1$ est soit en avance soit en retard (comme pour le cas de base de la récurernce). Dans le cas où l'indisponibilité $k+1$ avance, elle peut interrompre la plus grande tâche allouée à l'intrevalle $k+1$. Le tampon va absober cettte tâche qui va se terminer par conséquence avant e_i^{k+1}. Dans le cas d'un retard de l'indisponibilité $k+1$, les tâches ne vont pas être interrompues et vont donc se terminer avant $s_i^{k+1} - d_i^{k+1}$.

Dans le cas 2), l'indisponibilité k est en retard et l'indisponibilité $k-1$ se termine avant $s_i^{k-1} - d_i^{k-1}$ et toutes les tâches ordonnancées dans les premiers k intervalles sont exécutées avant $s_i^k - d_i^k$ par l'hypothèse de réccurence. L'espace entre $s_i^k - d_i^k$ et e_i^{k+1} contient assez d'espace tampon pour exécuter toutes les tâches alloées à l'intervalle $k+1$. Cette durée tolère la plus grande tâche a être interrompue deux fois, une première fois par l'indisponibilité k qui est en retard et une deuxième fois par l'indisponibilité $k+1$ qui est en avance puisque $d_i^k \geq M_i^{k+1}$ et $d_i^{k+1} \geq M_i^{k+1}$. Dans le cas ou l'indisponibilité $k+1$ est en retard, l'indisponibilité k se termine soit avant ou après $s_i^{k+1} - d_i^{k+1}$. Dans le premier cas, l'indisponibilité k peut interrompre la plus grande tâche ordonnancée dans l'intervalle $k+1$ et qui peut être absorbée par le tampon $d_i^k \geq M_i^{k+1}$. Ainsi, les tâches ordonnancées dans les $k+1$ premiers intervalles se terminent avant $s_i^{k+1} - d_i^{k+1}$. Dans le deuxième cas, aucune tâche ne sera interrompue dans l'intervalle $k+1$ et ces tâches se terminent avant $s_i^{k+1} - d_i^{k+1} - u_i^k - d_i^k$.

Dans le cas 3), l'indisponibilité k est en retard, l'indisponibilité $k-1$ se termine après $s_i^{k-1} - d_i^{k-1}$ et toutes les tâches des k premiers intervalles s'exécutent avant $s_i^k - d_i^k - u_i^{k-1} - d_i^{k-1}$. Les tâches de l'intervalles $k+1$ sont exécutées après cet intervalle par conséquences. Comme $2p_{\max} \leq a_{\min}$ et $\delta_i^k \leq a_i^{k+1}$, l'indisponibilité $k-2$ (pour $k \geq 3$) l'indisponibilité $k-2$ ne peut interrompre aucune de ces tâches.

Ce constat reste valide même si le décalage de la première indisponibilité qui se termine après l'horizon fait que la dernière disponibilité devienne plus courte que a_{min} parce que $2p_{\max}$ doit être inférieur ou égal à durée de la disponibilité k qui est succédée par la disponibilité $k+1$.

Si l'indisponibilité $k+1$ est en avance, alors la plus grande tâche parmis celle ordonnancées dans l'intervalle $k+1$ peuvent être interrompues par les indisponibilité $k-1$, k, ou $k+1$. Comme $d_i^{k-1} d_i^k$, d_i^k et d_i^{k+1} sont tous plus grand ou égale à M_i^{k+1}, alors il existe assez d'espace entre $s_i^k - d_i^k - u_i^{k-1} - d_i^{k-1}$ et e_i^{k+1} pour absorber ces trois interruptions.

Les deux autres cas (l'indisponibilité k interrompe la plus grande tâche de l'intervalle $k+1$ et aucune tâche de l'intervalle n'est interrompue) peuvent être traités d'une façon analogue.

Comme nous venons de le voir, dans tous les cas, la recurrence H^{k+1} est vraie quand la recurrence H^k est supposée vraie. Le cas général est alors prouvé.

Le cas ou l'indisponibilité avance a été traité dans [17]. □

La preuve ci-dessus ne requiert aucune information qui concerne la plateforme puisque chaque machine est traitée d'une façon indépendante. Par conséquent cette proposition reste valide même pour le cas des machines uniformes.

5.3.2 Règle de tampon paramétré

La règle de tampon précédente assure une stabilité de 1. Telle quel, elle n'offre aucun compromis entre l'efficacité et la stabilité.

Afin de concevoir un algorithme plus souple en terme de compromis entre ces deux critères, nous modérons la taille du tampon par un coefficient $\beta \in [0.1]$.

Définition 7. *Pour chaque intervalle et pour chaque processeur, le tampon vérifie la contrainte suivante :*

$$d_i^k \geq \beta \max(M_i^k, M_i^{k+1}, M_i^{k+2})$$

Ainsi, au lieu d'avoir une règle rigide, qui conduit à une stabilité optimale, nous aurons une règle *moins stricte*, mais, qui offre plus de compromis que la version rigide.

Afin d'obtenir de meilleures bornes que celles obtenues dans [17], pour le cas où les indisponibilités avancent, nous proposons l'utilisation de cette règle couplé avec l'algorithme 3 qui permet de sélectionner la prochaine tâche à exécuter.

L'algorithme considère en effet, lors du traitement d'un intervalle k, une liste de tâches candidates est formée par toutes les tâches non-exécutées dans les intervalles précédants (ligne 1). La tâche sélectionnée ne doit pas avoir une taille supérieure à la plus grande tâche de l'intervalle courant (ligne 6). De plus, les tâches ordonnancées dans les intervalles suivants ne doivent pas être retardées (ligne 3). Si une telle tâche n'existe pas, l'exécution des tâches

ordonnancés dans l'intervalle qui suit est avancée (ligne 8). Ainsi, si une perturbation interrompe une tâche celle-ci est immédiatement re-exécutée (parce qu'elle est la plus longue) et l'exécution des autres tâches ordonnancées dans l'intervalle courant est différée.

5.4 Analyse de la stabilité

Nous présentons dans ce paragraphe une stratégie d'exécution d'ordonnancement qui se base sur le paramétrage de la taille du tampon ainsi que sur le mécanisme d'exécution dynamique des tâches allouées à un processeur. Nous établissons une borne pour la stabilité dans le cas où les indisponibilités avancent puis, nous généralisons pour le reste des situations.

5.4.1 Algorithme d'exécution dynamique

Étant donnée une allocation généré avec un algorithme d'ordonnancement, lors de la phase d'exécution nous utilisons l'algorithme 3 pour l'ordonnancement des tâches.

Algorithm 3 Exécution dynamique sur le processeur i durant l'intervalle k à l'instant t (donc, $e_i^{k-1} \leq t \leq e_i^k$)

Input: Les tâches non exécutées des intervalles précédents J, les tâches non exécutées de l'intervalle courant J_k, la fonction d'allocation π, le temps courant t

Output: tâche j à exécuter

1: $J_c \leftarrow \{j : j \in J, b_i p_j \leq M_i^k\} \cup J_k$ {tâches candidates}
2: **if** $\pi(i, k+1) \neq \emptyset$ **then**
3: $\quad J_c \leftarrow \{j : j \in J_c, t + b_i p_j \leq e_i^k\}$
4: **end if**
5: **if** $J_c \neq \emptyset$ **then**
6: $\quad$ **return** $\arg\max_{j \in J_c} p_j$
7: **else if** $\pi(i, k+1) \neq \emptyset$ **then**
8: $\quad$ **return** $\arg\max_{j \in \pi(i,k+1)} p_j$
9: **end if**

5.4.2 Etude du cas où l'indisponibilité avance

Avant d'introduire la borne supérieure pour le cas où l'indisponibilité avance, nous montrons que le pire des cas est obtenu en considérant une instance avec un seul interval. Pour cela, nous introduisons la notion d'ordonnancement restreint à un sous-ensemble d'intervalle.

Définition 8. ***Ordonnancement restreint*** *: Un ordonnancement à un ensemble d'intervalle, est un ordonnancement dans lequel certains intervalles qui précèdent l'horizon sont ignorés. Donc, il garde la même allocation que celle de l'ordonnancement initial pour l'ensemble des intervalles considérés. Son horizon est égal à l'horizon de l'ordonnancement initial moins la somme des durées des intervalles ignorées. Le motif de disponibilité des intervalles qui succède à l'horizon est la même pour les deux ordonnancement. (initial et restreint).*

Nous annonçons alors le résultat suivant :

Lemme 1. *Supposons que* $2p_{\max} \leq a_{\min}$, $-a_i^k \leq \delta_i^k \leq a_i^{k+1}$ *et que* $\widetilde{u}_i^k = u_i^k$ *et que l'algorithme 1 est utilisé pour l'exécution. Pour tout ordonnancement, il existe un sous-ordonnancement restreint à un seul interval dont l'horizon est la date de fin de l'indisponibilité et dont la stabilité est supérieure à celle de l'ordonnancement initial.*

Démonstration. Comme les processeurs sont indépendants nous allons limiter notre étude à un seul processeur. Nous choisissons le processeur tel que la date de la fin d'exécution de sa dernière tâche soit égale au makespan perturbé $\tilde{C}_{max}$. La stabilité de cet ordonnancement est par définition exprimé comme suit

$$\sigma \triangleq \frac{\widetilde{C}_{\max}}{\lambda} = \frac{\lambda + r_1 + r_2}{\lambda} \tag{5.1}$$

où r_1 est le travail non effectué avant l'horizon à cause des interruptions et qui s'exécute après l'horizon à cause des interruption et qui s'exécute après l'horizon (ce qui reste à faire quand l'horizon est atteint), et r_2 le temps passé à recalculer les tâches que qui forment r_1 et qui ont dues être redémarrées à cause des interruption qui se produisent après l'horizon.

Nous considérons un ordonnancement à $k+1$ intervalles dont la stabilité est σ^{k+1}. D'une part nous considérons un premier sous-ordonnancement formé par les $k+1$ intervalles sauf un qui est ignoré. Son horizon est égal à l'horizon de l'ordonnancement initial moins la durée de l'intervalle ignoré.

D'autre part, nous considérons le sous ordonnancement formé par cet intervalle. Sa date de début est la date de début de la disponibilité et son horizon est la durée de l'intervalle.

Nous notons σ^k et σ^{courrant} leurs stabilités respectives. Par construction, les stabilités sont cumulatives *i.e.*,

$$\lambda^{k+1} \triangleq \lambda^k + \lambda^{\text{courant}}$$

La quantité de travail qui n'est pas effectuée avant l'indisponibilité est la somme de travaux interrompues moins l'espace libre (vide). L'espace libre est cumulatif. De plus, l'algorithme dynamique d'exécution assure qu'aucune tâche plus grande que la plus grande tâche de l'intervalle courant n'est exécutée. La somme des travaux interrompues est par conséquence cumulative parce que dans le pire cas, la plus grande tâche est interrompue. Il en résulte que

$$r_1^{k+1} = r_1^k + r_1^{\text{courant}}$$

.

Finalement, recalculer deux quantités ensemble permet d'exécuter les tâches sur un seul intervalle plus facilement. De plus, à cause de l'utilisation de l'algorithme dynamique, la durée des tâches recalculée est plus petite que celles initialement allouée à l'ordonnancement initial. Donc, $r_2^{k+1} \leq r_2^k + r_2^{\text{courant}}$.

$$\begin{aligned}
\sigma^{k+1} &\leq \frac{\lambda^k + \lambda^{\text{courant}} + R_1^k + R_1^{\text{courant}} + R_2^k + R_2^{\text{courant}}}{\lambda^k + \lambda^{\text{courant}}} \\
&\leq \frac{\sigma^{\text{courant}}\lambda^k + \sigma^k\lambda^{\text{courant}}}{\lambda^k + \lambda^{\text{courant}}} \\
&\leq \theta\sigma^k + (1-\theta)\sigma^{\text{courant}}
\end{aligned}$$

La dernière ligne est obtenue en considérant que $\theta \triangleq \frac{\lambda^k}{\lambda^k + \lambda^{\text{courant}}}$. ainsi, σ^{k+1} est inférieure ou égale à la somme pondérée de σ^k et σ^{courant}. Par conséquence, la plus grande stabilité est obtenue avec un seul intervalle (si $\sigma^{\text{courant}} \geq \sigma^{k+1}$, Le résultat est évident, sinon, les mêmes arguments peuvent être utilisé itérativement à σ^k). □

Nous rappelons que le ratio d'indisponibilité γ caractérise le plus grand rapport entre la taille de l'indisponibilité et la taille de l'indisponibilité qui

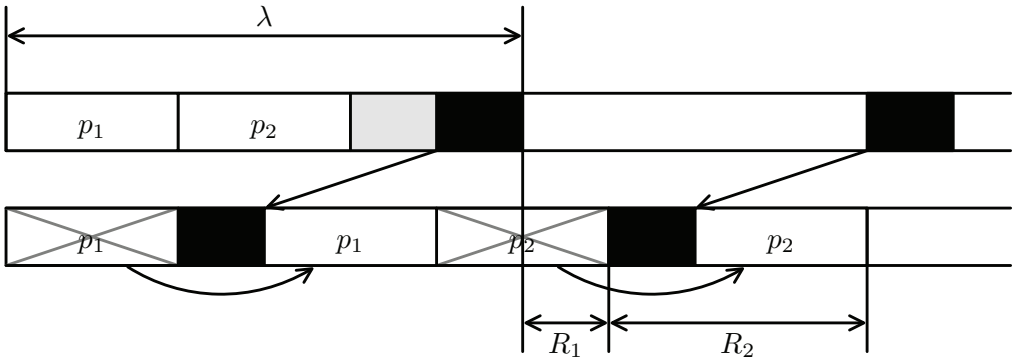


FIGURE 5.3 – Première ligne : ordonnancement avec deux tâches. Seconde ligne : le pire cas d'exécution de cet ordonnancement.

la précède (définition 2.1, page 35). Il exprime le plus grand pourcentage pendant lequel une machine reste indisponible par rapport à la taille de sa disponibilité précédente.

Théoreme 3. *Avec l'hypothèse* $2p_{\max} \leq a_{\min}$, $-a_i^k \leq \delta_i^k \leq 0$, $\widetilde{u}_i^k = u_i^k$ *et* $\beta \neq 1$, *la stabilité de tout ordonnancement, quand les indisponibilités surviennent plutôt que prévu, est majorée par*

$$\sigma \leq 2 - \frac{2\beta}{2+\beta} + \gamma$$

et cette borne est atteinte.

Démonstration. D'après lemme 1, il est possible de limiter l'analyse du pire cas de stabilité à un seul intervalle.

Une instance du pire cas contient deux tâches de taille respectives p_{max} et αp_{max}. Le reste de l'intervalle consiste en un tampon de taille βp_{max}. La taille de l'indisponibilité est arbitrairement petite.

Pour cette instance, l'horizon λ correspond exactement à la fin de l'intervalle. Enfin, la taille de la disponibilité suivante est a_{min}, et qui est suivie d'une indisponibilité de taille maximale u_{max}.

Cette instance représente un pire cas pour les raisons suivantes : Il y a toujours de l'espace pour ordonnancer la tâche de durée p_{max}. En plus, le pire cas se produit quand cette tâche est interrompue. Le reste du travail consiste en une seule tâche. En effet, s'il existe plusieurs tâches dont la durée

est inférieure à p_{max}, elles peuvent être remplacées par une tâche unique et de même durée, ce qui peut détériorer la stabilité si elle est interrompue. Finalement, le tampon est suffisamment large pour satisfaire le cas où les indisponibilités avancent puisque $M_i^1 = p_{\max}$.

Rappelons que la stabilité est donnée par l'équation 5.1.

$$\sigma \triangleq \frac{\widetilde{C}_{\max}}{\lambda} = \frac{\lambda + R_1 + R_2}{\lambda}$$

L'horizon de l'instance du pire cas est

$$\lambda = (1 + \alpha + \beta)p_{\max}$$

Dans le scenario le plus pessimiste, la tâche de durée maximale est interrompue juste avant la fin. Comme c'est l'algorithme 3 qui est utilisé pour réagir aux interruptions, cette tâche sera ré-exécutée immédiatement après l'interruption. De plus, comme $2p_{\max} \leq a_{\min}$, il n'y aura pas d'autres interruptions. Puisque la tâche de durée $\alpha p_{\max}$ doit être retardée, une partie de cette tâche devra être exécutée après l'horizon $R_1 = (1 - \beta)p_{\max}$ puisqu'une partie du tampon est utilisée pour l'exécution.

Cette tâche peut être interrompue par la première indisponibilité qui suit l'horizon. Par conséquence, $R_2 = u_{\max} + \alpha p_{\max}$. Le pire cas pour la stabilité est alors,

$$\begin{aligned}\sigma &= 1 + \frac{(1-\beta)p_{\max} + u_{\max} + \alpha p_{\max}}{(1+\alpha+\beta)p_{\max}} \\ &= 1 + \frac{1 - \beta + \frac{u_{\max}}{p_{\max}} + \alpha}{1 + \alpha + \beta}\end{aligned}$$

D'autre part, le ratio d'indisponibilité vérifie

$$a_{\min} \leq \lambda = (1 + \alpha + \beta)p_{\max}$$

Donc $\frac{u_{\max}}{a_{\min}} \geq \frac{u_{\max}}{(1+\alpha+\beta)p_{\max}}$ et $\gamma(1 + \alpha + \beta) \geq \frac{u_{\max}}{p_{\max}}$.

$$\begin{aligned}\sigma &\leq 1 + \frac{1 - \beta + \alpha}{1 + \alpha + \beta} + \gamma \\ &\leq 1 + \frac{2 - \beta}{2 + \alpha} + \gamma\end{aligned}$$

La dernière ligne est obtenue en mettant α à 1, ce qui maximise la stabilité. Cette borne est atteinte quand $a_{\min} = \lambda$. □

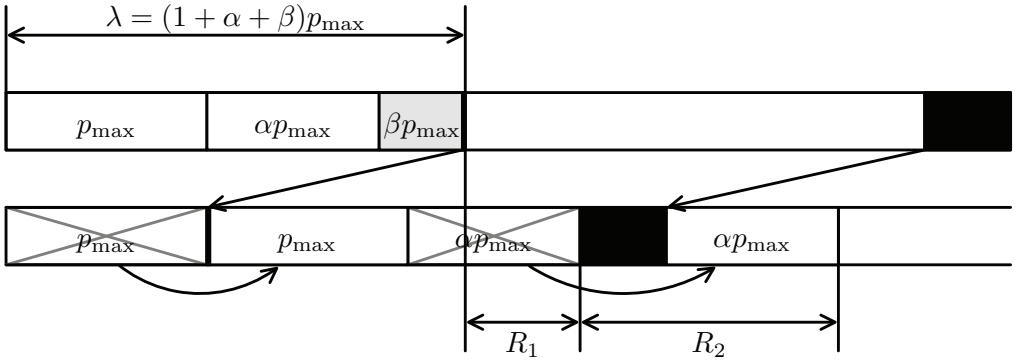


FIGURE 5.4 – Première ligne : le pire cas pour l'ordonnancement avec un intervalle unique quand l'indisponibilité avance. Seconde ligne : l'ordonnancement produit au pire cas

5.4.3 Analyse du cas général

Lemme 2. *Avec l'hypothèse* $2p_{\max} \leq a_{\min}$, $-a_i^k \leq \delta_i^k \leq a_i^{k+1}$ *et* $\widetilde{u}_i^k = u_i^k$, *la stabilité de tout ordonnancement qui respecte la contrainte de tampon paramétrée peut être reproduite en considérant uniquement deux intervalles.*

Démonstration. la preuve ressemble à la preuve du lemme 1 sauf qu'elle considère $k + 2$ intervalles. Avec cette configuration, la pire stabilité est atteinte quand une tâche de durée $p_{\max}$ est ordonnancée dans chaque intervalle de disponibilité.

L'algorithme 3 est utilisé pour l'exécution dynamique, donc une tâche peut être interrompue plus que deux fois, par une succession de deux indisponibilité perturbée dont la première est retardée.

Il existe un pire cas dans lequel les interruptions dues aux indisponibilités postérieures à l'horizon peuvent interrompre n'importe quelle tâche au plus deux fois : toute tâche qui est interrompue plus, peut échanger sa première interruption (qui se produit à cause d'un retard) avec une tâche de durée plus longue allouée à l'intervalle précédent. Une telle tâche existe puisque nous utilisons l'algorithme 3 comme algorithme d'exécution dynamique.

Toutefois, la première interruption d'une tâche, qui est interrompue deux fois, n'est pas nécessairement dûe à un retard. Une deuxième interruption de la tâche reste toujours possible. Par conséquence, r_2^{k+2} correspond aux tâches qui sont exécutées deux fois au pire cas.

Considérons maintenant un sous-ordonnancement composé par deux intervalles : le premier contient une tâche qui est interrompue deux fois par les indisponibilités qui succèdent l'horizon et un autre qui contient des tâches qui ne sont pas interrompues par ces indisponibilités. Ces deux intervalles sont décalés pour commencer au début de l'ordonnancement. Le reste des intervalles est compacté pour former un deuxième sous-intervalle.

Comme il existe une tâche de taille $p_{\max}$ dans chaque disponibilité, la quantité de travail qui reste à ré-exécuter est maximale quand la tâche de taille maximale est interrompue. Nous obtenons alors la même relation entre r_1^{k+2}, r_1^k et r_1^2 que celle du lemme 1.

$$r_1^{k+2} = r_1^k + r_1^2$$

le même raisonnement conduit aussi à la relation

$$\lambda^{k+2} = \lambda^k + \lambda^2$$

dans le pire cas, r_2^k et r_2^2 sont cumulatifs, donc nous obtenons :

$$r_2^{k+1} \leq r_2^k + r_2^{\text{courant}}$$

en effet, r_2^2 est le temps passé à calculer deux fois la tâche qui a été re-exécutée deux fois dans l'ordonnancement original.

Finalement, le scenario qui produit la pire stabilité peut être conçu avec deux intervalles au plus.

□

Théoreme 4. *Avec l'hypothèse* $2p_{\max} \leq a_{\min}$, $-a_i^k \leq \delta_i^k \leq a_i^{k+1}$, $\widetilde{u}_i^k = u_i^k$ *et* $\beta \neq 1$, *la stabilité de tout ordonnancement qui respecte la contrainte de tampon paramétré est majorée par*

$$\sigma \leq 2 - \frac{3\beta}{4+\beta} + \gamma$$

Démonstration. Grâce au lemme 2, il suffit d'analyser le pire cas obtenu avec deux intervalles.

Le cas d'un intervalle unique est traité par théorème 3. En effet, pour ce cas, les indisponibilités ne peuvent interrompre des tâches que quand elles avancent. Le cas du retard n'a aucun impact sur l'ordonnancement.

En considérant deux intervalles, il est possible de construire le pire cas, dans lequel l'indisponibilité qui se termine en même temps que quand l'horizon avance. En effet, la première indisponibilité interrompe la plus grande tâche qui se termine soit avant soit après. Ainsi, il reste suffisamment d'espace pour exécuter une tâche avant la dernière indisponibilité qui peut alors interrompre cette dernière tâche (à cause de l'utilisation de l'algorithme 3, l'exécution de la plus grande tâche est favorisée).

Nous supposons que la durée des tâches est p_{max} et $\alpha_1 p_{\max}$ pour la première disponibilité et p_{max} et $\alpha_2 p_{\max}$ pour la deuxième disponibilité. Le travail qui ne peut donc pas être exécuté avant l'horizon est donc $R_1 = 2(1-\beta)p_{\max}$. En supposant que la plus grande tâche est interrompue deux fois après l'horizon, on a

$$R_2 = 2(u_{\max} + \max(\alpha_1, \alpha_2)p_{\max})$$

.

Par conséquence, la pire stabilité est

$$\begin{aligned}
\sigma &= 1 + \frac{2(1-\beta)p_{\max} + 2(u_{\max} + \max(\alpha_1, \alpha_2)p_{\max})}{(2+\alpha_1+\alpha_2+2\beta)p_{\max}} \\
&\leq 1 + \frac{2(1-\beta) + \max(\alpha_1, \alpha_2)}{2+\alpha_1+\alpha_2+2\beta} + \gamma \\
&\leq 1 + \frac{4-2\beta}{4+\beta} + \gamma
\end{aligned}$$

La deuxième ligne est obtenue en initialisant l'une des valeurs de α à la plus petite valeur $(1-\beta)$ et l'autre à la plus grande valeur (1).

Quand $\gamma = 0$ et $\beta > 0.5$, cette borne est atteinte pour des valeurs de $\alpha_1 = 1-\beta$ et $\alpha_2 = 1$. Si $\beta \leq 0.5$, alors il existe une seule interruption après l'horizon. □

Corollaire 1. *La stabilité de tout ordonnancement est majorée par* $2+\gamma$.

5.5 Algorithmes bi-objectifs pour l'ordonnancement avec contraintes d'indisponibilités

Dans cette section nous proposons une large variété d'heuristiques pour le problème de minimisation du makespan et de maximisation de la stabilité.

Cette variété consiste à utiliser plusieurs techniques issue du domaine de la recherche opérationnelle et sur la technique de tampon paramétré. Ainsi, toutes les heuristiques que nous proposons seront paramétré par une facteur β qui consiste qui représente le compromis entre a stabilité et la performance souhaité.

Rappelons que M_i^k et S_i^k représentent respectivement la taille de la plus grande tâche allouée à la disponibilité de l'intervalle k du processeur i et la somme de toutes les tailles de tâche allouée à ce même intervalle.

Comme toutes les variétés proposées respectent la contrainte du tampon paramétré, nous définissons la procédure *getslack()* qui permet de renvoyer la taille nécessaire du tampon pour respecter la contrainte. Le principe de la procédure est d'initialiser la valeur du tampon à $b_i p_j$ puis d'ajuster cette valeur en fonction des allocations faites aux intervalles précédant et suivant.

Algorithm 4 Procédure getSlack($b_i p_j$, k)

Input: La plus longue durée de tâche de l'intervalle k sur le processeur i
Output: un tampon valide

1: $d \leftarrow b_i p_j$
2: **if** $M_i^{k-1} \neq 0$ **then**
3: **if** $b_i p_j > M_i^{k-1}$ **then** {le tampon précédent est insuffisant}
4: $d \leftarrow d + b_i p_j - M_i^{k-1}$
5: **end if**
6: **end if**
7: **if** $M_i^{k+1} > b_i p_j$ **then** {le tampon suivant est insuffisant}
8: $d \leftarrow d + M_i^{k+1} - b_i p_j$
9: **end if**
10: **return** βd

Dans l'algorithme 4, si l'intervalle précédent n'a pas encore été considéré pour l'allocation (ligne 2), alors, le calcul du tampon est retardé puisque la longueur de la plus grande tâche ne peut pas être connue.

5.5.1 Algorithme glouton

Notre premier algorithme proposé, présenté dans la figure

Cet algorithme considère toutes les tâches disponibles et pour chacune d'entre elles, il cherche de meilleur date de début d'exécution (boucle de la ligne 5).

Pour chaque processeur, les disponibilités sont considérées itérativement jusqu'à ce que l'on trouve la première disponibilité qui permet l'exécution de la tâche en respectant la contrainte du tampon. La tâche est finalement affectée à la disponibilité qui donne la meilleure date de fin de tâche.

A la fin de l'ordonnancement, les tâches de chaque disponibilité sont triées selon l'ordre décroissant de durée de tâches.

Algorithm 5 Continuous Greedy (CG)

Input: un ensemble de n tâches, un paramètre β, un ensemble de machines
Output: une fonction d'allocation π

1: Trier les tâches par coût décroissant (croissant dans la cas de SPT)
2: **for all** $j \in [1..n]$ **do** {Considérer les tâches dans l'ordre de tri}
3: **for all** $i \in [1..m]$ **do** {considérer chaque processeur i candidat pour l'allocation}
4: $k \leftarrow 1$
5: **while** $S_i^k + b_i p_j + \text{getSlack}(\max(b_i p_j, M_i^k), k) > a_i^k$ **do** {Chercher la première disponibilité qui respecte la contrainte de tampon}
6: $k \leftarrow k + 1$
7: **end while**
8: $E_i \leftarrow e_i^{k-1} + u_i^k + S_i^k + b_i p_j + \text{getSlack}(\max(b_i p_j, M_i^k), k)$ {Meilleur date de fin de la tâche j sur le processeur i}
9: $K_i \leftarrow k$
10: **end for**
11: $I \leftarrow \arg\min_{i \in [1..m]} E_i$
12: $\pi(I, K_I) \leftarrow \pi(I, K_I) \cup \{j\}$ {Ordonnancer la tâche j dans l'intervalle K_I du processeur I}
13: **end for**

5.5.2 Heuristiques basées sur les intervalles

Nous présentons au début de cette section quelques stratégies générales dans lesquels les intervalles sont assimilés à des sacs qui sont considérés successivement pour l'allocation des tâches. Ces stratégies sont utilisés par la suite à l'intérieur d'autres procédures de plus haut niveau qui indiquent l'ordre avec lequel les intervalles vont être traités (visités).

Mécanisme de sélection des tâches

Étant donné un intervalle k d'une machine i et un ensemble de tâches non-ordonnancées. Le mécanisme de sélection permet de choisir une politique qui permet de sélectionner les tâches à ordonnancer dans chaque intervalle.

Les deux premières heuristiques que nous proposons sont issues du problème du sac-à-dos, à savoir, FFD et FFI. Avec l'algorithme FFD, les tâches sont triés selon l'ordre décroissant de durée (ou de coût). Les tâches sont ensuite alloué au premier sac qui les rencontre et dont l'espace vide est supérieur à la taille de la tâche en question. Ainsi, les plus longues tâches seront toujours affectées au premier intervalle considéré. Avec FFI, c'est l'ordre inverse des tâches qui est considéré.

Algorithm 6 First Fit Decreasing (FFD)

Input: un ensemble de n tâches, un paramètre β, un ensemble trié de disponibilités A

Output: une fonction d'allocation π

1: trier les tâches par ordre décroissant de coût (croissant dans le cas de FFI)
2: **for all** $j \in [1..n]$ **do** {Considérer les tâches dans l'ordre de tri}
3: **for all** $a_i^k \in A$ **do** {Considérer chaque disponibilité k sur la machine i dans l'ordre}
4: **if** $S_i^k + b_i p_j + \text{getSlack}(\max(b_i p_j, M_i^k), k) \leq a_i^k$ **then**
5: $\pi(i,k) \leftarrow \pi(i,k) \cup \{j\}$
6: break
7: **end if**
8: **end for**
9: **end for**

L'alternative la plus proche à ces deux algorithmes consiste à utiliser LPT et SPT. La différence majeure entre ces deux stratégies par rapport à FFD et FFI, est que les premières tentent de trouver la meilleure affectation à la tâche courante, alors que les deuxièmes cherchent à maximiser la somme des tâches à affecter à l'intervalle courant. Par conséquence, avec LPT, les premières plus grandes tâches sont distribuées sur plusieurs intervalles de disponibilité.

Le dernier mécanisme de sélection de tâche que nous proposons est basé sur la programmation dynamique. Les étapes principales de cet algorithme

sont :

1. Trier les tâches par ordre croissant de durée (ou de coût)
2. Résoudre le problème de sac à dos sans contrainte de tampon
3. Ajouter la contrainte de tampon et utiliser le résultat précédent

Comme les tâches sont triées par ordre croissant de durée la tâche qui sera supprimée afin d'ajouter la contrainte de tampon est désormais la dernière et elle est la plus longue.

Schéma d'ordonnancement

Le premier algorithme que nous proposons est appelé Interval Greedy. Il s'agit d'un algorithme glouton qui visite les intervalles par ordre croissant de date de fin. Pour chaque intervalle visité, il est possible d'utiliser cinq variantes présentées ci-dessus afin de sélectionner l'ensemble des tâches à allouer à la disponibilité courante.

La deuxième approche que nous proposons est basée sur la technique de l'approximation duale. Étant donné un horizon λ, l'algorithme considère tous les intervalles dont la date est antérieur à λ. Certains intervalles qui intersectent λ sont tronqués.

L'ensemble des valeurs que prend λ est défini par une procédure de recherche dichotomique. Ainsi, une borne supérieure de l'horizon u est initialisée à la somme de toutes les durées des tâches. La borne inférieur l est initialisée à 0. Nous entamons par la suite la boucle qui fait la recherche dichotomique : λ est initialisée à la moitié de la somme de u et de l. Par la suite, si toutes les tâches rentrent avant λ alors on met à jour la valeur de u à λ sinon, c'est la valeur de l qui est mise à λ.

Pour chaque horizon, les disponibilités sont remplies selon l'ordre croissant de durée (ligne 6). Notre intuition suggère qu'il est plus facile de remplir les petits intervalles d'abord.

5.5.3 Résumés des heuristiques conçues

La figure 5.5 résume l'ensemble des heuristiques conçues.

Algorithm 7 Programmation dynamique (DP)

Input: un sous-ensemble de tâches J, un paramètre β, un processeur i, les plus grande tâches allouées aux intervalles précédents et suivants (M^{k-1} et M^{k+1})

Output: l'ensemble de tâches allouées $\pi(i,k)$

1: Trier les indexes des tâches par ordre croisant de coût de tâches
2: Initialiser chaque élément des matrices P et P_s à 0
3: **for all** $j \in J$ **do** {Considérer les tâches selon l'ordre de tri}
4: **for all** $l \in [1..a_i^k]$ **do** {Considérer un sous-intervalle}
5: **if** $b_i p_j \leq l$ **then** {tester si la tâche peut être allouée}
6: $P[j][l] \leftarrow \max(P[j-1][l],$
$P[j-1][l - b_i p_j] + b_i p_j)$
7: $d \leftarrow \text{getSlack}(b_i p_j, k)$
8: $P_s[j][l] \leftarrow \max(P_s[j-1][l],$
$P[j-1][l - b_i p_j - \lceil d \rceil] + b_i p_j)$
9: **end if**
10: **end for**
11: **end for**
12: $l \leftarrow a_i^k$ {Initialiser la phase de chainage-arrière de la soution}
13: $s \leftarrow$ **true** {Mémoriser la matrices des espaces tampons}
14: **for all** $j \in J$ **do** {Considérer les tâches dans l'ordre inverse}
15: **if** $(P_s[j][l] \neq P_s[j-1][l]$ **or not** $s)$ **and**
$(P[j][l] \neq P[j-1][l]$ **or** $s)$ **then**
16: $\pi(i,k) \leftarrow \pi(i,k) \cup \{j\}$
17: **if** s **then**
18: $d \leftarrow \text{getSlack}(b_i p_j, k)$
19: $l \leftarrow l - b_i p_j - \lceil d \rceil$
20: $s \leftarrow$ **false** {Basculer à l'autre matrice}
21: **else**
22: $l \leftarrow l - b_i p_j$
23: **end if**
24: **end if**
25: **end for**

Algorithm 8 Interval Greedy (IG)

Input: un ensemble de processeurs
Output: un ensemble ordonné d'indisponibilité A

1: Trier les intervalles selon leurs date de fin
2: **return** Les disponibilité des intervalles selon l'ordre donné par le trie

Algorithm 9 Dual Approximation (DA)

Input: un ensemble de n tâches, un paramètre β, un ensemble de machines
Output: une fonction d'allocation π

1: $u \leftarrow$ la borne supérieure de l'horizon λ
2: $l \leftarrow 0$
3: **while** $u - l > 1$ **do**
4: $\lambda \leftarrow \frac{u+l}{2}$
5: $A \leftarrow \{a_i^k : e_i^k \leq \lambda\} \cup \{a_i^k - (e_i^k - \lambda) : \lambda < e_i^k \wedge e_i^k \leq a_i^k + \lambda\}$ {Couper le dernier intervalle}
6: trier A par durée décroissante
7: faire appel à la stratégie d'ordonnancement voulu pour obtenir π
8: **if** $\bigcup_{i,k} \pi(i,k) \neq [1..n]$ **then**
9: $u \leftarrow \lambda$
10: **else**
11: $l \leftarrow \lambda$
12: **end if**
13: **end while**
14: **return**

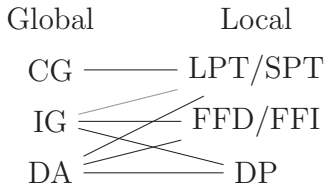


FIGURE 5.5 – Schéma récapitulatif des combinaisons d'heuristiques possibles

5.6 Conclusion

Nous avons dans ce chapitre étudié le problème d'ordonnancement avec contraintes d'indisponibilité et d'incertitude de ressources. Ce genre de problème est issu de l'émergence de nouvelles plateformes de calcul distribué éparpillé à travers le globe.

Après avoir étudié dans les chapitres précédents des cas particulier du problème (une seule indisponibilité par processeur, un seul modèle de perturbation), nous sommes passé dans ce chapitre à l'étude du cas général. En effet, l'étude porte sur des instances avec plusieurs indisponibilités par processeur et avec un modèle de perturbation permettant aux indisponibilité de se produire avant ou après la date prévue.

Nous avons dans un premier temps proposé un algorithme d'exécution dynamique qui permet de réagir aux éventuelles perturbations qui causerait des interruptions de tâches et nous avons prouvé les garanties de stabilité. Étant donnée que nous adoptons une approche proactive, nous avons conçu un ensemble d'heuristiques plus ou moins couteuse (en terme de complexité temporelle) afin de concevoir un ordonnancement de référence.

Nous allons dans le chapitre qui suit, expérimenter les différentes approches conçues et déterminer les paramètres qui permettent la meilleur exploitation possible des approches présentées.

Chapitre 6

Simulations

6.1 Introduction

Les chapitres précédents ont été consacrés à la conception d'algorithmes paramétrables offrant un compromis entre la performance et la stabilité. Nous avons montré que ces critères sont contradictoires et que trouver une solution à ce problème demeure une tâche difficile. Le paramètre (noté β) dans ces algorithmes représente le compromis souhaité entre les deux critères.

Nous disposons ainsi de 12 variantes d'algorithmes d'ordonnancement.

Bien que l'aspect théorique de ces algorithmes permet d'offrir une variété de choix pour les paramètres de l'ordonnancement, et en particulier pour la valeur du compromis. La détermination d'une valeur qui réalise une bonne performance reste notre principal but.

Dans ce chapitre, nous reportons les expérimentations qui ont été réalisées afin d'atteindre cet objectif. Nous commençons par évaluer le comportement de l'algorithme GAPS qui a été conçu dans le chapitre 4 et nous déterminons la meilleure valeur du compromis. Nous passons ensuite à la campagne de tests effectuées afin de déterminer, parmi toutes les variantes proposées dans le chapitre 5, l'ensemble d'heuristiques qui produisent des résultats satisfaisants.

6.2 Caractéristiques des instances

La bonne exploitation des ressources offertes par les grilles de calcul est un objectif commun à tous ses exploitants. Afin de mieux rentabiliser l'utili-

sation, les chercheurs se sont concentrés sur le comportement des applications soumises sur les grilles ainsi que sur le comportement des ressources. Plusieurs projets de récolte de traces ont été mis en place [52]. De plus, plusieurs travaux ont visé l'étude du comportement des applications sur ces plateformes gigantesques[27].

6.2.1 Workload

Plusieurs études ont été effectuées afin de mieux caractériser les durées des tâches dans les environnements de grilles de calcul. Il est évident que les types de tâches vont varier selon le type de la grille et la communauté qui soumet les applications.

Dans le cadre du projet BOINC, un résumé des caractéristiques liées aux différents projets a permis de connaitre paticulièrement la taille des tâches. Nous avons remarqué un grand écart entre ces tailles selon le projet considéré, qui va de 30 minutes à plusieurs jours voire mois.

Dans le cadre de ce travail, les traces des tâches ont été collectées depuis le projet Docking@home[77] (celles-ci ont été fournies par Michela Taufer). Ces traces ont reporté le temps d'exécution de plus de 150.000 tâches.

Nous avons dans la figure 6.2.1 dessiné l'évolution des durée des tâches et nous avons remarqué que plus de 75% des tâches ont une durée inférieure à 24h et que la moyenne des durées des tâches est autour de 40 minutes.

Une première analyse des traces montre la présence de tâches de courte durée allant de quelques secondes à quelques minutes. Dans la pratique, elles correspondent aux tâches qui ont été interrompues durant leurs exécutions. Nous avons éliminé 382 tâches dont la durée est inférieure à 30 minutes. Nous considérons en effet que cette durée est une borne inférieure raisonnable pour la taille des tâches dans un environnement de grille de calcul [1].

6.2.2 Plateforme cible

Les traces des disponibilités des processeurs ont été collectées à partir du projet BOINC [8] entre le 1er avril 2007 et le 12 Février 2008. Cette opération a pu être effectuée grâce au déploiement du client BOINC sur $112,268$ hôtes du projet SETI@home[9]. Pour chaque processeur, les traces fournissent les

1. statistique reporté à l'adresse suivante
`http://www.boinc-wiki.info/Catalog\_of\_BOINC\_Powered\_Projects`.

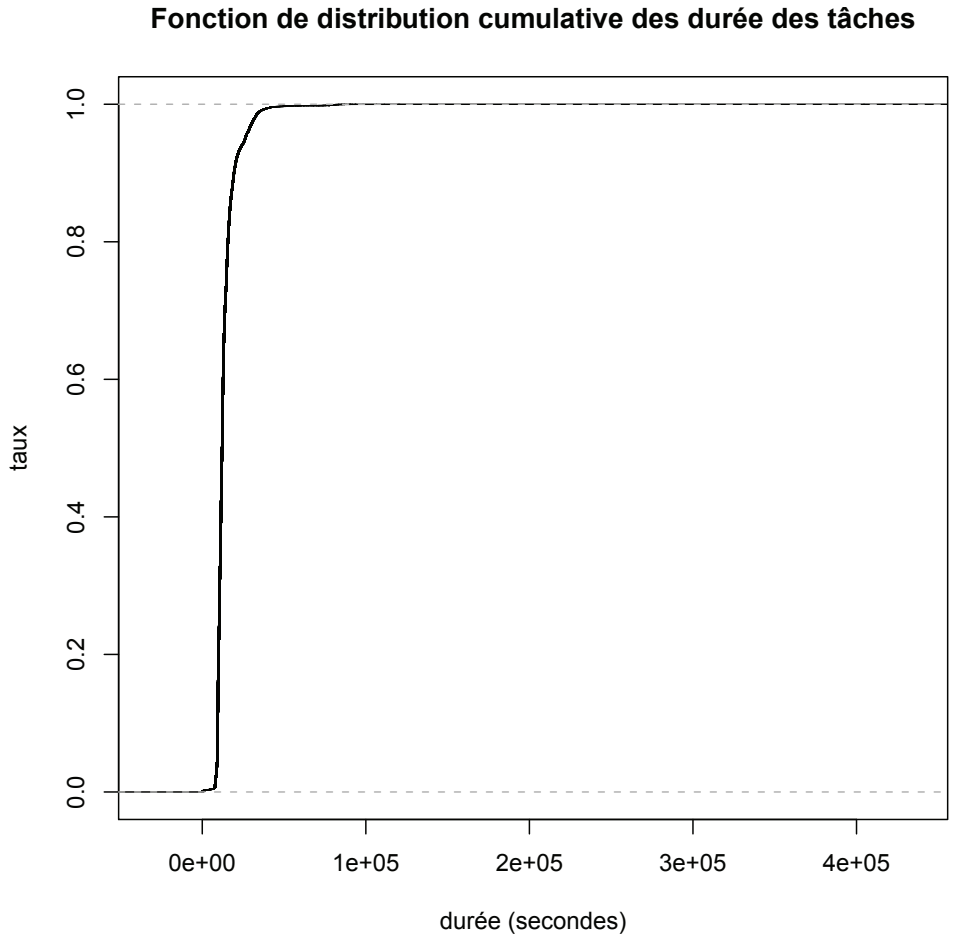


FIGURE 6.1 – Courbe de la fonction de distribution cumulative des durées des tâches

dates de début et de fin des périodes de disponibilité. Ces traces représentent $16,293$ années cumulées de disponibilité de processeurs. 81% de la disponibilité provient des PC personnels, 17% provient des PC situés au travail et 2% sont issues de PC situés dans des institutions académiques.

L'étude détaillée de ces traces a été faite dans [53, 52] pour extraire les modèles de disponibilité des hôtes. Les principales caractéristiques de ces

traces sont :

- La moyenne et la médiane des durée des intervalles sont respectivement $20h$ et $8h$.
- La moyenne de disponibilité est 5.25 fois plus longue que la disponibilité de ressources des entreprises[54].
- 60% des durées des intervalles de taille moyenne est inférieures à $24h$ d'où le besoin de mécanismes de tolérances ou de prévoyance aux pannes.
- plus que 40% des hôtes sont disponibles au moins 80% du temps.

6.3 Simulation de GAPS

6.3.1 Plan d'expérience

Les instances utilisées pour simuler GAPS sont composées d'un ensemble de tâches et de processeurs. Ces ensembles sont générés aléatoirement suivant une loi uniforme à partir des traces. De plus, 20% de l'ensemble des machines est libre (sans indisponibilité programmée). Cette hypothèse provient du constat reportant que 20% de l'ensemble des machines sont disponibles plus de 95% du temps [52]. Le nombre de tâches est 3000 alors que le nombre de processeur est 300.

L'entrée de l'algorithme GAPS, pour chaque simulation, est une instance et un paramètre β. Nous mesurons une borne inférieure pour le makespan, le makespan de référence et le makespan perturbé. Cette dernière valeur est obtenue en perturbant l'ordonnancement 30 fois et en calculant la valeur médiane. Les perturbations sont générées en utilisant une loi uniforme. Le ratio d'approximation du makespan est obtenu en divisant la valeur du makespan par celle de la borne inférieure alors que le ratio de stabilité est obtenu en divisant la valeur de makespan de l'ordonnancement perturbé par celle du makespan de l'algorithme de référence.

6.3.2 Interprétation

L'effet de β sur la performance de GAPS a été reporté dans la figure 6.2. Dans les boites verticales, la ligne en gras représente la valeur médiane. Pour faciliter la lecture, nous avons lié les médianes par une ligne.

Comme prévu, la stabilité décroit pour des valeurs de β proches de 1 alors

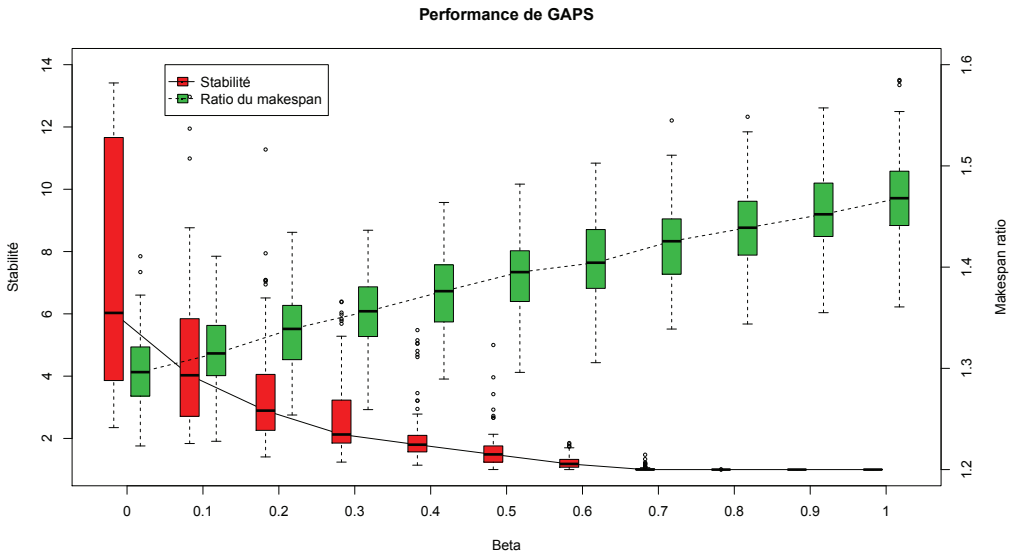


FIGURE 6.2 – Effet du paramètre β sur les ratios de stabilité et de makespan de GAPS. Chacune des 1100 mesures représentent une simulation avec 300 processeurs et 3000 tâches. Pour chacune des 11 valeurs de β, GAPS est exécuté 100 fois avec différentes instances.

que la valeur du makespan croit. Nous avons constaté aussi que pour des valeurs de β proches de 0, le makespan se dégrade d'une façon remarquable en présence de perturbations. L'augmentation de la valeur de β mène à un meilleur compromis entre la stabilité et le makespan (une dégradation de l'ordre de 20% sur la makespan). Notons aussi qu'il n'est pas nécessaire de sélectionner une valeur élevée de β pour obtenir une bonne stabilité. En effet, à partir d'une valeur β supérieure à 0.7, la stabilité devient proche de 1.

6.4 Expérimentation des algorithmes d'ordonnancement dans des environnements fortement perturbés

Dans cette section, nous effectuons des expérimentations qui permettent d'évaluer tous les algorithmes proposés dans le chapitre 5. Nous avons développé deux modules qui permettent de tester tous les algorithmes conçus.

Le premier module, intitulé *gensched*, permet de calculer l'ordonnancement qui lui sera spécifié en paramètre. Les arguments passés à ce module sont la méthode globale, la méthode locale et la valeur de β. Ce module génère l'ordonnancement conséquent.

Cet ordonnancement est ensuite perturbé avec le module *distsched* selon les algorithmes présentés dans le chapitre 5. Nous mesurons ensuite le makespan de l'ordonnancement perturbé, la date de complétion de chaque tâche et la date de complétion de la dernière tâche de chaque processeur.

Comme la migration n'est pas autorisée, il est possible que certaines tâches ayant été interrompues ne trouvent pas l'espace nécessaire pour être exécutées après avoir été perturbées. Ces tâches auront donc un temps de complétion infini.

Dans le reste de cette section, nous allons évaluer le comportement de nos heuristiques selon différents critères et nous choisissons celles qui ont réalisent le meilleur compromis. Nous notons les ordonnancements X/Y où X est la stratégie globale et Y est la stratégie locale. Par exemple, C/L signifie Continoues Greedy/LPT.

Les instances ont été extraites des traces fournies par le projet BOINC. Un ensemble de 10000 tâches a été choisi parmi l'ensemble présenté dans la section 6.2. Nous disposons aussi de 10 instances de processeurs de taille 500.

6.4.1 Performances des variantes proposées

Nous avons effectué une série d'expériences dont le but est de voir le comportement détaillé des algorithmes d'ordonnancement. Dans ce paragraphe, nous nous intéressons à la performance des différentes heuristiques conçues.

Pour cette expérience, nous avons choisi l'instance de processeur qui offre le plus de disponibilité afin de réduire le nombre de tâches perturbées avec temps de complétion infini.

Nous avons, pour chaque β allant de 0 à 1 (101 valeurs de beta), et pour chaque variante des algorithmes conçus théoriquement, exécuté l'algorithme d'ordonnancement et nous avons récupéré l'horizon et la borne inférieure. Nous avons dessiné dans les figures 6.3 et 6.4 les makespan généré puis, la performance des algorithmes qui est définie ici comme étant le rapport entre l'horizon et la borne inférieur.

Nous avons remarqué que la meilleure performance est obtenue pour l'algorithme D/D. Cependant, cette heuristique prend un temps relativement long (une vingtaine d'heures) pour générer l'ordonnancement par rapport aux heuristiques générées par Continoues Greedy et Interval Greedy(quelques minutes). En contre partie, la pire performance est donnée par les heuristiques I/L et I/S. L'écart entre ces deux variantes et le reste des heuristiques s'explique par le fait que Interval Greedy ne considère pas le dernier intervalle sur chaque machine et qu'en plus, LPT et SPT ne font pas d'optimisation à l'intérieur de chaque intervalle.

Le reste des heuristiques renvoient des performances proches mais, c'est le temps de calcul des heuristiques basées sur l'approximation duale qui fait que celles-ci sont moins pratiques que celles basées sur Continoues Greedy et Interval Greedy.

Nous avons remarqué que, à part les algorithmes I/L et I/S, l'évolution de la performance en fonction de β est faible. Par exemple, pour $\beta = 0$, la performance de C/L est 1.2 alors qu'elle est de l'ordre de 1.7 pour $\beta = 1$.

6.4.2 Stabilité

Nous analysons dans ce paragraphe la stabilité des ordonnancements conçus. Pour chaque β allant de 0 à 1 avec un pas de 0.2, nous calculons les ordonnancements des différentes variantes proposés.

Chaque ordonnancement généré est ensuite perturbé 30 fois. Nous mesurons ensuite l'horizon et la date de fin d'exécution sur chaque processeur.

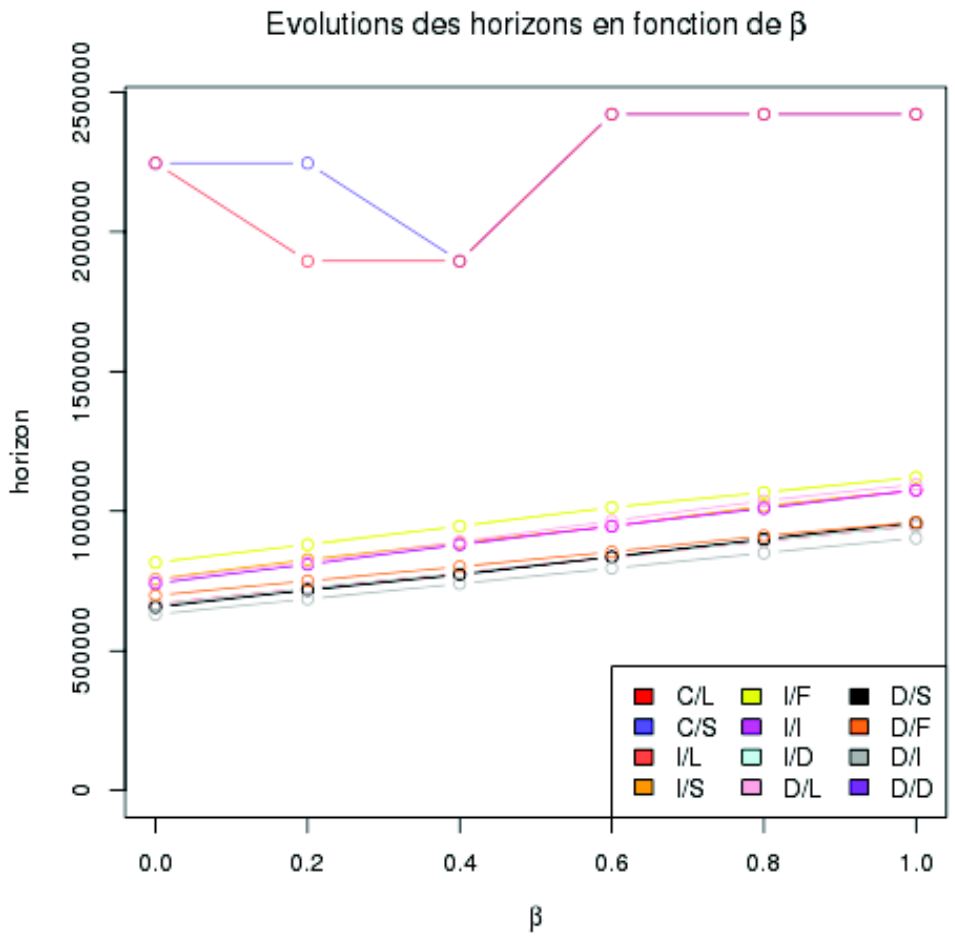


FIGURE 6.3 – Horizons des variantes conçues

Nous avons étudié l'impact des perturbations sur la stabilité des ordonnancements. Nous avons remarqué que ces ordonnancements sont très sensibles aux ordonnancement quand $\beta = \{0, 0.2\}$. En effet, comme le montre la figure 6.5 tous les ordonnancements, des 12 variantes étudiées renvoient une stabilités infinies. Dans la pratique, ce constat signifie que la disponibilité perturbée d'au moins l'un des processeurs ne permet pas l'exécution de tout la workload initialement alloué.

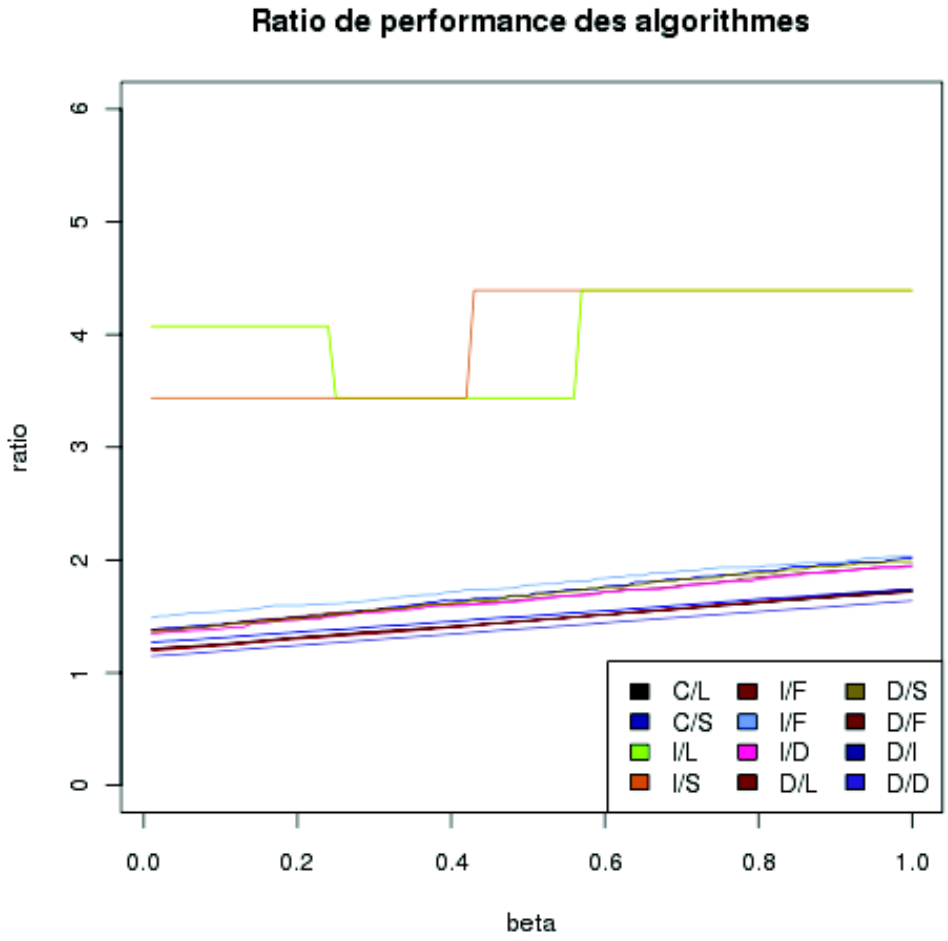


FIGURE 6.4 – Ratio de performance des variantes

En effectuant une analyse plus profonde des processeurs pathologiques, nous avons remarqué que contrairement aux intuitions, cette instabilité est plutôt liée au motif perturbé et non pas à l'espace disponible ni au nombre de tâches. Il existe dans la pratique, des motifs perturbés qui ne réussissent pas à placer certaines tâches malgré le nombre très réduit de tâches affectées initialement.

Nous avons par la suite, considéré uniquement les ordonnancements avec

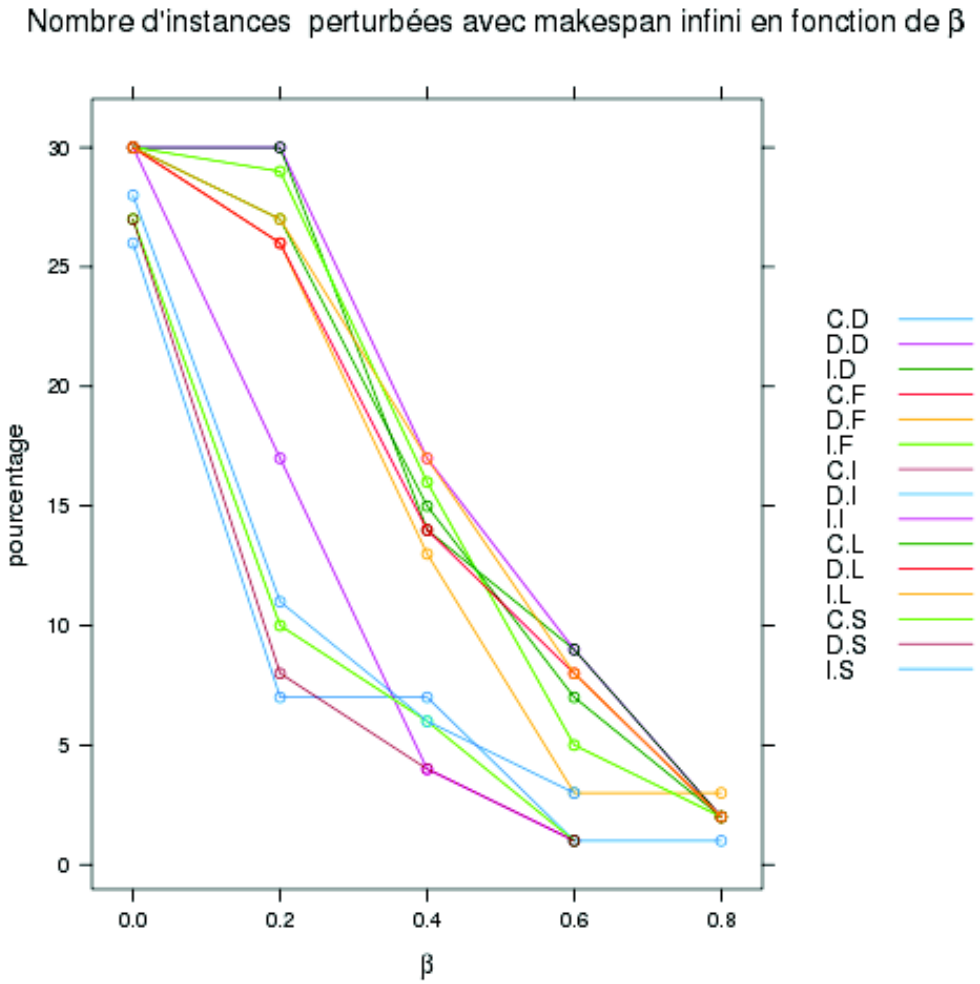


FIGURE 6.5 – Occurrences avec makespan perturbé infini

makespan perturbé fini, calculé leurs stabilités et dessiné la moyenne de la stabilité de ces ordonnancements en fonction de β. Comme nous pouvons le voir sur la figure 6.6, les ordonnancements à l'exception des ordonnancements I/S qui renvoient une stabilité proche de 2 pour $\beta = 0.2$, le reste des ordonnancements renvoient une stabilité relativement acceptable, en particulier, les ordonnancements avec méthode locale SPT.

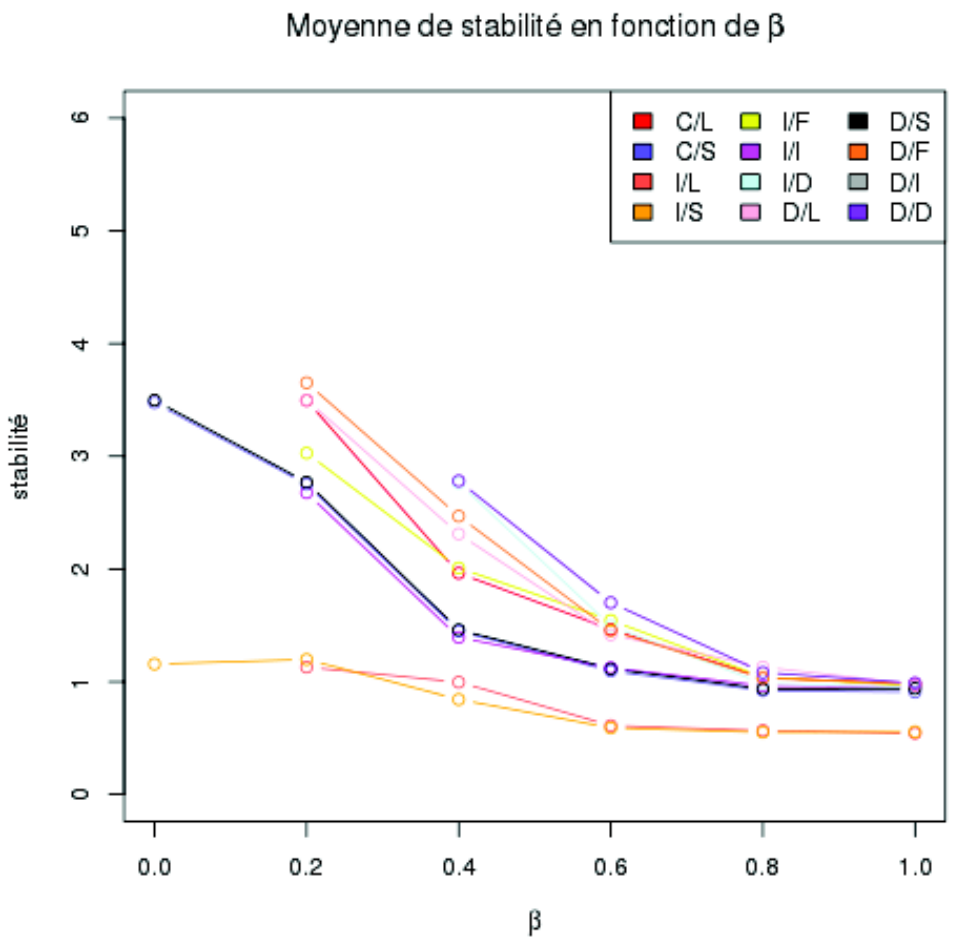


FIGURE 6.6 – Stabilité en fonction de β

6.4.3 Impact du paramètre β

Pour mieux analyser l'effet du paramètre β sur la qualité des solutions produites, nous avons repris les expériences décrites ci-dessus mais pour plus 101 valeurs de β. Ainsi, nous générons, pour chaque valeur de β allant de 0 à 1 et avec pas de 0.01, et pour chaque variante proposée, l'ordonnancement correspondant.

Chaque ordonnancement obtenu est perturbé ensuite 30 fois (la perturba-

tion est générée avec une loi uniforme). Nous renvoyons le makespan perturbé, le makespan initial et nous calculons la stabilité.

Nous dessinons dans la figure 6.7, pour chaque β, et pour chaque algorithme, le nombre de fois où la configuration perturbée est non réalisable (i.e, le makespan perturbé est infini).

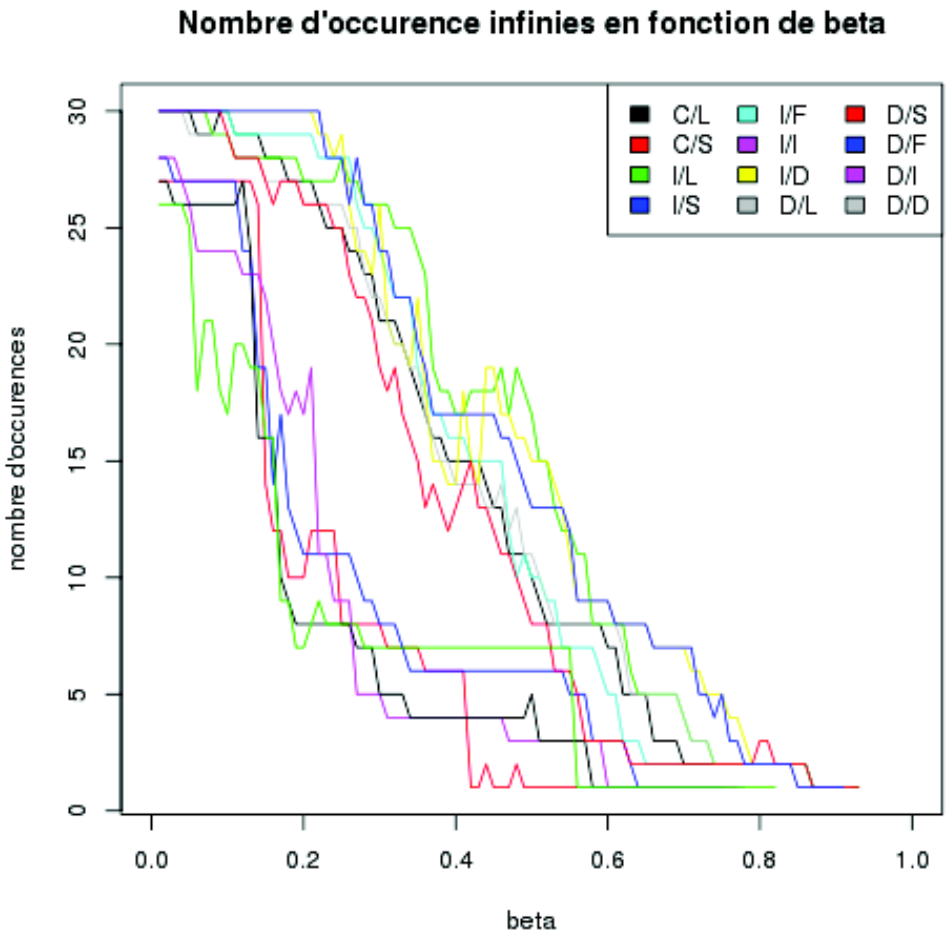


FIGURE 6.7 – Nombre d'occurrences infinies en fonction de β

Comme pour les analyses précédentes, le nombre de fois où cette configuration amène à un ordonnancement non réalisable, décroit en fonction de β.

Cette baisse est importante dans l'intervalle [0.2, 0.6].

Nous remarquons aussi que certains algorithmes sont moins sensibles que d'autres aux perturbations. Par exemple, la combinaison C/S possède le plus faible nombre d'occurrences infinies et ce nombre s'annule pour une valeur de $\beta = 0.84$. Les combinaisons I/L et I/D sont, en contre partis, les plus sensibles. Les heuristiques C/L et D/L continuent à être sensibles le plus en fonction de *beta*. Leur sensibilité infinie s'annule pour une valeur de β supérieure à 0.9.

Pour les instances avec stabilité finie, nous dessinons la figure 6.8 qui présente la variation de la stabilité (rapport entre makespan perturbé et l'horizon initial).

Nous remarquons que les heuristiques I/S et I/L offrent une meilleure stabilité que le reste des algorithmes. Cette stabilité s'explique par le fait que ces algorithmes ne considèrent pas le dernier intervalle de disponibilité sur les machines. Par conséquence, Leurs performance est mauvaise, par contre l'espace non utilisé pour l'allocation des tâche absorbe les effets des perturbations.

Parmi les heuristiques qui restent, c'est la combinaison I/I qui donne la meilleure stabilité qui varie de 3 (pour des valeurs de β proches de 0) à 1 pour de valeur de β proches de 1. La pire stabilité est par contre obtenue par D/D. Ce résultat est dû à l'optimisation que fait cette heuristique qui occupe au mieux tous intervalles de disponibilité trouvés. Par conséquent, il reste très peu de place avant l'horizon pour absorber les conséquences d'une perturbation. Il suffit donc, d'une seule longue indisponibilité pour que la stabilité soit dégradée remarquablement.

6.4.4 Caractérisation de la sensibilité des processeurs

Une analyse des traces de l'exécution révèle que le makespan infini est dû au fait que certains processeurs ne réussissent pas à re-exécuter toutes les tâches qui lui ont été initialement allouées.

Dans ce paragraphe, nous effectuons une analyse profonde des traces afin de déterminer si certains processeurs sont potentiellement plus sensibles que d'autres, et dans le cas échéant, quelle est la cause de cette sensibilité. Nous prenons comme exemple d'instance, les perturbations de l'ordonnacement C/L pour une valeur de $\beta = 0$.

Pour chaque ordonnancement perturbé, nous avons enregistré la date de fin d'exécution de la dernière tâche sur tous les processeurs utilisés. Nous

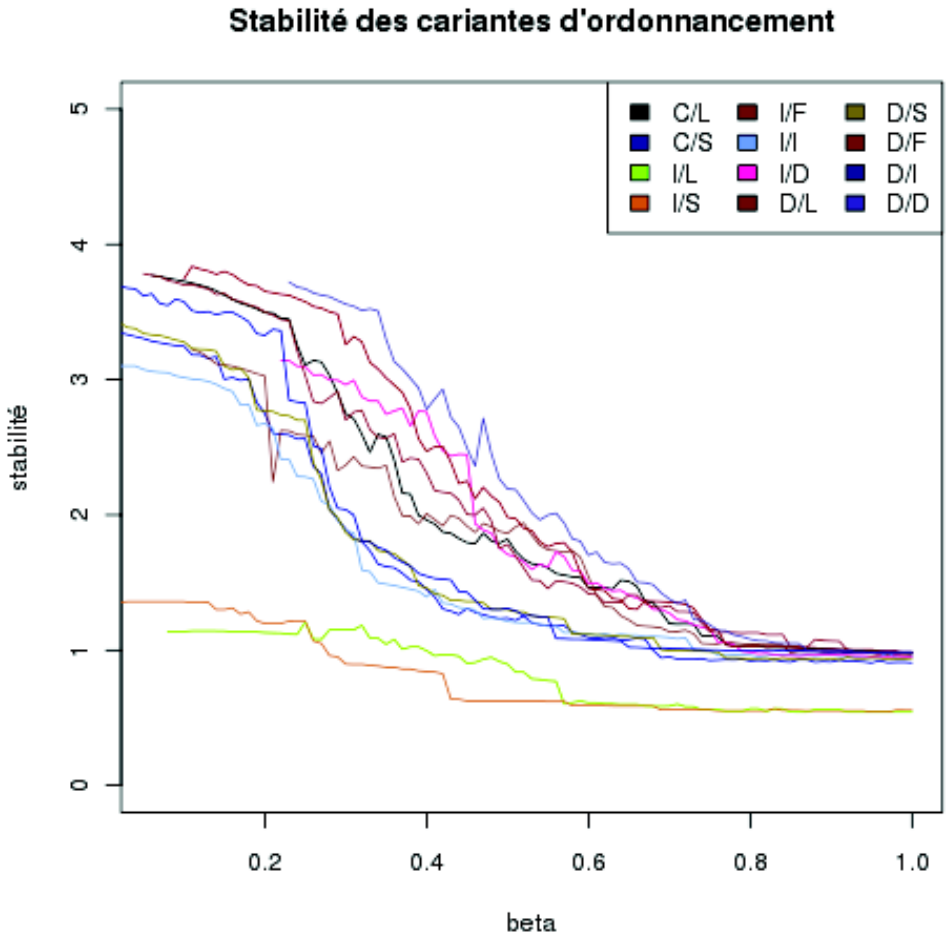


FIGURE 6.8 – Stabilité des ordonnancements en fonction de β

disposons pour cette analyse d'un motif de disponibilité initial et nous nous limitons à 10 instances perturbés. Pour chacune d'elles, nous mesurons la moyenne de durée des disponibilités de chaque intervalle.

L'ensemble des processeurs est ensuite partitionné en deux ensembles : *sensibles* et *insensibles*. Dans ce paragraphe, un processeur est dit insensible s'il a réussit à exécuter toutes les tâches qui lui ont été allouées pour toutes les instances perturbée. Dans le cas contraire, ce processeur est dit sensible.

Pour chaque processeur de ces ensembles, nous calculons la moyenne des durées de disponibilité et nous dessinons les valeurs obtenus sur la figure 6.9.

Cette figure montre que les processeurs sensibles ont une durée moyenne d'intervalle assez faible par rapport au reste des processeurs. Cette durée est de l'ordre de 358.65 minutes (6 heures environ). Dans la pratique, cela veut dire plus un processeur possède de disponibilité, plus, il a de chance de réagir aux perturbations en trouvant de l'espace vide pour l'exécution des tâches et ainsi, d'éviter le cas où une tâche ne pourrait pas être re-exécutée après une perturbation.

Le même raisonnement a été reproduit afin d'évaluer l'impact du nombre de disponibilité d'un processeur sur la stabilité (figure 6.10). Nous avons par consequence re-exécuté les même étapes mais en mesurant le nombre d'intervalles au lieu de la disponibilité. Par rapport à ce critère, nous avons remarqué qu'en moyenne, le nombre d'intervalle par processeur n'a pas d'impact direct sur la sensibilité.

Finalement, pour connaitre le taux des processeurs sensibles, nous avons dessiné l'ecdf des moyennes des durée des processeurs. Nous remarquons sur la figure 6.11 que 34% des processeurs sont potentiellement sensibles et le le reste présente moins de risque de conduite à un ordonnancement perturbé non-réalisable.

6.4.5 Effet de la loi de probabilité

L'incertitude qui porte sur les dates d'occurrences des évènements dans les systèmes distribués sont plutôt liés aux habitudes et aux comportements des fédérateurs de ressources. Bien que la littérature ait abordé des problèmes d'ordonnancement avec incertitude, nous ne connaissons pas de travaux ayant modélisé le comportement aléatoire de l'incertitude.

Nous avons dans ce paragraphe nous comparons le comportement des algorithmes par rapport aux lois de probabilité. Nous avons donc étudié le comportement d'une instance formée par 500 machines et 10000 tâches sur deux types d'instances perturbés par les lois uniformes et exponentielle.

Les perturbations avec la loi uniforme a déjà été décrite dans le paragraphe 6.4.2. Pour le deuxième type de l'instance, l'amplitude des perturbations est générée par une loi exponentielle de taux $\frac{1}{length}$ où $length$ est la moyenne des longueurs des disponibilité du processeur considéré.

Nous avons d'abord mesuré la disponibilité totale et la taille du plus

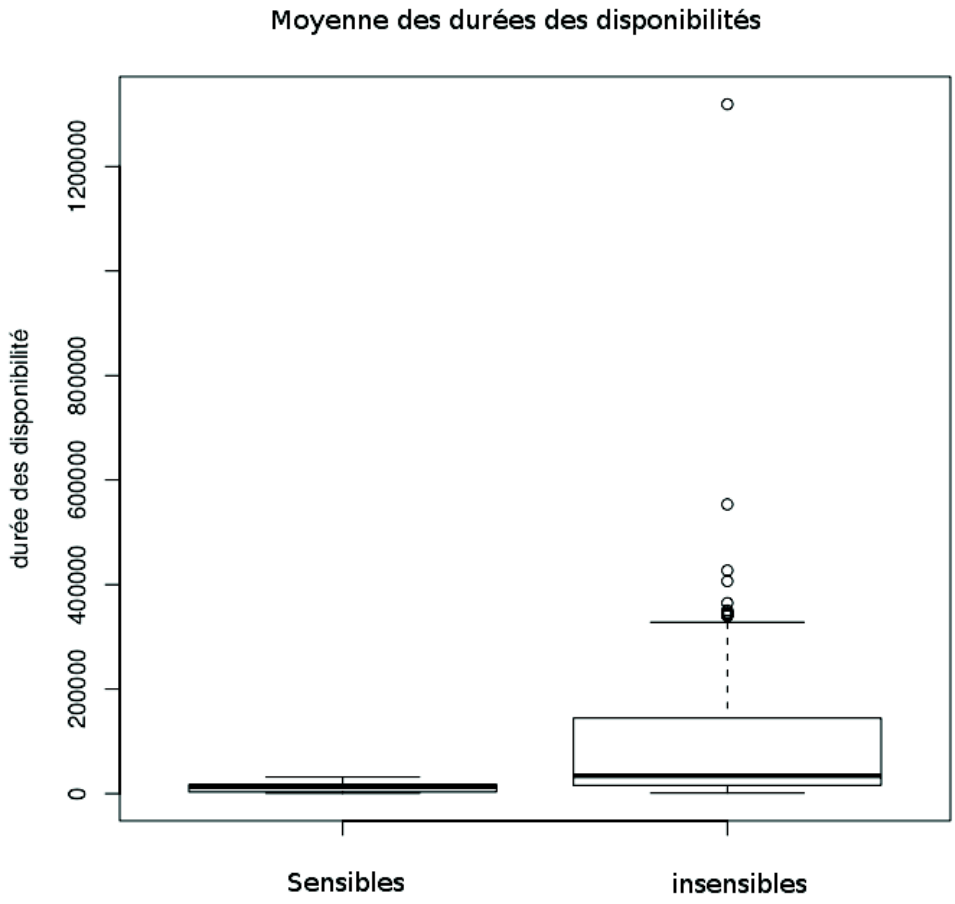


FIGURE 6.9 – Durée des disponibilités des processeurs sensibles et insensibles

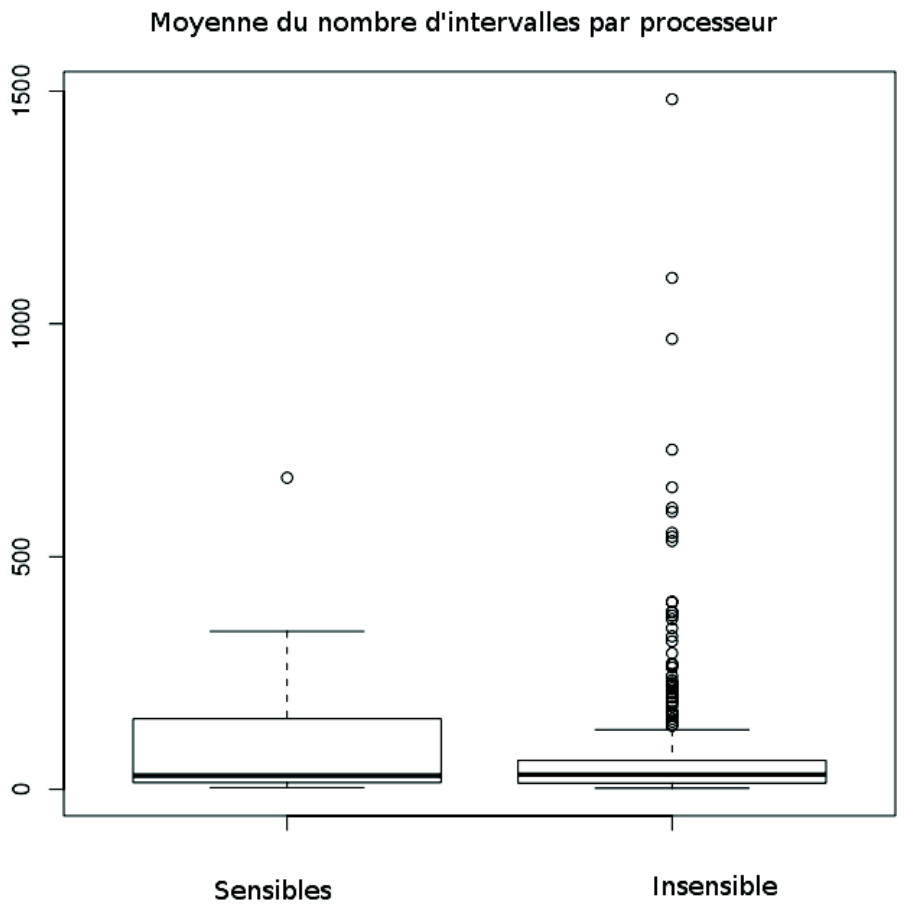


FIGURE 6.10 – Nombre d'intervalles des processeurs sensibles et stables

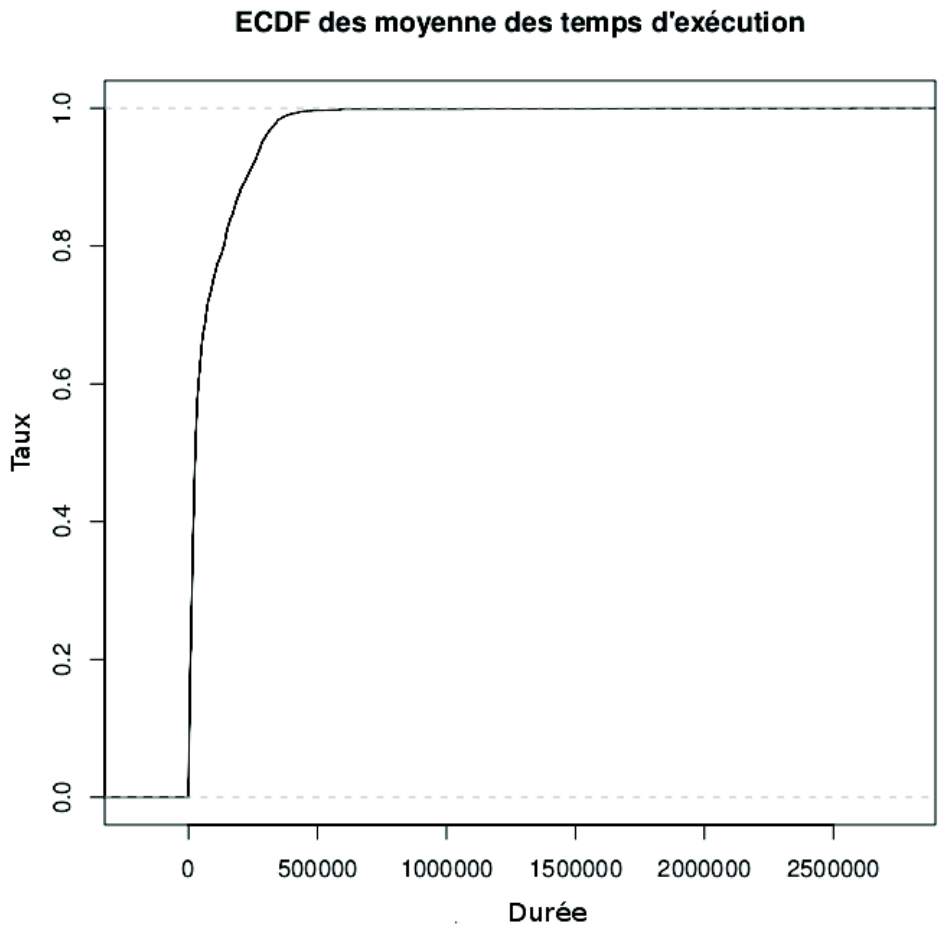


FIGURE 6.11 – ECDF des moyennes des temps d'exécution

grand intervalle respectivement pour l'instance originale et celles perturbées par les lois uniformes et exponentielles. Pour chaque loi, 30 instances ont été générées et c'est la moyenne de la somme totale des disponibilités qui a été calculée. Le résultat est reporté dans le tableau 6.4.5.

Instance	Disponibilité totale	plus longue disponibilité
Instance originale	608560845	13430000
Perturbation avec loi uniforme	620216696	417900
Perturbation avec loi exponentielle	846412427	1516000

Moyenne de disponibilité et taille du plus grand intervalle

Le tableau ci-dessus montre que les instances générées avec la loi exponentielle fournissent en moyenne plus de disponibilité de processeur par rapport aux instances générées avec la loi uniforme.

Nous avons par la suite étudié le comportement des ordonnancements sur des instances perturbées générées avec les lois citées ci-dessus. Pour cela, nous avons générée des ordonnancements avec les différentes heuristiques disponibles puis, nous les avons exécuté sur deux jeux de 30 instances perturbées, la première générée avec la loi uniforme et la deuxième avec la loi exponentielle.

Nous avons, comme dans la section 6.4.2, séparé les aspects qualitatifs et quantitatifs dans l'étude des instances perturbées. Pour chaque heuristique, nous mesurons le nombre d'instances non réalisables en fonction de β pour chaque loi de probabilité (Figure 6.4.5). Nous mesurons aussi la moyenne des horizons des instances réalisables en fonction de β et nous les dessinons dans la figure 6.4.5.

Nous avons remarqué que le nombre d'instances non réalisables avec des perturbations est plus réduit quand la perturbation est générée avec une loi uniforme. Ce nombre converge très vite vers 0 en fonction de β. Seule l'heuristique I/D présente un nombre élevé d'instances non-réalisables quand $\beta = 0$.

L'autre constat est que la moyenne des makespan des ordonnancements perturbés avec la loi exponentielle est toujours inférieure à celles perturbées par la loi uniforme. Ce constat vient confirmer la précédente analyse qui montre que les instances perturbées avec une loi uniforme ont tendance à être instables quand β se rapproche de 0. Cette instabilité est en effet due au fait que l' exécution des tâches est à chaque fois retardée à cause de l'incertitude. Ce décalage est plus important avec la loi uniforme que avec la loi

exponentielle. Comme la disponibilité (par processeur) est finie, nous avons plus de chance d'aboutir à une instance non-réalisable avec la loi uniforme.

6.4.6 Étude expérimentale du regret sur des scénarios clairvoyants

Dans cette série d'expériences, nous effectuons une analyse à propos des scénarios clairvoyants. Nous considérons l'ensemble de 10000 tâches utilisées lors de la série d'expérimentation précédente ainsi que les 10 scénarios perturbés générés. Pour chaque combinaisons de méthode globale et locale utilisée, et pour chacun des 10 scenarios perturbés, nous calculons 6 ordonnancements générés pour les valeurs de β allant de 0 à 1 avec un pas de 0.2 et nous renvoyons l'horizon et la borne inférieur du makespan.

Nous avons aussi calculé toutes les combinaisons d'ordonnancements possibles, pour les méthodes globales C et I, avec la même instance des 10000 tâches et avec les disponibilités initiales (non perturbées). Pour chaque expérience, nous renvoyons l'horizon et la borne inférieure du makespan.

Pour chaque valeur de β et pour chaque algorithme, nous calculons la moyenne des 10 horizons des instances perturbés. Nous calculons par la suite le rapport entre les valeurs moyennes de la borne inférieur de l'horizon. Ce rapport représente la performance des algorithmes si on savait a priori ce que aurait été les perturbations, c.a.d, dans une configuration clairvoyante.

Nous dessinons dans la figure 6.14 la performance de ces ordonnancements en fonction de β. Nous avons remarqué que les résultats sont semblables aux performances avec les instances non perturbées. Ainsi, l'algorithme D/D réalise la meilleure performance parmi tous les algorithmes disponibles. Les algorithmes I/L et I/S ont la pire performance qui s'approche de 3. Cependant la variation, en fonction de β est peu importante par rapport aux autres combinaisons. Le reste des combinaisons a une performance moins bonne mais proche de celles de C/L.

Nous avons aussi calculé les différents ordonnancements de l'instance originale, c.a.d, sans perturbations et nous renvoyons l'horizon et la borne inférieure. Pour chaque algorithme et pour chaque valeur de β nous calculons le rapport entre le makespan de la configuration clairvoyante et le makespan non perturbé. Ce rapport représente le regret dû aux perturbations. Nous dessinons ces rapports dans la courbe 6.15.

Les ordonnancements clairvoyants se comportent mieux que les ordonnancements initiaux. En effet, tous les ratios sont inférieurs et proche de 1 (entre 0.92 et 0.97). Cela s'explique par le fait que, connaître les perturbations nous permet d'optimiser l'affectation des tâches (a postériori). Bien évidemment, cette information ne peux pas être connue à l'avance dans les plateformes que nous étudions.

Par ailleurs, nous avons remarqué que le comportement des algorithmes basés sur Interval Greedy se comportent moins bien que le reste des variantes vis-à-vis de ces derniers critères. Pour des faibles valeurs de β, le rapport entre l'horizon clairvoyant et l'horizon initial est faible. A partir d'une valeur de β supérieure à 0.4, le rapport des algorithmes basés sur Interal Greedy se distingue par une tendance à la hausse (ou en zigzag). En contre parti, le rapport des algorithmes restants diminue en fonction de β.

6.4.7 Complexité temporelle

Les précédentes analyses ont été effectuées sans tenir compte du temps nécessaire pour la génération des ordonnancements. Cependant, nous avons remarqué au fur et à mesure de la conduite des expérimentations, que certaines heuristiques s'exécutent rapidement alors que d'autres prennent un temps relativement long.

Afin d'évaluer la complexité temporelle de ces variantes, nous avons mesuré le temps qu'elle ont mis à générer les ordonnancements. Rappelons que l'exécution s'est effectuée sur les machines de GRID'5000.

Pour chaque valeur de β de 0 à 1 avec pas 0.2 (6 valeurs), nous mesurons le temps que met chaque heuristique à générer la solution. Nous dessinons ensuite dans les figures 6.16 le résultat.

Comme prévu, le premier constat est que l'heuristique D/D est significativement plus couteuse que les autres heuristiques. Cela est du au fait que cette variante utilise un algorithme d'approximation dual combiné avec un algorithme de programmation dynamique. Pour effectuer l'approximation duale, il faut effectuer $log_2(W)$ itération où $W = \sum_{i,j} b_j p_i$ (somme

des durée de toutes les tâches). Chaque itération considère au plus tous les intervalles disponibles (N). L'exécution d'un algorithme de programmation dynamique dure $O(nlog(a_{max}))$. Finalement, au pire des cas, la complexité de cette procédure est $O(Nnlog(a_{max}))$.

Nous pouvons en conséquence considérer, que bien que cette variante donne la meilleure performance et une stabilité raisonnable, elle reste une approche non utilisable dans la pratique vu sa complexité temporelle coûteuse.

Pour une analyse plus fine, nous avons tracé dans la figure 6.17 le temps d'exécution du reste des variantes en éliminant la variante D/D. le reste des heuristiques peuvent être groupées en deux classes : les heuristiques D/S, D/L et I/D forment une classe qui produit un ordonnancement entre 40 minutes et 80 minutes et une seconde classe formé par le reste des algorithmes qui répondent au bout de 10 minutes.

6.5 Conclusion

Nous avons dans ce chapitre effectué un ensemble de simulations qui permettent de prévoir le comportement des différentes heuristiques développée dans ce document.

Dans un premier paragraphe, nous avons fait la simulation de l'algorithme GAPS developpé dans le chapitre 4. Cet algorithme réalise un compromis entre la performance et la stabilité. Il est paramétré par une un facteur β et l'objectif de la campagne fut la détermination de la valeur de β qui offre le meilleur compromis entre ces deux objectifs. Nous avons conclu que le meilleur compromis est obtenu pour une valeur de $\beta = 0.6$.

Nous avons par la suite simulé les algorithmes qui ont été conçus dans le chapitre 5. Cette variété d'algorithmes est de même paramétrée par β. L'objectif de l'expériementation est de trouver quelles sont les variantes de l'algorithmes qui offrent un compromis raisonnable vis-à-vis de la stabilité et de la performance. Par ailleurs, lors de la phase de simulation, il s'est avéré que certains algorithmes de cet ensemble mettent un temps non raisonnable pour calculer l'ordonnancement.

Après la prise en considérations de tout ces paramètres, nous avons conclu que les heuristiques C/S, D/F et D/I, utilisés avec des valeurs de β proches de 0.6 sont un compromis acceptable vis-à-vis de tous les critères considérés.

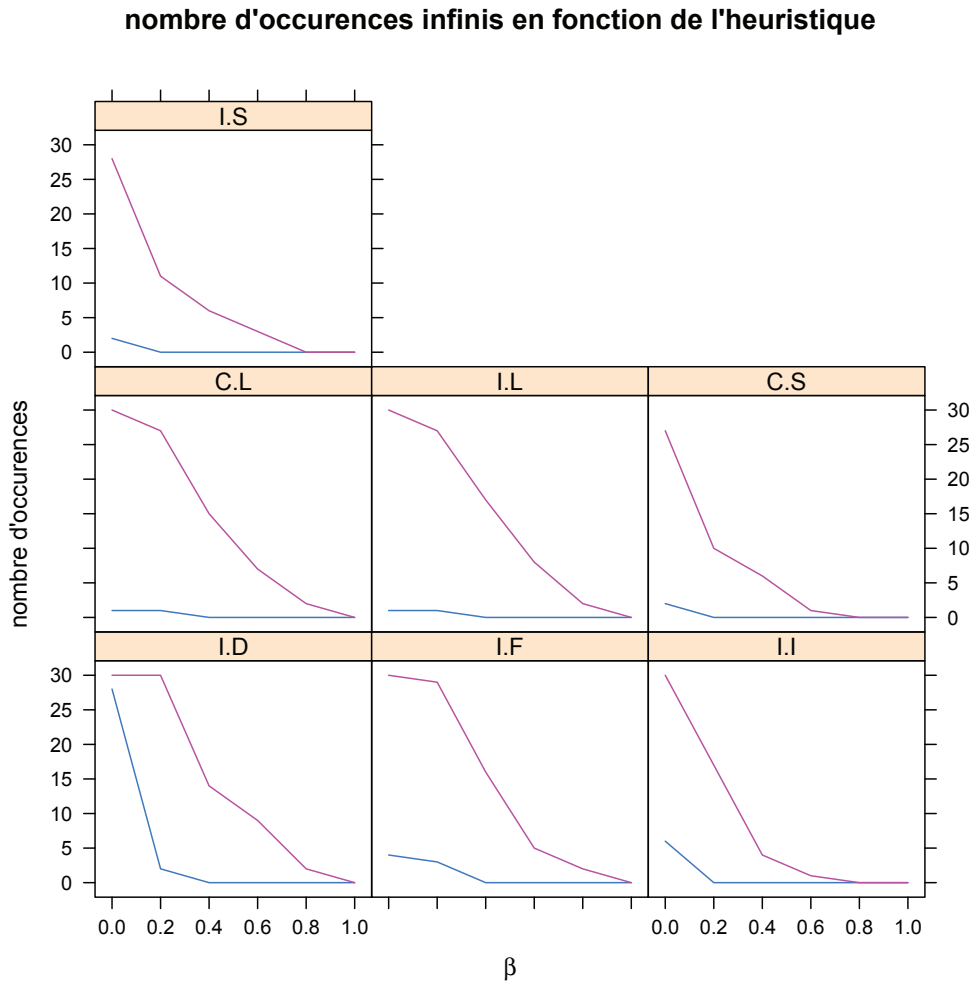


FIGURE 6.12 – Nombre d'instances non-réalisable

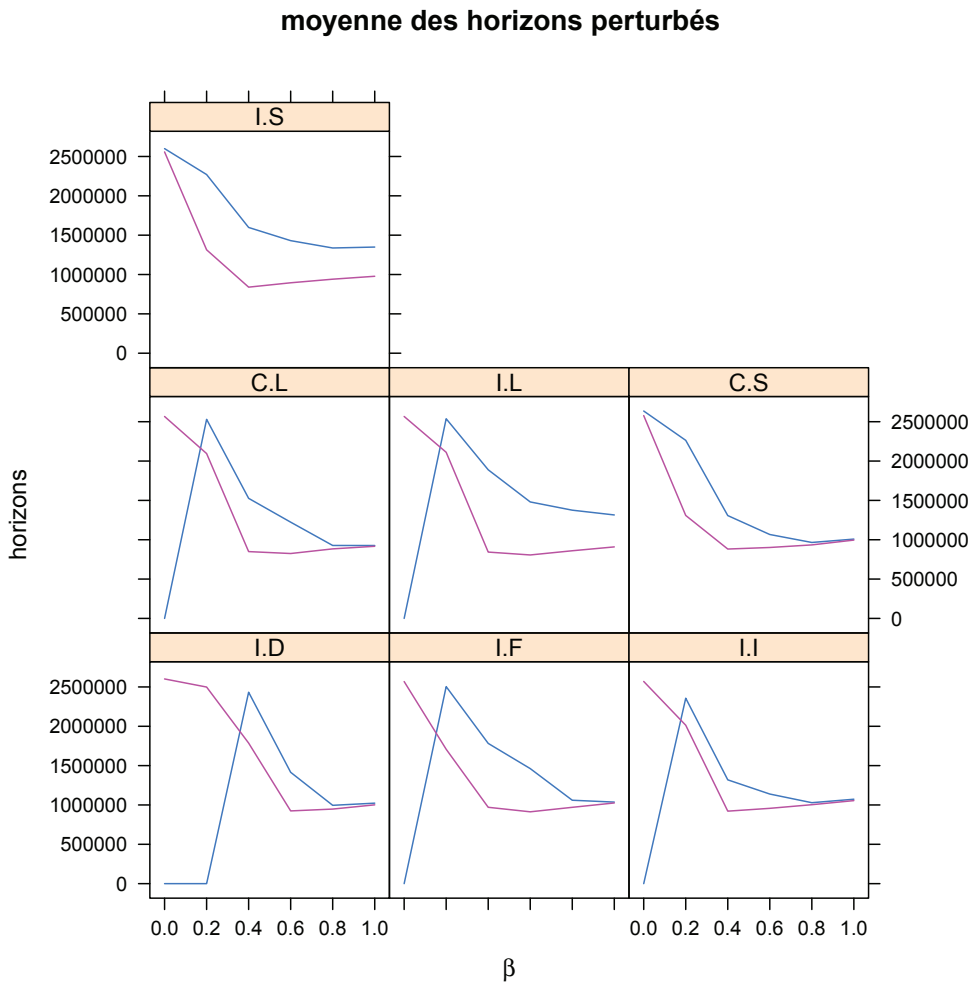


FIGURE 6.13 – Moyenne des makespan non perturbé

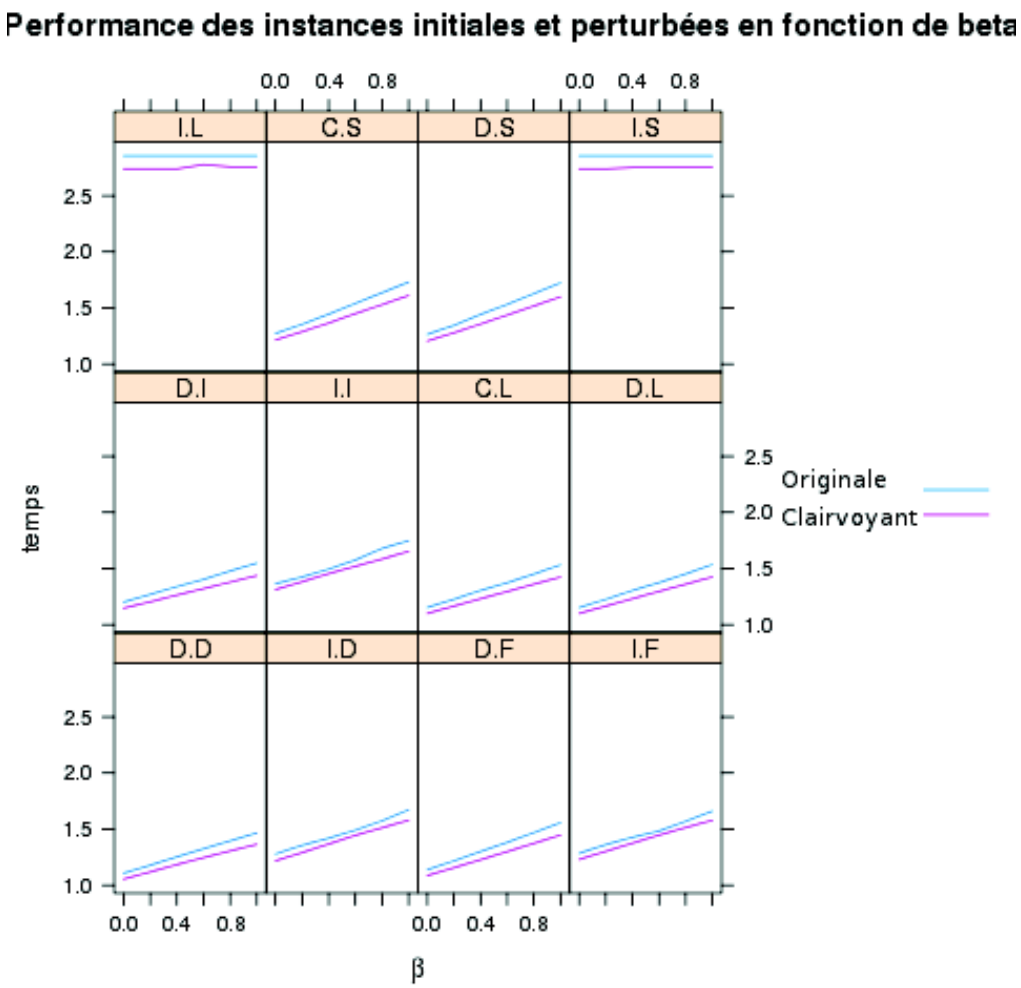


FIGURE 6.14 – Performances des algorithmes clairvoyants

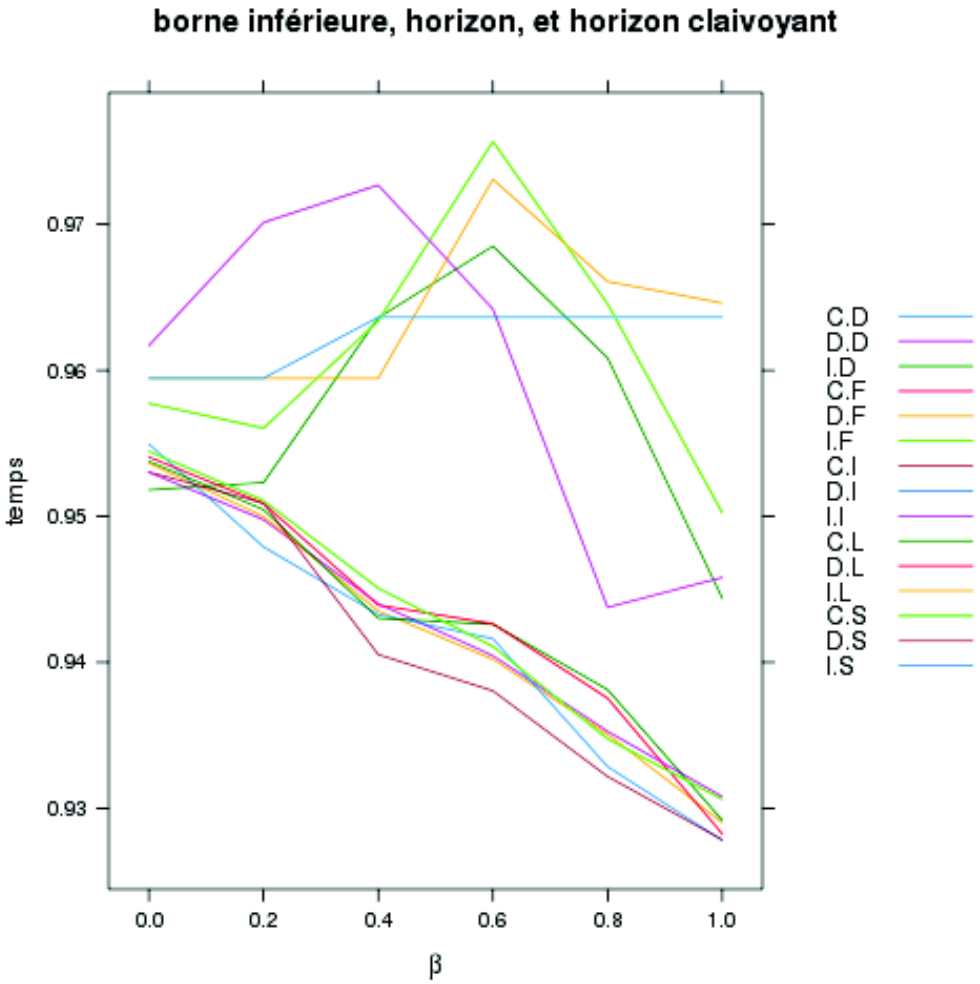


FIGURE 6.15 – Rapport entre l'horizon clairvoyant et l'horizon initial

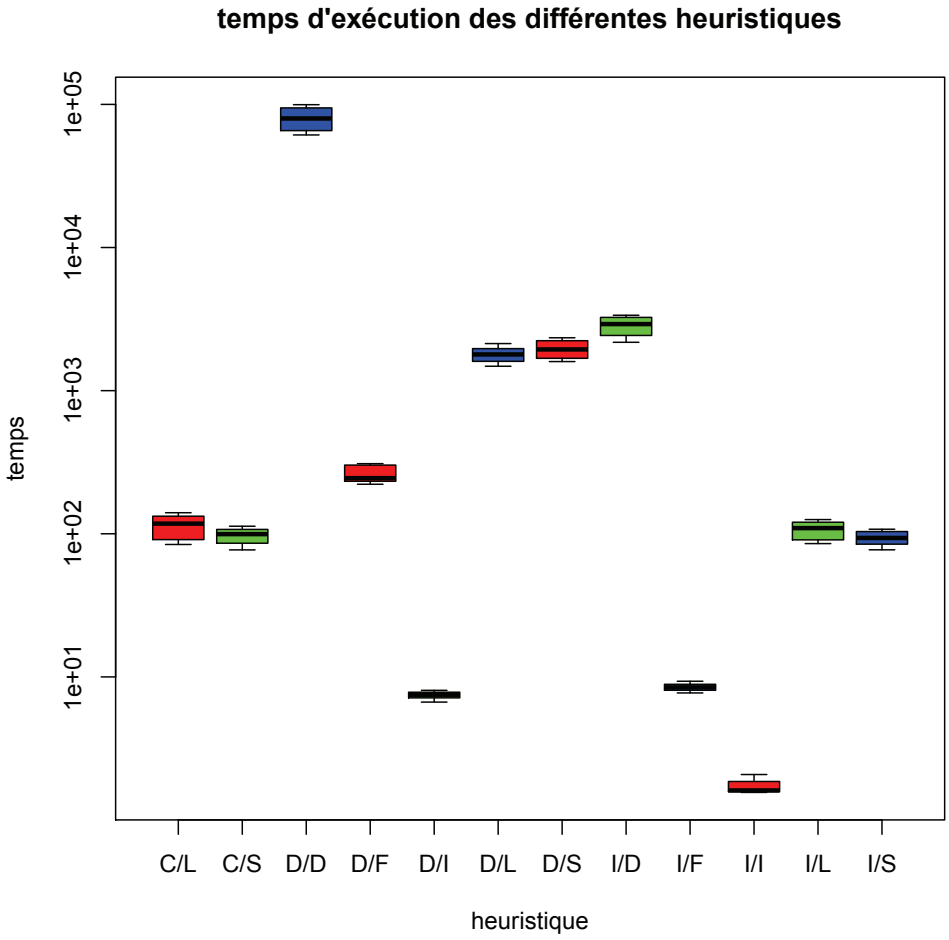


FIGURE 6.16 – Durée d'exécution des variantes conçues

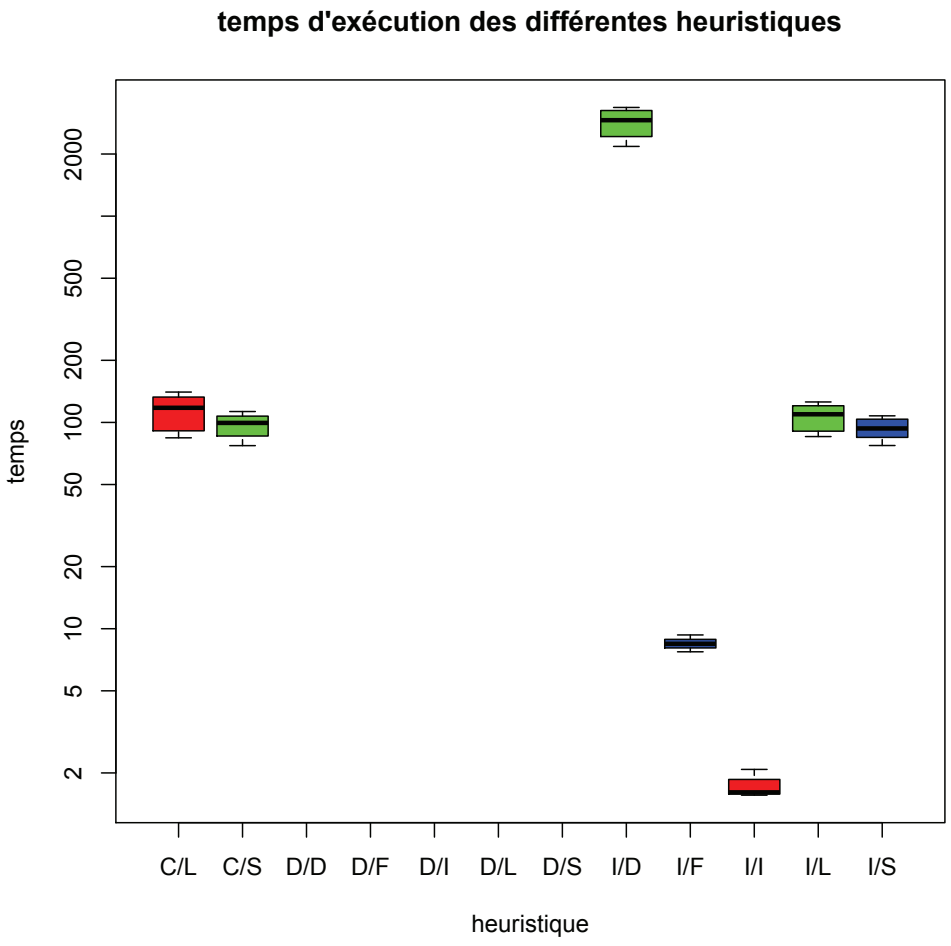


FIGURE 6.17 – Durée d'exécution des variantes conçues (sans D/D)

Chapitre 7

Conclusion

Nous assistons aujourd'hui à un paysage très vaste et en plein développement des plateformes dédiées au calcul distribué. Ces plateformes ont pour but d'offrir une capacité gigantesque de ressources (calcul, stockage, services ...) qu'on n'aurait jamais pu avoir avec une seule machine et à un coût raisonnable. Les plateformes de calcul global, notamment les grilles de calcul ont été l'une des solutions les plus adoptées pour construire de telle plateformes. Ce succès est dû au rapport intéressant entre la qualité de service offerte et leur coût relativement faible. Cependant, l'adoption de ces plateformes a engendré l'émergence de nouveaux problèmes auxquelles il faut trouver une solution pour optimiser leur utilisation.

Dans le cadre de cette thèse, nous avons abordé l'un des problèmes les plus critiques dans le domaine du calcul distribué, à savoir, l'ordonnancement. En particulier comme ces plateformes sont caractérisées par être fortement perturbées, il est essentiel de construire des ordonnancements qui doivent d'une part optimiser les fonctions objectifs classiques et d'autre part résister aux perturbations dues au caractère volatil des ressources. Les solutions que nous avons proposées ont adopté un modèle proactif qui consiste à construire des solutions initiales et qu'on fait légèrement modifier au fur et à mesure de l'exécution. La résistance aux aléa est effectuée grâce à un mécanisme de tampon qui est un espace libre qu'on laisse systématiquement devant les indisponibilités.

Notre première contribution, présentée dans le chapitre 3, est d'effectuer une étude préliminaire sur l'ordonnancement avec une seule contrainte d'indisponibilité machine. Nous avons analysé la performance d'algorithme basés sur LPT. Nous avons par la suite, conçu une famille d'algorithme flexibles

qui offrent un compromis entre la stabilité et la performance.

Dans les deux chapitres qui suivent, nous avons généralisé cette étude pour les cas de plusieurs indisponibilités par machines et où les indisponibilités peuvent aussi bien se produire en avance (chapitre 4) qu'en avance et en retard (chapitre 5). Nous proposons pour ces problèmes, douze algorithmes d'ordonnancement basés sur la technique de tampon qui offrent un compromis entre la stabilité et la performance.

Étant conscient de l'importance de produire une solution pratique qui peut être utilisée dans des ordonnanceurs réels, nous avons simulé les différentes approches conçues dans le dernier chapitre. Cette simulation a été effectuée en se basant sur des traces fournies par le projet BOINC. A la fin de cette campagne, nous avons pu identifier deux algorithmes qui réalisent de bon compromis.

Nous restons persuadés que ce travail ouvre des perspectives intéressantes à explorer. En effet, dans les modèles que nous avons adopté les tâches ne sont pas autorisées à migrer (la réaction à une éventuelle perturbation se fait en réexéctant la tâche sur le même processeur). Il est donc possible d'envisager une plus grande flexibilité dans les mécanismes de réaction aux perturbations. Par ailleurs, lors des études effectuées, nous avons pu identifier que certains processeurs sont plus sensibles que d'autres. Il est par conséquence possible d'envisager l'utilisation de la technique de la duplication des tâches au lieu du mécanisme du tampon pour anticiper les perturbations.

Bibliographie

[1] The project distributed.net : the fastest computed on the earth.

[2] The project united devices : www.ud.com.

[3] N. Abdennadher and R. Boesch. Xtremweb-ch : Une plateforme global computing pour les applications de haute performance. Technical report, Ecole d'Ingénieurs de Genève, August 2004.

[4] D. Abramson, D. Giddy, and W. Kotler. High performance parametric modeling with nimrod/g : Killer application for the global grid ? In *IPDPS '00 : Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, Washington, DC, USA, 2000. IEEE Computer Society, IEEE Computer Society.

[5] I. Adiri, J. Bruno, E. Frostig, and R. A. H. G. Kan. Single machine flow-time scheduling with a single breakdown. *Acta Inf.*, 26 :679–696, September 1989.

[6] S. Albers and G. Schmidt. Scheduling with unexpected machine breakdowns. *Discrete Applied Mathematics*, 110(2–3) :85 – 99, 2001.

[7] M. Aloulou, P. M.C., and A. Vignier. Predictive-reactive scheduling for the single machine problem. In *the 8th Workshop on Project Management and Scheduling*, pages 3–5, Valencia, Spain, 2002.

[8] D. P. Anderson. Boinc : A system for public-resource computing and storage. In *5th International Workshop on Grid Computing (GRID)*, pages 4–10, Pittsburgh, USA, Nov.

[9] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home : An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11) :56–61, 2002.

[10] A. Arjav, B. Gerald, and L. Mario. The organic grid : Self-organizing computation on a peer-to-peer network. In *ICAC*, pages 96–103, New York, NY, USA, 2004.

[11] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *IPTPS*, pages 256–267, 2003.

[12] J. Billaut, A. Moukrim, and E. Sanlaville, editors. *Flexibilité et robustesse en ordonnancement (in French, English version to appear)*. Lavoisier, 2005.

[13] J. Billaut, A. Moukrim, and E. Sanlaville, editors. *Flexibility and Robustness in Scheduling*. Wiley-ISTE, 2008.

[14] J. Bżewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Scheduling computer and manufacturing processes*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[15] G. Boss, P. Malladi, S. Quan, L. Legregni, and H. Hall. Cloud computing. Technical report, IBM high performance on demand solutions, 2007.

[16] P. Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3rd edition, 2001.

[17] L.-C. Canon, A. Essafi, G. Mounié, and D. Trystram. A Bi-Objective Scheduling Algorithm for Desktop Grids with Uncertain Resource Availabilities. In *Euro-Par*, Bordeaux, France, Sept. 2011.

[18] P. Chretienne, E. G. Coftman, J. K. Lenstra, and Z. Liu, editors. *Scheduling Theory and Its Application*. John Wiley and sons, 1995.

[19] W. Cirne, F. Brasileiro, J. Sauvé, N. Andrade, D. Paranhos, E. Santos-neto, R. Medeiros, and F. Campina Gr. Grid computing for bag of tasks applications. In *Proceeding of the 3rd IFIP Conference on E-Commerce, E-Business and EGovernment*, Sao Paulo, Brazil, 2003.

[20] W. Cirne, F. V. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the world, unite!!! *J. Grid Comput.*, 4(3) :225–246, 2006.

[21] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles*. InterEditions, collection IIA, 1993.

[22] F. A. B. da Silva, S. Carvalho, H. Senger, E. R. Hruschka, and C. R. G. de Farias. Running data mining applications on the grid : A bag-of-tasks approach. In *ICCSA (2)*, pages 168–177, 2004.

[23] A. Davenport and J. Beck. A survey of techniques for scheduling with uncertainty. Available from

http ://www.eil.utoronto.ca/profiles/chris/gz/uncertainty-survey.ps, 2003.

[24] F. Deblaere, E. Demeulemeester, and W. Herroelen. Proactive policies for the stochastic resource-constrained project scheduling problem. *European Journal of Operational Research*, 214(2) :308 – 316, 2011.

[25] F. Diedrich, K. Jansen, F. Pascual, and D. Trystram. Approximation Algorithms for Scheduling with Reservations. *Algorithmica*, 58(2) :391–404, 2010.

[26] F. Diedrich and U. M. Schwarz. A framework for scheduling with online availability. In *European Conference on Parallel Processing*, pages 205–213, 2007.

[27] P. A. Dinda. A prediction-based real-time scheduling advisor. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 35–, Washington, DC, USA, 2002. IEEE Computer Society.

[28] G. F. Dror and R. Larry, editors. *Job Scheduling Strategies for Parallel Processing, IPDPS 2000 Workshop, JSSPP 2000, Cancun, Mexico, May 1, 2000, Proceedings*, volume 1911 of *Lecture Notes in Computer Science*. Springer, 2000.

[29] P.-F. Dutot, G. Mounié, and D. Trystram. Scheduling Parallel Tasks : Approximation Algorithms. In J. T. Leung, editor, *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*, chapter 26, pages 26–1 – 26–24. CRC Press, 2004.

[30] T. Estrada, D. A. Flores, M. Taufer, P. J. Teller, A. Kerstens, and D. P. Anderson. The effectiveness of threshold-based scheduling policies in boinc projects. In *Proceedings of the 2nd IEEE International Conference in e-Science and Grid Computing*, Amsterdam, Netherlands, 2006.

[31] L. Eyraud, G. Mounié, and D. Trystram. Analysis of Scheduling Algorithms with Reservations. In *21st IEEE International Parallel & Distributed Processing Symposium*, 2007.

[32] M. Fallah, M. Aryanezhad, and B. Ashtiani. Preemptive resource constrained project scheduling problem with uncertain resource availabilities : Investigate worth of proactive strategies. In *2010 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 646 –650, Dec. 2010.

[33] I. Foster and C. Kesselman. *The Grid : Blueprint for a New Computing Infrastructure*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Elsevier, 2004.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[35] C. Germain, V. Néri, G. Fedak, and F. Cappello. Xtremweb : Building an experimental platform for global computing. In *Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, GRID '00, pages 91–101, London, UK, 2000. Springer-Verlag.

[36] M. Girault. *Initiation aux processus aléatoires :*. Probabilités, statistique, recherche opérationnelle. Dunod, 1959.

[37] F. Glover and M. Laguna. *Tabu search*, pages 70–150. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[38] R. L. Graham, E. L. Lawer, J. K. Lenstra, and A. H. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling : A survey. *Ann. Disc. Math*, 5 :287–326, 1979.

[39] N. G. Hall and M. E. Posner. Sensitivity analysis for scheduling problems. *J. of Scheduling*, 7(1) :49–83, jan 2004.

[40] H. Heithem Abbes, C. Cérin, and M. Jemni. A decentralized and fault-tolerant desktop grid system for distributed applications. *Concurrency and Computation : Practice and Experience*, 22(3) :261–277, 2010.

[41] W. Hopp and M. Spearman. *Factory physics : foundations of manufacturing management*. McGraw-Hill, 2001.

[42] L. Hui, H. Yu, and L. Xiaoming. A lightweight execution framework for massive independent tasks. In *2008 Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–9. IEEE, Nov. 2008.

[43] H.-C. Hwang and S. Y. Chang. Parallel Machines Scheduling with Machine Shutdowns. *Computers & Mathematics with Applications*, 36(11) :21–31, Aug. 1998.

[44] H.-C. Hwang, K. Lee, and S. Y. Chang. The effect of machine availability on the worst-case performance of LPT. *Discrete Applied Mathematics*, 148(1) :49–61, Apr. 2005.

[45] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22 :463–468, October 1975.

[46] E. Jeanvoine, C. Morin, and D. Leprince. Vigne : Executing easily and efficiently a wide range of distributed applications in grids. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 394–403. Springer, 2007.

[47] E. Jeanvoine, L. Rilling, C. Morin, and D. Leprince. Using Overlay Networks to Build Operating System Services for Large Scale Grids. Research Report PI 1773, Université de Rennes, 2005.

[48] D. Johnson. *Near-optimal Bin Packing Algorithms*. Massachusetts Institute of Technology, project MAC. Massachusetts Institute of Technology, 1973.

[49] I. Kacem. Approximation algorithms for the makespan minimization with positive tails on a single machine with a fixed non-availability interval. *J. Comb. Optim.*, 17(2) :117–133, 2009.

[50] N. Kasap, H. Aytug, and A. Paul. Minimizing makespan on a single machine subject to random breakdowns. *Operations Research Letters*, 34 :29–36, 2006.

[51] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[52] D. Kondo, A. Andrzejak, and D. P. Anderson. On correlated availability in internet-distributed systems. In *Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing*, pages 276–283, Tsukuba, Japan, Sept. 2008.

[53] D. Kondo, G. Fedak, F. Cappello, A. A. Chien, and A. Casanova. Characterizing resource availability in enterprise desktop grids. *Future Gener. Comput. Syst.*, 23(7) :888–903, 2007.

[54] D. Kondo, M. Taufer, C. L. B. III, H. Casanova, I. Henri, C. Andrew, and A. A. Chien. Characterizing and evaluating desktop grids : An empirical study. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'04*, 2004.

[55] O. Lambrechts, E. Demeulemeester, and W. Herroelen. Proactive and reactive strategies for resource-constrained project scheduling with uncertain resource availabilities. *Journal of Scheduling*, 11(2) :121–136, 2008.

[56] O. Lambrechts, E. Demeulemeester, and W. Herroelen. Time slack-based techniques for robust project scheduling subject to resource uncertainty. *Annals of Operations Research*, pages 1–22, 2010.

[57] E. L. Lawler. Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 4 :339–356, 1979.

[58] C.-Y. Lee. Parallel machines scheduling with nonsimultaneous machine available time. *Discrete Applied Mathematics*, 30(1) :53–61, Jan. 1991.

[59] C.-Y. Lee. Machine scheduling with an availability constraint. *Journal of Global Optimization*, 9(3) :363–382, 1996.

[60] C.-Y. L. Lee and G. Yu. Parallel-machine scheduling under potential disruption. *Optimization Letters*, 2 :27–37, 2008.

[61] A. Legrand and Y. Robert. *Algorithmique parallèle : cours et exercices corrigés*. Sciences Sup. Dunod, 2003.

[62] W. Li and C. J. Stochastic scheduling on a single machine subject to multiple breakdowns according to different probabilities. *Operations Research Letters*, 18 :81–91, 1995.

[63] C. J. Liao, D. L. Shyur, and C. H. Lin. Makespan minimization for two parallel machines with an availability constraint. *European Journal of Operational Research*, 160(2) :445–456, June 2005.

[64] Z. Liu, W. Dou, W. M. Zhang, and P. Zou. Paradropper : A general-purpose global computing environment built on peer-to-peer overlay network. In *ICDCS Workshops*, pages 954–957, 2003.

[65] A. Mahjoub. *Etude de la Robustesse des Algorithmes d'Ordonnancement et de Localisation*. PhD thesis, INP Grenoble, 2004.

[66] A. Mahjoub, J. Pecero Sánchez, and D. Trystram. Scheduling with uncertainties on new computing platforms. *Computational Optimization and Applications*, 48(2) :369–398, 2011.

[67] R. Mellouli, C. Sadfi, C. Chu, and I. Kacem. Identical parallel-machine scheduling under availability constraints to minimize the sum of completion times. *European Journal of Operational Research*, 197(3) :1150–1165, 2009.

[68] A. Montresor and H. Meling. Messor : Load-balancing through a swarm of autonomous agents. In *Proceedings of 1st Workshop on Agent and Peer-to-Peer Systems*, pages 125–137, 2002.

[69] D. Nurmi, J. Brevik, and R. Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. In *Euro-Par'05*, pages 432–441, Lisboa, Portugal, 2003.

[70] M. L. Pinedo. *Scheduling : Theory, Algorithms, and Systems*. Prentice Hall, 2001.

[71] E. Sanlaville and S. Günter. Machine scheduling with availability constraints. *Acta Informatica*, 35(9) :795–811, 1998.

[72] S. Saroiu, P. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Storage and Retrieval for Image and Video Databases*, 2002.

[73] U. Schwarz. Online scheduling on semi-related machines. *Information Processing Letters*, 108 :38–40, 2008.

[74] U. Schwarz and F. Diedrich. Scheduling algorithms for random machine profiles. Unpublished manuscript.

[75] S. Smallen, W. Cirne, J. Frey, F. Berman, R. Wolski, M. Su, C. Kesselman, S. Young, and M. Ellisman. Combining workstations and supercomputers to support grid applications : The parallel tomography experience. Technical report, La Jolla, CA, USA, 2000.

[76] W. Smith. *Various optimizers for single-stage production*. Research report. University of California, 1955.

[77] M. Taufer, M. Crowley, D. J. Price, A. A. Chien, and C. L. B. III. Study of a highly accurate and fast protein-ligand docking method based on molecular dynamics. *Concurrency and Computation : Practice and Experience*, 17(14) :1627–1641, 2005.

[78] S. Varrette. *Sécurité des Architectures de Calcul Distribué : Authentification et Certification de Résultats*. PhD thesis, INP Grenoble and Université du Luxembourg, Sept. 2007. In French.

[79] S. Varrette, S. Georget, J. Montagnat, J. Roch, and F. Leprevost. Distributed authentication in grid5000. In *OTM Workshops*, pages 314–326, Agia Napa, Cyprus, 2005.

[80] G. E. Vieira, J. W. Herrmann, and E. Lin. Rescheduling manufacturing systems : A framework of strategies, policies, and methods. *Journal of Scheduling*, 6 :39–62, 2003.

[81] T. Vincent and J. Billaut. *Multicriteria Scheduling - Theory, Models and Algorithms (2. ed.)*. Springer, 2006.