Minimisation du nombre de tâches d'un système temps réel par regroupement

Antoine Bertout

▶ **To cite this version:**

Antoine Bertout. Minimisation du nombre de tâches d'un système temps réel par regroupement. Informatique [cs]. Université de Lille1, 2015. Français. NNT : . tel-01271368

HAL Id: tel-01271368
https://hal.science/tel-01271368

Submitted on 9 Feb 2016

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.

Public Domain

Minimisation du nombre de tâches d'un système temps réel par regroupement

Antoine BERTOUT

Thèse en vue de l'obtention du titre de
Docteur en informatique
DE L'UNIVERSITÉ DE LILLE
École Doctorale des Sciences Pour l'Ingénieur de Lille - Nord De France

présentée et soutenue le 25/11/2015

JURY

Liliana CUCU-GROSJEAN	Chargée de Recherche, HDR, INRIA Paris-Rocquencourt, France	Rapporteur
Marco DI NATALE	Professeur des Universités, Scuola Superiore Sant'Anna di Pisa, Italie	Examinateur
Julien FORGET	Maître de Conférence, Université de Lille, France	Co-Encadrant
Giuseppe LIPARI	Professeur des Universités, Université de Lille, France	Examinateur
Richard OLEJNIK	Ingénieur de Recherche, HDR, CNRS, CRIStAL	Directeur
Pascal RICHARD	Professeur des Universités, Université de Poitiers, France	Rapporteur

Université de Lille

MEL MÉTROPOLE EUROPÉENNE DE LILLE

CRIStAL Centre de Recherche en Informatique, Signal et Automatique de Lille

Remerciements

J'ai toujours pris un plaisir curieux ou un curieux plaisir à lire les remerciements des manuscrits de thèse. Cette fois, c'est à mon tour de me livrer.

J'adresse mes premiers remerciements à mes encadrants. J'exprime toute ma gratitude à Julien Forget. Très présent, il a été central dans l'aboutissement de mon doctorat. J'ai apprécié sa rigueur scientifique et ses qualités humaines. C'est désormais à moi de voler de mon propre zèle. Merci à Richard Olejnik pour son suivi et sa gentillesse.

Je remercie également les membres de mon jury. Tout d'abord Liliana Cucu-Grosjean et Pascal Richard, qui ont accepté de consacrer de leur temps précieux à la relecture de mon travail. Ensuite, je remercie Marco Di Natale et Giuseppe Lipari pour leur participation à mon jury en tant qu'examinateurs.

Je garderai un excellent souvenir de mon passage au sein de l'équipe Emeraude. Je me rappellerai mes sympathiques collègues, Abdoulaye, Clément, Giuseppe, Mahyar, Philippe, Pierre et Laure qui à l'époque, m'avait vivement incité à m'engager dans cet ouvrage. J'ai une pensée également pour Frédéric qui m'a accordé sa confiance en ce qui concerne l'enseignement. Je remercie Houssam pour son entièreté et nos discussions fertiles, quel qu'en soit le sujet. Un grand merci à mes camarades de l'IRCICA, Michel, Anne-So, Ahmed et Xavier. Je n'oublierai pas nos bons moments et nos franches tranches de rigolade.

Du côté de l'association des doctorants Tilda, j'accorde une mention spéciale à Fabien, Guillaume, Alexandre et Géraud. C'était une parenthèse agréable et enrichissante de faire partie du bureau avec vous. Merci à Julie de m'avoir donné l'envie de participer à l'aventure.

Merci au cercle d'escrime de Douai, ma deuxième famille depuis 18 ans, constitutive, avec tous ceux qui vont suivre, de ce que j'ai pu devenir aujourd'hui. Alors merci à mes francs amis tireurs, en particulier à mon maître d'armes Fred et à David.

Je ne peux pas omettre de citer mes deux colocs adorés, mes deux amis, Yves et Gauthier. Ils m'ont supporté, écouté et ont enduré les soirées où la thèse ne s'arrêtait pas à la porte du 31. Je remercie ma belle bande d'amis proches, appuis inconditionnels, Ismaël, Hélo, Bree, Cécile, Justine, Barbie et Mariche. Je n'oublie pas mes vieux comparses Bruno et Doumé ainsi que mon noyau dur, Antoine, Ben, Jeannot et Kuku.

J'ai une pensée toute particulière pour ma petite famille. À papy et mamie Auby, pour leur perpétuelle leçon de courage et leur tendresse. À ma grande sœur Caro et à mon joyeux beau-frère Michou, tous deux toujours aux petits soins à mon égard. Merci à mes parents, pour leur constant soutien et leur confiance sans faille. À Sandra, pour tout ce qu'elle m'apporte au quotidien.

Titre de la thèse / Title

Minimisation du nombre de tâches d'un système temps réel par regroupement

Minimizing the number of tasks of a real-time system by clustering

Résumé / Abstract

Les systèmes embarqués des domaines de l'aéronautique ou de l'automobile sont en interaction permanente avec leur environnement. Ils récupèrent de l'information depuis leurs capteurs, traitent les données et réagissent par le biais de leurs actionneurs. Ces systèmes critiques se doivent non seulement de produire des résultats corrects du point de vue logique mais aussi de les réaliser dans le temps imparti. Cette particularité les classe dans la famille des systèmes temps réel. Dans les domaines cités, les fonctionnalités sont à l'origine définies au regard de la dynamique du système et leur nombre peut atteindre plusieurs milliers. Les systèmes d'exploitation temps réel, logiciels responsables du traitement de ces fonctionnalités sur le matériel, limitent généralement le nombre de traitements implantables, en raison des surcoûts engendrés par leur gestion. Dans ce travail, nous nous intéressons donc à des techniques de réduction du nombre de ces traitements, de manière à passer outre les limitations des systèmes d'exploitation temps réel. Nous proposons des algorithmes de regroupement qui assurent que les contraintes de temps soient respectées. Ces méthodes visent des architectures monoprocesseurs et multiprocesseurs pour des traitements communicants.

Embedded systems dedicated to aeronautics or automotive interact permanently with their environment. They get information from their sensors, process the data and react with their actuators. Such systems have to execute the functionalities correctly, but also to process them within the allocated time. This feature classifies those systems in the category of real-time systems. In the cited domains, those functionalities are originally defined accordingly to the dynamics of the system and their number can reach several thousand. The real-time operating systems, software which handles the processing of those functionalities on the hardware, generally limit the number of functionalities, due to the overhead caused by their management. In this work, we are interested in techniques that reduce the number of those functionalities so to overstep those restrictions. We propose clustering algorithms that ensure that timing constraints are respected. These methods are applied to monoprocessor and multiprocessor architecture with communicating processes.

Mots clés / Key Words

systèmes embarqués - système temps réel - conception de systèmes embarqués - ordonnancement - regroupement de tâches - association de tâches - runnable-to-task mapping - optimisation combinatoire

embedded systems - real-time systems - design of embedded systems - scheduling - task clustering - task mapping - runnable-to-task mapping - combinatorial optimization

Table des matières

Remerciements iii

Table des matières v

Table des figures vii

Table des notations ix

Introduction 1

Motivations et état de l'art 3

1 Modélisation de systèmes temps réel 5
1.1 Définitions . . . 5
1.2 Notions d'ordonnancement temps réel . . . 7

2 Programmation de systèmes temps réel critiques 11
2.1 Processus, threads et tâches . . . 11
2.2 Changements de contexte et préemptions . . . 13
2.3 Surcoûts d'un grand nombre de tâches . . . 14
2.4 Regroupement de tâches . . . 15

3 Travaux connexes 17
3.1 Systèmes distribués . . . 17
3.2 Limitation des préemptions . . . 17
3.3 Méthodologies de conception de systèmes embarqués . . . 18
3.4 Langages synchrones . . . 21

4 Prototype 23
4.1 Vue d'ensemble . . . 23
4.2 Génération de tâches . . . 23
4.3 Tests d'ordonnançabilité . . . 24
4.4 Fonctions d'import et d'export . . . 25
4.5 Annonce du plan . . . 27

Regroupement de tâches indépendantes en contexte monoprocesseur 29

5 Modèle et ordonnançabilité d'un regroupement 31

5.1 Modèle de tâche temps réel 31
5.2 Modèle de regroupement 31
5.3 Ordonnancement monoprocesseur 34
5.4 Impact du regroupement 38
5.5 Regroupement à coût nul 40
5.6 Coût du regroupement 41

6 Minimisation du nombre de tâches 43
6.1 Complexité du problème de regroupement de tâches 43
6.2 Non-ordonnançabilité viable 44
6.3 Heuristique 46
6.4 Formulation en un problème d'optimisation linéaire en nombres entiers 52

7 Expérimentations 57
7.1 Génération d'ensembles de tâches 57
7.2 Évolution du nombre de tâches 58
7.3 Comparaisons des fonctions de coût 58
7.4 Variation des bornes de génération des échéances 62
7.5 Évolution des changements de contexte et des préemptions 62

Regroupement de tâches avec contraintes de précédence en contexte monoprocesseur 65

8 Modèle et ordonnançabilité du regroupement de tâches 67
8.1 Modèle de tâche dépendante 67
8.2 Modèle de regroupement 68
8.3 Ordonnancement de tâches avec contraintes de précédence 71
8.4 Impact du regroupement sur l'ordonnançabilité 74

9 Minimisation du nombre de tâches 77
9.1 Espace des solutions et complexité 77
9.2 Adaptation de l'heuristique 78

10 Expérimentations 83
10.1 Génération des contraintes de précédence 83
10.2 Variation des paramètres de génération du graphe 85

Regroupement de tâches avec contraintes de précédence en contexte multiprocesseur 91

11 Ordonnancement Multiprocesseur 93
11.1 Architecture multiprocesseur 93
11.2 Ordonnancement multiprocesseur 94
11.3 Ordonnancement partitionné pour tâches avec contraintes de précédence 96

12 Regroupement de tâches multiprocesseur 101
12.1 Adaptation du problème monoprocesseur 101
12.2 Minimisation du nombre de tâches 102
12.3 Expérimentations 106

Conclusion et perspectives 109

Table des figures

1.1 Flight Application Software (FAS) 6
1.2 Modèle de tâche périodique 7
1.3 Contraintes de précédence représentées par un graphe acyclique dirigé (DAG) 8

2.1 Modèles des processus (tiré de [50]) 12

3.1 Exemple d'une tâche séquenceur avec $T_{cycle} = 30$, $t_{tic} = 5$, $R_1 = (1, 5, 5)$, $R_2 = (3, 10, 10)$ et $R_3 = (2, 20, 20)$ 20

4.1 Exemple de génération d'une tâche avec $u_i = 0.3$, $T_i = 10$, $d_1 = 0$ et $d_2 = 1$ 24
4.2 Description d'un ensemble de tâches au format BNF 25
4.3 Export d'un graphe de tâches depuis Graphviz 26

5.1 Diagramme d'exécution des tâches de l'ensemble du tableau 5.1 sous DM 35
5.2 Diagramme d'exécution des tâches de l'ensemble du tableau 5.1 sous EDF 37
5.3 Représentation graphique des dbf de chaque tâche de l'ensemble du tableau 5.1 et de leur cumul 39
5.4 Impact du regroupement de tâches dans le cas (5.3b) 39
5.5 Regroupement de tâches à coût nul (cas (5.3a)) 40

6.1 Illustration du principe de l'heuristique : regroupements successifs 48

7.1 Impact du regroupement sur le nombre de tâches en DM ($d_{min} = 0$, $d_{max} = 1$) . . . 58
7.2 Impact du regroupement sur le nombre de tâches en EDF ($d_{min} = 0$, $d_{max} = 1$) . . . 59
7.3 Comparaison des fonctions de coût en DM 59
7.4 Comparaison des fonctions de coût en EDF 60
7.5 Comparaison des fonctions de coût en DM en fonction de la densité (n = 300) 61
7.6 Comparaison des fonctions de coût en EDF en fonction de la densité (n = 300) . . . 61
7.7 Variation de la borne inférieure de génération des échéances sur le nombre de tâches après regroupement en DM (u = 0.80, n = 300) 62
7.8 Variation de la borne supérieure de génération des échéances sur le nombre de tâches après regroupement en DM (u = 0.80, n = 300) 63
7.9 Impact de la diminution du nombre de tâches sur les changements de contexte et les préemptions en DM (n max = 300, u = aléatoire, $d_{min} = 0$, $d_{max} = 1$) 64
7.10 Impact de la diminution du nombre de tâches sur les changements de contexte et les préemptions en EDF (n max = 300, u = aléatoire, $d_{min} = 0$, $d_{max} = 1$) 64

8.1 Regroupements infaisables représentés par des arcs rouges en pointillés ajoutés au graphe 68

8.2 Graphe de la figure 8.1 après regroupement des sommets (tâches) τ_2 et τ_5 en τ_{2_5} . . 69
8.3 Regroupement infaisable de τ_1 et τ_3 70
8.4 Effet du regroupement sur le diagramme d'exécution de $\mathcal{T}$ 75
8.5 Évolution des contraintes de précédence lors du regroupement 75

9.1 Espace de recherche en fonction de la topologie du graphe 77
9.2 Matrice d'adjacence du graphe de la figure 1.3 79

10.1 Exemple de génération d'un graphe de tâches par niveau 85
10.2 Impact de la densité des contraintes de précédence sur la diminution du nombre de tâches (n = 200, u aléatoire, $d_{min} = 0$, $d_{max} = 1$) à partir du modèle d'Erdös-Rényi 86
10.3 Impact du nombre de niveaux et de la probabilité des contraintes de précédence entre niveaux adjacents sur le nombre total de contraintes de précédence dans la génération par niveau 87
10.4 Impact de la probabilité des contraintes de précédence sur la diminution du nombre de tâches (n = 200, u = 0.7, $d_{min} = 0$, $d_{max} = 0.8$) à partir de la génération par niveau à 90% 88

11.1 Diagramme d'exécution des tâches d'un ensemble asynchrone sous EDF 100

12.1 Regroupement infaisable des tâches τ_j et τ_k du processeur m_1 102
12.2 Ensemble de tâches $\mathcal{R}$ exécuté sur deux processeurs 102
12.3 Ensemble de tâches $\mathcal{R}$ après regroupement des tâches τ_i et τ_j en τ_{ij} 103
12.4 Partitionnement du graphe de tâches de la figure 8.1 sur trois unités de calcul 104
12.5 Impact du taux d'utilisation sur le nombre total de tâches après regroupement (n = 200, p = 0.25, $d_{min} = 0$ et $d_{max} = 1$) 107

Table des notations

τ_i	i-ième tâche
C_i	Pire temps d'exécution de τ_i
D_i	Échéance relative de τ_i
T_i	Période d'activation de τ_i
O_i	Date de réveil du premier travail de τ_i
L_i	Laxité de τ_i
$o_{i.n}$	Date de réveil du n-ième travail de τ_i
$d_{i.n}$	Échéance absolue d'un n-ième travail de τ_i
$e(\tau_{i.n})$	Date de fin d'exécution du n-ième travail de τ_i
$s(\tau_{i.n})$	Date de début d'exécution du n-ième travail de τ_i
Φ	Assignation de priorités
$e_\Phi(\tau_{i.k})$	Date de fin d'exécution du n-ième travail de τ_i sous Φ
$s_\Phi(\tau_{i.k})$	Date de début d'exécution du n-ième travail de τ_i sous Φ
$\tau_i \rightarrow \tau_j$	τ_i précède τ_j
$\tau_i \prec \tau_j$	τ_i est un prédécesseur direct de τ_j
$\tau_i \triangleright \tau_j$	τ_i et τ_j sont dépendantes
$\tau_i \not\triangleright \tau_j$	τ_i et τ_j sont indépendantes
R_i	Pire temps de réponse de τ_i
u_i	Taux d'utilisation processeur de τ_i
U	Taux d'utilisation processeur total
H	Hyper-période
$\mathrm{hp}(i)$	Ensemble des tâches de priorité supérieure ou égale à τ_i privé de τ_i
$preds(i)$	Ensemble des prédécesseurs directs de τ_i
$succs(i)$	Ensemble des successeurs directs de τ_i
π_i	Priorité régulière (ou nominale) de τ_i
γ_i	Seuil de préemption de τ_i
D_i^*	Échéance relative encodée de τ_i
O_i^*	Date de réveil encodée de τ_i

Introduction

Contexte

Ce travail s'inscrit dans le contexte des systèmes informatiques embarqués. Ceux-ci ont la particularité d'êtres dédiés à des tâches spécifiques exercées de manière autonome. Le terme *système embarqué* englobe généralement la partie *matérielle* qui correspond à ses composants électroniques et la *partie logicielle* qui relève des instructions exécutées pour son contrôle. Ce manuscrit cible cette dernière catégorie et en particulier les applications *critiques* qui régissent par exemple le comportement des véhicules de l'aérospatiale, de l'automobile ou encore du matériel médical. En amont, ces systèmes disposent de capteurs afin de récupérer des données issues de leur milieu ambiant. Ces informations sont traitées et déterminent les interactions des actionneurs avec leur environnement en aval.

Ces dispositifs s'activent à des fréquences particulières et doivent impérativement terminer leurs missions dans le temps imparti. Cette caractéristique distingue les systèmes embarqués *temps réel* qui exigent que les résultats produits par le logiciel soient corrects du point de vue logique et qu'ils interviennent au moment requis.

La chaîne de conception des systèmes temps réel couvre les phases allant de la modélisation de la dynamique du système jusqu'à celle de l'implantation, c'est-à-dire la concrétisation de l'application vers du code machine. Les concepteurs du domaine de l'aérospatiale spécifient jusqu'à plusieurs milliers de fonctionnalités pour définir le comportement de leurs systèmes temps réel. L'implantation de ces fonctionnalités n'est pas directe et procède de plusieurs étapes. Elle suppose notamment une phase de déploiement des fonctionnalités vers un ensemble d'entités à grain à plus large, nommées tâches, supportables par les systèmes d'exploitation. Cette phase de déploiement exige le respect des propriétés fonctionnelles et temporelles du système. Elle est indispensable en raison des surcoûts liés au traitement et à la gestion mémoire d'un très grand nombre de fonctionnalités. C'est à ce niveau que se place les recherches effectuées à l'occasion de mon doctorat.

Contribution

Dans ce manuscrit, nous mettons en évidence les surcoûts engendrés par la programmation d'un grand nombre de tâches sur un système d'exploitation temps réel. Ceux-ci sont principalement causés par l'empreinte mémoire des tâches et par leur gestion sur le système. Nous nous intéressons aux méthodes qui permettent de limiter ces surcoûts et à cet effet, nous proposons de minimiser le nombre de tâches d'un système temps réel par des regroupements. Dans la littérature, le regroupement est souvent vu comme un problème d'allocation de traitements vers des ensembles de tâches à plus gros grain. Il est donc question d'associer un ensemble conséquent de traitements à un ensemble plus restreint de tâches, tout en conservant le comportement initial du système. Une des caractéristiques majeures d'un système temps réel est l'ordonnançabili-

té, c'est-à-dire sa propension à respecter ses contraintes de temps. En ce sens, nous veillons à respecter les contraintes de temps initiales du système et à maintenir son caractère ordonnançable. Par conséquent, nous examinons par la même l'impact du regroupement de tâches sur l'ordonnançabilité du système.

Nous affirmons que le regroupement de tâches est un sujet complexe qui combine la difficulté de problèmes d'optimisation combinatoire et d'ordonnancement. En conséquence, nous nous appuyons sur des méthodes de résolution approchées mais efficaces afin de minimiser le nombre de tâches d'un système temps réel, en garantissant le respect des contraintes de temps. Nous appliquons ces techniques de regroupement de tâches dans les contextes monoprocesseur et multiprocesseur. À travers les contraintes de précédence, nous prenons également en compte la possibilité que les tâches communiquent entre elles.

L'intégralité des solutions proposées sont implantées dans un outil. Celui-ci fait également fonction de plateforme d'essai pour démontrer l'efficacité des algorithmes développés dans ce travail, à travers plusieurs expérimentations.

Plan général

La première partie est dédiée à la conception et à la programmation des systèmes temps réel. Notre travail est à la frontière de ces deux sujets et cette partie introductive met en perspective les motivations qui ont conduit les recherches menées. Elle passe également en revue les travaux analogues et le prototype développé. Par ailleurs, elle se conclut par une annonce plus détaillée du contenu des contributions du manuscrit. En deuxième partie, nous posons les bases du regroupement de tâches, en particulier pour des tâches indépendantes dans le contexte monoprocesseur. Dans la troisième partie, nous examinons la situation où les tâches peuvent s'échanger des données, ce qui implique de considérer des relations de dépendance dans l'ordonnancement. Nous consacrons la quatrième et dernière partie à l'adaptation des techniques de regroupement au cas multiprocesseur, dans lequel plusieurs tâches peuvent être exécutées en parallèle.

Cadre

Le présent manuscrit esquisse un panorama de mes activités de recherche réalisées au sein de l'équipe Emeraude du laboratoire CRIStAL. Ces travaux de thèse ont été cofinancés par l'Université de Lille 1 et par la Métropole Européenne de Lille (MEL), dans le cadre du projet SUNRISE[1] ayant trait au développement des villes intelligentes.

[1] `http://smartcity-lgcge-cuaf.univ-lille1.fr/sunrise-smartcity.html`

Première partie

Motivations et état de l'art

Chapitre 1

Modélisation de systèmes temps réel

Ce premier chapitre est consacré à la spécification de systèmes temps réel. Il a pour objectif de caractériser le type de système visé dans cette thèse et à apporter des savoirs de base en ordonnancement temps réel. Ces connaissances seront complétées au fur et à mesure du manuscrit lorsque nécessaire, pour éviter une entrée en matière trop indigeste.

1.1 Définitions

1.1.1 Systèmes ciblés

Dans ce travail, nous nous intéressons à la conception de systèmes composés d'un grand nombre de traitements parfois appelés *nœuds* (près d'un millier dans certains cas). La figure 1.1 correspond au type d'applications considéré dans ce travail. Il s'agit d'une version simplifiée d'un système de contrôle de vol d'un satellite (FAS), le Véhicule de Transfert Automatique (ATV), conçu par Airbus Defence and Space pour le ravitaillement de la Station Spatiale Internationale (ISS). Les traitements effectués sur cet engin correspondent grossièrement à des sous-programmes et sont représentés dans la figure 1.1 par des boîtes rectangulaires. Les lignes verticales en pontillés distinguent en particulier les traitements relatifs aux entrées et aux sorties de l'application. D'une part, nous disposons de *capteurs*, dispositifs qui recueillent un phénomène de l'environnement et d'autre part d'*actionneurs*, organes qui engendrent un phénomène physique en retour. Ils font par exemple référence à une opération de mesure de la rotation angulaire par un gyroscope pour les capteurs ou au contrôle de l'orientation d'un panneau solaire pour les actionneurs. Le reste des traitements est dédié à des calculs intermédiaires. Les fréquences notées au-dessus des boîtes indiquent que les traitements fonctionnent de manière périodique. Les flèches reliant les boîtes dénotent des dépendances entre les traitements comme des communications de données. L'impératif de respect des contraintes fonctionnelles et temporelles des traitements classe le système donné en exemple dans la famille des systèmes temps réel.

1.1.2 Définitions

Il existe plusieurs interprétations et définitions des systèmes temps réel. Dans ce manuscrit, nous nous accorderons sur la définition suivante [25] :

Définition 1 (Système temps réel)
Un système temps réel est un système dont la correction ne dépend pas uniquement du résultat logique du calcul mais également du moment auquel les résultats sont produits.

Un système temps réel est donc un système qui assure que les traitements produisent des résultats conformes (fonctionnellement corrects) et au bon moment (temporellement corrects). Un résultat

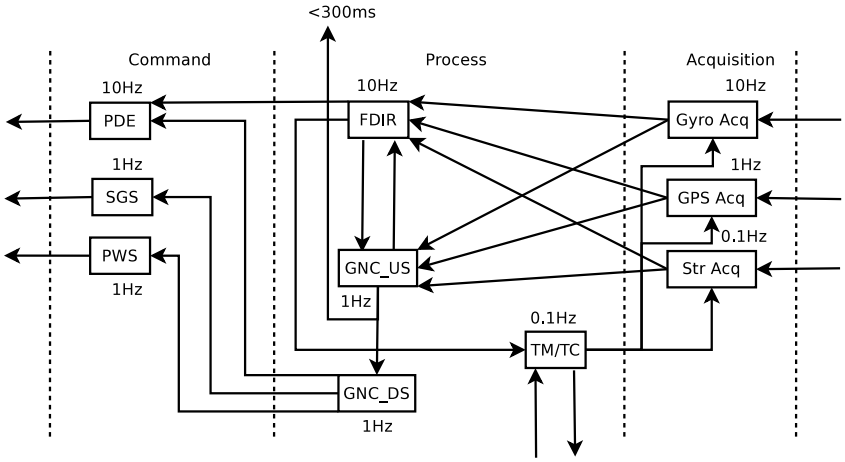


Fig. 1.1 : Flight Application Software (FAS)

logiquement correct mais ne respectant pas le délai imparti est considéré comme globalement incorrect au même titre qu'un résultat logiquement erroné. À la différence de la notion courante, le temps réel ne fait pas référence ici à l'instantanéité. Il ne s'agit pas nécessairement de produire un résultat au plus vite mais de le produire à la date attendue. Le plus souvent, il est nécessaire que le traitement se termine dans un certain délai, c'est-à-dire avant ou à un instant précis appelé **échéance**. Si un traitement produit un résultat à une date ultérieure à son échéance, on considère qu'une (son) échéance est manquée.

Différents niveaux de criticité

Les systèmes temps réel dits *critiques* correspondent aux systèmes **temps réel dur**, systèmes pour lesquelles il est intolérable qu'une échéance soit manquée au risque de causer des conséquences graves, telles que des blessures ou des pertes humaines. Les centrales nucléaires ou le guidage de missiles représentent de tels systèmes à haute criticité. Dans le domaine de l'informatique embarqué, l'automobile et l'aéronautique regorgent de systèmes critiques à l'image des équipements déclencheurs d'airbags ou des logiciels de contrôle de vol de satellite (similaire au FAS présenté en figure 1.1). Il est crucial que les résultats soient disponibles au moment voulu et un résultat obtenu trop tard est inutilisable, à l'instar d'un système anti-missile qui recevrait la position d'un objet volant avec du retard.

Tous les systèmes temps réel ne requièrent pas un déterminisme temporel aussi fort. Par exemple, un logiciel de diffusion de flux vidéo est tenu de produire un certain nombre d'images à intervalles réguliers. Le fait de manquer une ou plusieurs échéances ne provoque pas l'arrêt du système multimédia. La qualité de la vidéo est dégradée mais le service peut continuer de fonctionner sans risque. Il s'agit d'un **système temps réel mou**. Le meilleur service possible est visé (notion de *best effort*) mais les retards dans l'obtention des résultats sont tolérés car les conséquences ne sont pas dramatiques.

À la frontière entre les systèmes temps réel dur et mou, les systèmes **temps réel ferme** tolèrent une certaine proportion d'échéances manquées. Ils ne considèrent que les résultats obtenus à temps et sont liés à la notion de *qualité de service* (QoS).

Dans ce manuscrit, nous nous focaliserons sur les systèmes temps réel dur. La section qui suit a pour objectif de présenter les connaissances de bases relatives à l'ordonnancement des systèmes temps réel.

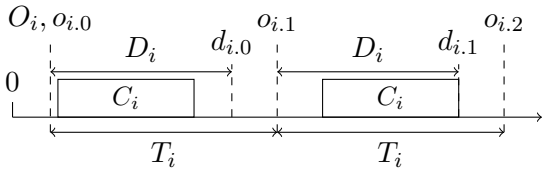


FIG. 1.2 : Modèle de tâche périodique

1.2 Notions d'ordonnancement temps réel

Dans la littérature temps réel, un traitement est souvent retranscrit dans la notion de *tâche*. L'ordonnancement désigne le planning d'exécution des tâches, c'est-à-dire l'ordre d'exécution des tâches au cours du temps. Dans les systèmes d'exploitation, l'ordonnanceur désigne l'algorithme qui détermine l'ensemble de ces décisions. Dans cette section, nous passons en revue les notions et les définitions de base de l'ordonnancement temps réel.

1.2.1 Modèle de tâche

Caractéristiques temporelles

Nous nous appuierons sur le modèle de tâche périodique fondateur de Liu et Layland [74] dans lequel une tâche τ_i est caractérisée par un *pire temps d'exécution* C_i, une *période d'activation* T_i, une *échéance relative* D_i et une date de réveil O_i. Un résumé des notations utilisées dans ce manuscrit est disponible dans la table des notations en page ix.

Le pire temps d'exécution (Worst Case Execution Time ou WCET en anglais) correspond à une estimation du plus long temps nécessaire à la fin de l'exécution de la tâche. L'influence des autres tâches sur celui-ci n'est pas prise en compte dans l'estimation car il est dépendant des décisions prises à l'ordonnancement. Il est généralement obtenu par des techniques d'analyse statique de code ou par des mesures expérimentales. Toutefois, la méthode d'estimation n'est pas abordée dans ce travail et le pire temps d'exécution est considéré comme un paramètre du modèle.

La période d'activation d'une tâche suppose qu'elle est exécutée de manière cyclique. Chaque exécution (répétition) d'une tâche est appelée *travail*. Chaque travail correspond à une instance de tâche. Dans l'exemple du FAS de la figure 1.1, la fréquence notée sur les boîtes rectangulaires correspond à la période d'activation du traitement. La n-ième activation d'un travail d'une tâche τ_i survient à la date $O_i + (n-1) \times T_i$. Pour les tâches dites *sporadiques*, la période correspond à l'intervalle minimal de temps qui existe entre deux activations successives d'une même tâche. Dans ce travail, nous nous cantonnons au modèle périodique strict.

L'échéance est dite relative car elle est relative à la période d'activation de la tâche. Elle fixe la date à laquelle l'instance de la tâche doit avoir terminé son exécution. Si la tâche n'a pas terminé son exécution avant cette date, il est dit qu'elle rate (ou manque) son échéance. Chaque échéance relative à un travail est appelée *échéance absolue*. L'échéance est :

implicite si l'échéance est égale à la période ($D_i = T_i$) ;

contrainte si elle est inférieure ou égale à la période ($D_i \leq T_i$) ;

arbitraire sinon, elle peut donc être supérieure à la période.

Lorsque les échéances sont implicites, les tâches sont souvent représentées par le couple (C_i, T_i). L'échéance absolue d'un n-ième travail d'une tâche τ_i notée $d_{i.n}$ survient à la date

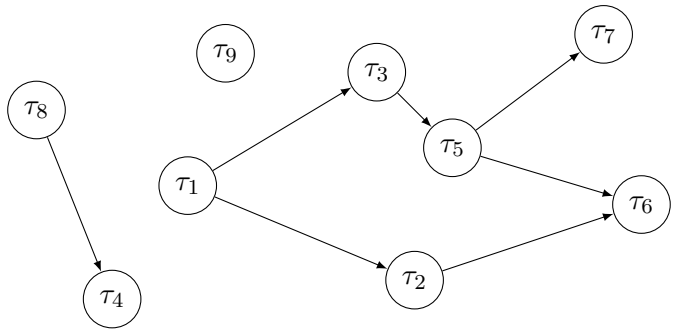


Fig. 1.3 : Contraintes de précédence représentées par un graphe acyclique dirigé (DAG)

$O_i + (n-1) \times T_i + D_i$.

La date de réveil O_i (ou *offset* en anglais) d'une tâche correspond à la date à laquelle la première instance de la tâche peut commencer à s'exécuter. Nous considérons que les tâches sont par nature *synchrones*, c'est-à-dire qu'elles sont prêtes à s'exécuter à la date 0 et qu'elles n'ont donc pas de date de réveil particulière (ou alors égale à 0). Cependant, pour les besoins de l'ordonnancement, des dates de réveil pourront être fixées ultérieurement.

Nous aurons également besoin dans la suite de la notion de **pire temps de réponse** d'une tâche R_i.

Définition 2 (Pire temps de réponse)
Le pire temps de réponse R_i indique la durée maximale qu'il existe entre le réveil d'une instance de la tâche τ_i et sa date de fin d'exécution.

Les caractéristiques temporelles citées plus haut sont utilisées pour décider de l'ordre d'exécution des tâches. Les tâches peuvent également être liées entre elles par des contraintes de précédence.

Contraintes de précédence

Les contraintes de précédence définissent un ordre partiel sur les tâches. Une tâche τ_i précède une tâche τ_j ($\tau_i \rightarrow \tau_j$) si τ_j ne peut commencer son exécution avant que τ_i ait terminé la sienne. Les contraintes de précédence sont dites *simples* lorsqu'elles ne s'appliquent qu'à des tâches de même période et *étendues* dans le cas général. Généralement, les contraintes de précédence transcrivent des communications de données entre les tâches. Par exemple, si τ_i produit une donnée que requiert et consomme τ_j alors $\tau_i \rightarrow \tau_j$. Dans ce travail, la notion de contraintes de précédence ne fera référence qu'aux contraintes de précédence simples. Les contraintes de précédence sont couramment représentées par un graphe acyclique dirigé (DAG pour Directed Acyclic Graph en anglais) dans lequel les tâches sont représentées par les sommets et les contraintes de précédence par les arcs. Dans le graphe de la figure 1.3 présenté en exemple, l'arc allant de τ_1 vers τ_2 signifie que $\tau_1 \rightarrow \tau_2$. τ_1 précède également τ_5 mais contrairement au premier cas il ne s'agit pas d'une précédence *directe* puisque les sommets ne sont pas adjacents dans le graphe. Notons que dans un tel graphe tous les sommets ne sont pas nécessairement connectés entre-eux, à l'image de la tâche τ_9 qui ne possède ni arc entrant ni arc sortant.

À partir des contraintes de temps et des contraintes de précédence, il va être question de définir un ordre d'exécution valide pour que tous les traitements puissent se faire en temps et en heure. Ces questions sont relatives à l'ordonnançabilité d'un système (ou ensemble) de tâches.

1.2.2 Ordonnancements basés sur les priorités

Il existe une grande variété d'approches d'ordonnancement, nous nous focaliserons sur les ordonnancements basés sur l'assignation de priorités aux tâches. Chaque instance de tâche (ou travail) se voit assigner une priorité de sorte à ce que le travail avec la priorité la plus grande soit toujours exécuté en premier. La manière d'assigner les priorités aux tâches est dans ce manuscrit indifféremment nommée algorithme d'ordonnancement ou politique d'ordonnancement. Nous distinguons différentes politiques d'ordonnancement basées sur les priorités :

Priorité fixée par tâche
Dans cette classe d'ordonnancement, une priorité est attribuée définitivement à une tâche. Cela signifie que toutes les instances de la tâches (les travaux) partagent et conservent la même priorité. Rate Monotonic (RM) [74] et Deadline Monotonic (DM) [71] appartiennent à cette classe d'ordonnancement.

Priorité fixée par travail
Dans cette catégorie, les priorités sont assignées de manière définitive à un travail. Pour une même tâche, ses différentes instances pourront se voir attribuer des priorités différentes mais la priorité accordée à un travail ne peut pas varier au cours du temps. La politique Earliest Deadline First (EDF) [74] illustre cette classe d'ordonnancement.

Priorité dynamique par travail
Lorsque la priorité d'une instance de tâche peut varier au cours du temps, il s'agit d'une d'assignation de priorité dynamique par travail à l'image de la politique d'ordonnancement Least Laxity First (LLF) [39].

1.2.3 Faisabilité, ordonnançabilité et optimalité

Dans cette section, nous définissons plusieurs notions théoriques relatives à l'ordonnancement. Par souci de concision, la définition de faisabilité est restreinte à l'étude des tâches indépendantes. Elle sera complétée dans la partie propre aux tâches liées par des contraintes de précédence.

Définition 3 (Faisabilité d'un ensemble de tâches indépendantes)
Un ordonnancement faisable est un ordonnancement dans lequel toutes les tâches respectent leurs échéances.

Formellement, soit $\mathcal{J}$ dénotant une ensemble non fini de travaux $\mathcal{J} = \{\tau_{i.k}, 1 \leq i \leq n, k \in \mathbb{N}\}$. Soit une assignation de priorités Φ, nous définissons deux fonctions $s_\Phi, e_\Phi : \mathcal{J} \to \mathbb{N}$, où $s_\Phi(\tau_{i.k})$ est la date de début et $e_\Phi(\tau_{i.k})$ la date de fin d'exécution de $\tau_{i.k}$ dans l'ordonnancement produit par Φ.

Définition 4
Soit $\mathcal{S} = (\{\tau_i\}_{1 \leq i \leq n})$ un ensemble de tâches et une assignation de priorités Φ. $\mathcal{S}$ est faisable *sous Φ si et seulement si : $\forall \tau_{i.k}, e_\Phi(\tau_{i.k}) \leq d_{i.k} \wedge s_\Phi(\tau_{i.k}) \geq o_{i.k}$*

Définition 5 (Faisabilité)
Un test de faisabilité est un test qui, pour un ensemble de tâches donné, retourne une réponse positive s'il existe un ordonnancement faisable. Si la réponse est négative alors aucun algorithme ne peut générer un ordonnancement faisable pour cette ensemble de tâches.

Définition 6 (Test d'ordonnançabilité)
Un test d'ordonnançabilité est un test qui, pour un ensemble de tâches et un algorithme d'ordonnancement donnés, retourne une réponse positive si l'algorithme produit un ordonnancement faisable.

Notons que la notion de faisabilité est plus générale que celle d'ordonnançabilité. Toutes les politiques d'ordonnancement citées plus haut sont *optimales* dans leurs classes d'ordonnancement.

Définition 7 (Optimalité)
Un algorithme d'ordonnancement $\mathcal{A}$ est optimal dans sa classe d'ordonnancement si, étant donné un ensemble de tâches ordonnançable par un algorithme quelconque de cette même classe, $\mathcal{A}$ peut ordonnancer cet ensemble de tâches.

Elles font également partie de la catégorie des algorithmes d'ordonnancement *préemptifs*.

Définition 8 (Algorithme d'ordonnancement préemptif)
Un algorithme d'ordonnancement est préemptif s'il autorise qu'une instance de tâche en cours d'exécution soit interrompue par une autre instance de tâche de priorité plus élevée.

Enfin, les algorithmes d'ordonnancement considérés dans ce travail sont *conservatifs*.

Définition 9 (Algorithme d'ordonnancement conservatif)
Un algorithme d'ordonnancement est conservatif s'il ne peut pas être inactif alors que des tâches sont prêtes à l'exécution.

Les algorithmes d'ordonnancement seront détaillés par la suite lorsqu'ils seront employés dans le contexte monoprocesseur. Ils servent de base aux algorithmes d'ordonnancement multiprocesseur.

Chapitre 2

Programmation de systèmes temps réel critiques

Le chapitre précédent était consacré à la modélisation de systèmes temps réel critiques. Il s'agit dans ce chapitre de faire le lien avec l'implantation de tels systèmes, puis d'étudier les contraintes du passage du modèle théorique à sa programmation.

2.1 Processus, threads et tâches

Nous avons expliqué dans le chapitre 1 que les traitements étaient modélisés sous forme de tâches. Dans cette section, nous nous intéressons aux notions de processus et de threads dans les systèmes d'exploitation classiques et à l'implantation de la notion de tâche dans les systèmes d'exploitation temps réel.

2.1.1 Processus et threads

Nous détaillons dans cette partie les notions de processus et de threads dans le domaine des systèmes d'exploitation.

Processus

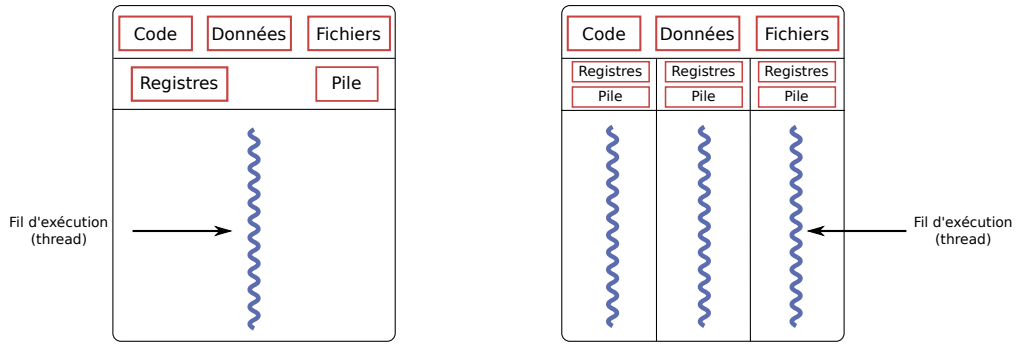
Un processus correspond à une instance d'un programme. Il s'agit de l'exécution du code d'un programme et de son contexte d'exécution. Un processus est représenté par un bloc de contrôle de processus (PCB pour Processus Control Block en anglais) qui contient généralement les informations suivantes :

L'état du processus : le processus peut être prêt à l'exécution, en cours d'exécution, bloqué, terminé, etc. ;

Le compteur ordinal : il contient l'adresse de la prochaine instruction à exécuter ;

Les valeurs des registres du processeur : elles permettent de rétablir le contexte du processus sur le processeur lorsque le processus a été interrompu et que l'on reprend son exécution. Le pointeur de pile et le compteur ordinal peuvent également faire partie de cette catégorie ;

Des informations sur la gestion mémoire : il s'agit des données propres à la manière dont la mémoire est gérée par le système, les pointeurs vers les segments de pile ou de données par exemple ;



(a) Modèle monothread (b) Modèle multithread

Fig. 2.1 : Modèles des processus (tiré de [50])

Des informations sur les entrées-sorties : elles correspondent aux périphériques d'entrées-sorties alloués au processus, aux descripteurs de fichiers, aux identifiants du processus, etc. ;

Des informations relatives à l'ordonnanceur : elles peuvent contenir des paramètres d'ordonnancement comme la priorité du processus mais aussi le temps d'exécution écoulé, etc.

La liste des éléments ci-dessus n'est pas exhaustive et peut varier en fonction de l'implantation du système.

Threads

Par défaut, chaque processus contient un fil d'exécution appelé *thread*. Cependant, posséder un unique fil d'exécution ne permet d'exécuter qu'un seul traitement à la fois (on parle d'un modèle *singlethread* en anglais). La plupart des systèmes d'exploitation modernes sont capables de gérer plusieurs fils d'exécution par processus (ils sont dits *multithread* en anglais). Par exemple, un navigateur web consiste en un seul et même processus mais il pourra gérer un fil d'exécution par onglet ouvert. Les différents fils d'exécution (threads) partagent généralement les données, le code du programme et les fichiers du processus. Toutefois, chaque thread possède ses propres copies des registres du processeur ainsi que sa pile d'exécution. La figure 2.1 expose la distinction entre processus et thread.

Tâches temps réel

Dans le modèle classique de programmation de système temps réel, chaque tâche correspond à une fonction qui boucle infiniment en l'attente d'un événement tel que la fin de sa période ou le dépassement son échéance [38]. Le squelette de code d'une tâche dans un langage de type C s'apparente au code suivant :

```
void task(){

  while(true){
  /* Fait quelque chose */
    waitForNextPeriod();
  }

}
```

Listing 2.1 : Squelette de code d'une tâche temps réel

Le comportement de la tâche est décrit dans le corps de la bouche `while`. Chaque tour de boucle correspond à une instance de la tâche. La fonction `waitForNextPeriod()` est exécutée à la fin de chaque tour de boucle et suspend la tâche jusque la prochaine période d'exécution.

Les threads sont souvent considérés comme des processus légers puisqu'ils partagent le même espace mémoire. L'abstraction de tâche dans la littérature temps réel s'apparente indifféremment à celle de processus ou de thread. Cependant, dans la plupart des implantations de systèmes d'exploitation temps réel (RTOS pour *Real-Time Operating System* en anglais) comme FreeRTOS, RTAI, VxWorks ou dans les langages dédiés au temps réel tels que Ada 2005, Java/RTSJ, C/Real-Time POSIX, la notion de tâche (comme définie dans le chapitre 1) est implantée à l'instar des threads.

Quelle que soit l'implantation choisie (processus ou thread) pour une tâche, les *changements de contexte* ont un coût non négligeable.

2.2 Changements de contexte et préemptions

Un système temps réel est composé d'un ensemble de tâches. L'ordonnanceur donne alternativement l'accès au(x) processeur(s) aux différentes tâches. Les tâches sont donc en concurrence pour l'accès à ce(s) dernier(s) et elles interfèrent les unes par rapport aux autres. Les passages d'une tâche à l'autre sont appelés des changements de contexte. Certains de ces changements de contexte sont dus à l'interruption de tâches par l'exécution d'autres, ce sont des préemptions.

2.2.1 Définitions

Chaque tâche possède un contexte d'exécution. Lorsque l'on passe de l'exécution d'une tâche (thread ou processus) à une autre, il est nécessaire d'enregistrer le contexte en mémoire et de charger le contexte de la nouvelle tâche, pour que le processeur dispose des informations nécessaires à son exécution. Il s'agit alors d'un changement de contexte. Si la tâche d'origine est exécutée derechef, il faudra restaurer (charger de nouveau) son contexte pour qu'elle puisse s'exécuter une fois de plus ou poursuivre son exécution. Un système temps réel connaît de nombreux changements de contexte pendant sa durée de fonctionnement. Bien que les temps de changement de contexte dépendent de l'implantation, le changement de contexte d'un processus vers un autre est généralement bien plus lent (cinq fois plus sur le système d'exploitation Oracle Solaris [50]) qu'un changement de contexte intervenant entre deux threads du même processus.

Ces changements de contexte se produisent dans deux situations :

- Soit la tâche courante τ_a a terminé son exécution normalement et une autre tâche τ_b va s'exécuter sur le processeur ;

- Soit, dans le cas où l'algorithme d'ordonnancement est préemptif, la tâche courante τ_a n'a pas terminé son exécution mais une tâche plus prioritaire τ_b va s'exécuter sur le processeur et interrompre l'exécution de τ_a.

Dans le second cas, nous dirons que la tâche τ_b *préempte* l'exécution de la tâche τ_a. En résumé, les préemptions provoquent des changements de contexte mais tous les changements de contexte ne sont pas dus aux préemptions.

On distingue les changements de contexte dits *explicites* des changements de contexte *implicites* ou *intrinsèques* dus aux préemptions [28].

2.3 Surcoûts d'un grand nombre de tâches

Les systèmes que nous visons sont composés d'un nombre important de tâches pouvant atteindre plus d'un millier. Leur grand nombre implique d'importants surcoûts liés à l'ordonnancement ou à l'occupation mémoire que nous étudierons dans cette section.

2.3.1 Surcoûts liés aux changements de contexte et aux préemptions

Dans cette section, nous examinons le coût des changements de contexte et en particulier de ceux causés par les préemptions.

Un changement de contexte nécessite donc de sauver le contexte (PCB) de la tâche courante puis de charger celui de la tâche arrivante, élue par l'ordonnanceur. D'autres coûts sont à prendre en compte [27] :

- **Les coûts pour l'ordonnanceur :** l'insertion de la tâche en cours d'exécution dans la file des tâches prêtes puis l'exécution de la tâche arrivante après changement de contexte ;
- **Les coûts liés au pipeline :** son nettoyage puis son remplissage lorsque la tâche arrivante s'exécute ;
- **Les coûts liés au cache :** pour recharger les lignes de caches évincées par la tâche arrivante après un défaut de cache.

Il y a une littérature importante sur l'estimation des délais induits par les défauts de cache lors des préemptions nommés CRPD (pour *Cache-Related Preemption Delays* en anglais) et sur leur prise en compte dans les analyses d'ordonnançabilité [69]. D'ailleurs, nous n'y retrouvons que peu souvent la distinction entre les changements de contexte explicites et implicites. Ceci provient du fait qu'il est généralement considéré que les changements de contexte explicites sont intégrés dans le paramètre de pire temps d'exécution C_i du modèle de tâche [28].

Connaître le nombre exact de changements de contexte implicites requiert de simuler l'ordonnancement des tâches [43] mais il existe des calculs de bornes supérieures [28]. Des bornes moins pessimistes sont disponibles pour des classes d'ordonnancement particulières, telles que celle des ordonnancements à priorité fixe par tâche [23].

Le comportement d'un ordonnanceur lors d'un changement de contexte a été décrit dans la section 2.2.1. Manipuler une file d'attente avec un grand nombre de tâches induit un coût additionnel (cf. [25]) puisqu'à chaque interruption d'horloge, la file d'attente est scannée et les tâches sont insérées dans la file des tâches prêtes à l'exécution, en fonction de leur date d'arrivée.

Globalement, le nombre de changements de contexte explicites augmente naturellement avec le nombre de tâches.

2.3.2 Impact sur la mémoire

Dans le pire cas, les algorithmes d'ordonnancement à priorité fixe par tâche assignent autant de priorités qu'il y a de tâches. Cette situation survient par exemple si les toutes les échéances relatives sont distinctes en DM ou si l'on applique l'algorithme d'Audsley [5]). En conséquence, le nombre de niveaux de priorité nécessaires à l'ordonnancement augmente naturellement avec un nombre de tâches important. Cela n'est pas négligeable puisque généralement les implantations d'ordonnanceurs sollicitent une file d'attente par niveau de priorité. Par ailleurs, plusieurs systèmes d'exploitation temps réel limitent le nombre maximal de priorités autorisées. C'est le cas de RTEMS [89] et de VxWorks [105] qui restreignent respectivement le nombre de niveaux de priorité maximal à 255 et 256. De plus, avoir pléthore de niveaux de priorités différents tend à augmenter la taille des piles si le nombre de préemptions augmente. Néanmoins, des techniques visant à diminuer le nombre de préemptions existent. Elles seront présentées dans le chapitre 3 relatif aux travaux connexes.

L'empreinte mémoire d'une tâche dépend grandement de l'implantation mais nous avons vu précédemment qu'elle est généralement composée d'espace réservé pour la sauvegarde des registres du processeur et d'une pile d'exécution. Les microcontrôleurs utilisés en temps réel sont dotés d'une mémoire de type RAM (pour *Random Access Memory* en anglais) limitée. Cette contrainte explique sans doute pourquoi de nombreux systèmes d'exploitation temps réel ne supportent qu'un nombre limité de tâches, à l'image de MicroC [66] de Micrium qui restreint le nombre de tâches à 256.

Des expérimentations préliminaires ont été menées sur un microcontrôleur temps réel Texas Instrument(TI) TMS570 embarquant le système d'exploitation temps réel FreeRTOS [49]. Elles révèlent que l'espace mémoire augmente linéairement avec le nombre de tâches en accord avec les valeurs théoriques annoncées. Ce travail a été réalisé par Mélanie Lelaure lors de son court stage auprès de notre équipe de recherche.

2.4 Regroupement de tâches

Les systèmes temps réel sont composés d'un nombre important de tâches, interagissant entre elles de manière concurrente pour l'accès aux ressources telles que la mémoire ou le processeur. Ces interférences provoquent une multitude de changements de contexte, en partie occasionnés par des préemptions. Du point de vue de la mémoire et de l'ordonnanceur, nous avons mis en exergue que ces changements de contexte étaient globalement coûteux. Aussi, la plupart des systèmes d'exploitation imposent des limites sur le nombre maximal de threads autorisés ainsi que sur le nombre de niveaux de priorité disponibles. En conséquence, il est nécessaire de regrouper plusieurs tâches dans un même thread de façon à rendre possible l'implantation de tels systèmes et plus généralement, afin de limiter les surcoûts causés par un grand nombre de threads.

Notre travail porte sur le regroupement de tâches, plus particulièrement sur la minimisation du nombre de ces tâches. Les systèmes temps réel imposent de respecter les contraintes de temps de ces tâches, c'est-à-dire que l'ensemble des tâches soient ordonnançables. Il faut donc veiller à ce que les regroupements effectués ne portent pas atteinte à cette ordonnançabilité du système. Dans cette thèse, nous attaquons le problème dans le cadre de tâches indépendantes et dépendantes, dans les contextes monoprocesseur et multiprocesseur. De quel manière regroupe-t-on les tâches ? Dans quelles conditions regroupe-t-on ces tâches ? Ce problème est-il difficile ? Comment veille-t-on à ce que l'ordonnançabilité soit préservée ? Voilà quelques questions auxquelles nous apportons des réponses dans ce manuscrit. Ces résultats sont implantés et vérifiés dans le cadre d'expérimentations.

Chapitre 3

Travaux connexes

Dans ce chapitre, nous passons en revue les différentes techniques similaires au regroupement de tâches, en particulier celles qui visent à placer des tâches ensemble pour accomplir l'implantation de systèmes temps réel. Plus généralement, nous présentons également des techniques qui visent à diminuer les surcoûts engendrés lors de l'implantation.

3.1 Systèmes distribués

Dans la littérature, le regroupement (alors appelé *clustering* en anglais) fait généralement référence aux méthodes d'allocation visant à distribuer des tâches sur des unités de calcul (des processeurs ou des cœurs). Cette notion diffère de la nôtre puisque dans le contexte des systèmes distribués, un regroupement correspond à un ensemble de tâches allouées à la même unité de calcul. Par exemple, les travaux [86, 2] ont pour objectif de minimiser les communications par le regroupement de tâches qui communiquent de manière importante. Les approches de [82, 55] regroupent des tâches en se basant sur les communications dans le but de réduire la taille du chemin critique (« makespan ») du système. Finalement, le nombre de tâches n'est cependant pas réduit.

3.2 Limitation des préemptions

Dans la littérature temps réel, les limites d'un système composé de tâches trop nombreuses sont souvent observées sous l'angle de l'ordonnancement, en particulier sur la manière de réduire le nombre de préemptions. En effet, comme énoncé plus tôt, avoir un nombre important de tâches peut nécessiter un grand nombre de niveaux de priorités et occasionner par la même, un nombre élevé de préemptions.

3.2.1 Seuil de préemption

La notion de seuil de préemption a été introduite par Xpress Logic, Inc. [67] dans le système d'exploitation temps réel ThreadX puis théorisée par Wang et Saksena [103]. Ce modèle concerne l'ordonnancement à priorité fixe par tâche et attribue une seconde priorité γ_i par tâche, qui correspond au seuil de préemption (supérieur ou égal à la priorité régulière π_i). L'ordonnanceur s'appuie sur les priorités régulières pour choisir l'ordre d'exécution des tâches. Cependant, chaque fois qu'une tâche est en cours d'exécution, c'est la priorité du seuil de préemption qui est considérée. Ainsi, cette tâche ne peut être préemptée que par des tâches ayant une priorité supérieure à son seuil de préemption ($\pi_j > \gamma_i$), ce qui a pour effet de globalement diminuer le nombre de préemptions. L'assignation de la priorité relative au seuil de préemption dure jusqu'à la fin de l'exécution du travail la tâche.

L'ordonnancement avec seuil de préemption est un compromis entre l'ordonnancement préemptif stricte (c'est-à-dire sans seuil de préemption) et l'ordonnancement non préemptif. En effet, le cas où chaque seuil de préemption est égal à la priorité régulière correspond au cas préemptif stricte et celui où chaque seuil de préemption est fixé à la priorité régulière maximale, au cas non préemptif.

Outre la réduction du nombre de préemptions, il a été démontré que cette démarche était capable d'ordonnancer des ensembles de tâches non ordonnançables selon les ordonnancements préemptifs strictes ou non préemptifs. Les auteurs dérivent plusieurs algorithmes capables de définir des seuils de préemptions de manière à obtenir un ensemble ordonnançable.

Saksena et Wang [92], concomitamment avec Davis et al. [38] soulignent qu'en fonction des priorités et des seuils assignés, il est possible que certaines tâches soient mutuellement non-préemptives. Cela signifie qu'il ne peut pas survenir de préemption entre ces tâches ($\pi_i \leq \gamma_j \wedge \pi_j \leq \gamma_i$). Il est alors considéré que ces tâches appartiennent au même groupe non-préemptif. Cette caractéristique présente l'intérêt de pouvoir diminuer le nombre de piles d'exécution requis. En effet, en mode préemptif stricte, il est nécessaire de disposer d'une pile par tâche pour pouvoir restaurer le contexte d'une tâche potentiellement préemptée par l'exécution d'une autre. Si nous avons l'assurance que certaines tâches ne peuvent mutuellement pas se préempter, nous pouvons associer la même pile d'exécution au groupe non-préemptif qu'elles forment.

À dessein également de limiter les préemptions, Baruah et al. ont introduit une technique de préemptions différées [10] (*Deferred Preemptions Scheduling ou DPS* en anglais) sous EDF. Chaque tâche se voit attribuer un paramètre q_i qui correspond à l'intervalle maximal pendant lequel elle peut s'exécuter de manière non-préemptive. Soit la durée maximale pendant laquelle un travail ne peut être préempté est calculée à partir de l'exécution d'un travail plus prioritaire, soit le programmeur spécifie explicitement dans le code de la tâche le début et la fin de la période non-préemptive par des primitives système. Requérant aussi l'instrumentation du code, la technique de Burns [24] consiste à découper une tâche en sous-tâches dans lesquelles les préemptions ne peuvent intervenir qu'aux dates de fin de celles-ci.

Ces techniques de limitations du nombre de préemptions permettent de répondre à une part des restrictions des systèmes d'exploitation temps réel. Dans la suite de l'état de l'art, nous verrons que les seuils de préemption sont souvent employés dans des approches plus globales de conception de systèmes embarqués. Elles ne sont néanmoins pas incompatibles avec le regroupement de tâches.

3.3 Méthodologies de conception de systèmes embarqués

3.3.1 Ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (IDM) désigne un domaine d'ingénierie logicielle dans lequel les applications sont décrites sous formes de modèles, d'abstractions de haut-niveau. L'IDM rassemble notamment des étapes de transformation de modèles et de génération de code.

Les langages de modélisation objet permettent de définir de manière abstraite, par notations graphiques ou textuelles les composants d'une application, d'un système. Le plus répandu d'entre eux en IDM est l'Unified Modeling Language (UML) [90] standardisé par l'organisme OMG. Il s'agit d'un langage généraliste pour lequel il existe des « profils », raffinements spécifiques au domaine d'application. À titre d'exemple, le profil MARTE [102] concerne la modélisation et l'analyse de systèmes embarqués temps réel. Plusieurs travaux considèrent la transformation de modèles abstraits de systèmes temps réel vers leur implantation en passant par une phase d'allocation de traitements atomiques vers des structures de plus gros grain. Cette étape est assimilable à une activité de regroupement.

Dans le champ de l'IDM, Mzid et al. [77] relèvent que le nombre de niveaux de priorités assignés lors de la conception excède le nombre autorisé par les systèmes d'exploitation temps réel. Ils proposent une méthode de regroupement des tâches qui vise à restreindre le nombre de niveaux de priorité. Une heuristique, proche de celle présentée en chapitre 6 dans le sens où elle regroupe les tâches deux à deux, s'y consacre. Le résultat n'étant pas jugé satisfaisant, les auteurs décrivent une formulation d'optimisation linéaire en nombres entiers (OLNE) du problème. Cette formulation a pour fonction objectif de maximiser le nombre de regroupements et de minimiser le taux d'utilisation. Leur modèle est assez similaire au nôtre à la différence qu'il autorise le regroupement de tâches de périodes harmoniques. Pour ce faire, les périodes des tâches doivent être multiples entre elles ($\frac{T_i}{T_j} = q$ avec $T_i \geq T_j$) et la période la plus courte est choisie pour le regroupement. Un compteur est alors inséré dans le code de la tâche pour exécuter les tâches aux fréquences adéquates. Conformément à leur modèle de regroupement, les auteurs considèrent que le regroupement des tâches harmoniques entraîne une hausse du taux d'utilisation. Ils visent à maximiser le nombre de regroupements tout en minimisant le taux d'utilisation du système.

Kim et al. [61] se basent sur le modèle des *transactions*. Une transaction est un graphe acyclique dirigé (DAG) de tâches de période unique sur lequel sont spécifiées des échéances correspondant à des contraintes de bout-en-bout. Un délai de bout-en-bout traduit le temps écoulé entre un *stimulus* et la *réaction* consécutive. Un stimulus correspond à un événement produit en entrée (depuis un capteur par exemple) et marque le début d'exécution d'une fonctionnalité. La réaction correspond à un événement produit en sortie (via un actionneur par exemple) et marque la fin d'exécution de la fonctionnalité concernée. Le délai de bout-en-bout est parfois également appelé *latence*. Les auteurs regroupent dans un premier temps les transactions qui partagent des ressources dans des threads « logiques » (tâches). Ensuite les threads logiques se voient attribuer des priorités fixes par tâche suivant l'algorithme d'Audsley [5] puis sont soumis à une analyse de pire temps de réponse. Après attribution de seuils de préemption maximaux [103], les threads logiques mutuellement exclusifs sont finalement regroupés dans des threads physiques (des threads niveau OS) suivant le principe de Saksena et al. [92] décrit en section 3.2.1.

En accord avec les spécifications du profil UML-RT [96], Saksena et al. [91] assignent des échéances et des priorités fixes aux éléments des transactions. Les auteurs décrivent ensuite sommairement un algorithme de type *séparation et évaluation* qui regroupe les éléments dans des tâches, en minimisant leur nombre tout en réduisant les communications. Dans un modèle où les fonctionnalités sont spécifiées par le biais de composants logiciels, Kodase et al. [64] s'inscrivent dans une démarche similaire à [91] en précédant les étapes d'assignation de priorité et de regroupement de tâches par une phase d'identification des transactions dans les composants logiciels. À la différence de [91], les auteurs s'orientent faire une solution approchée, le recuit simulé tout en optimisant les pires temps de réponse des transactions, le nombre de priorités distinctes et les communications.

Ces travaux ne considèrent que l'ordonnancement de priorité fixe par tâches et se restreignent au cadre monoprocesseur. Bien que le coût des communications soit souvent pris en compte, il n'y a généralement pas de contrainte de précédence considérée entre les threads.

3.3.2 AUTOSAR : runnable-to-task mapping

AUTOSAR

AUTomotive Open System ARchitecture (AUTOSAR) [7] est un partenariat regroupant des industriels de l'automobile ainsi que des équipementiers de systèmes embarqués temps réel. C'est également le nom de spécifications d'une architecture modulaire, qui vise à standardiser la production et la maintenance de logiciel embarqué pour l'automobile. Dans AUTOSAR, les composants logiciels sont spécifiés par des *SW-Component*. Chacun d'eux est composé d'un ou plusieurs *runnables* qui correspondent à des entités élémentaires exécutables, autrement dit à

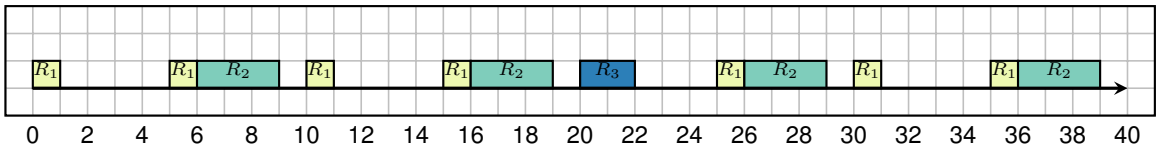


Fig. 3.1 : Exemple d'une tâche séquenceur avec $T_{cycle} = 30$, $t_{tic} = 5$, $R_1 = (1, 5, 5)$, $R_2 = (3, 10, 10)$ et $R_3 = (2, 20, 20)$

des tâches à grain très fin. Chacun des composants logiciels implante une fonctionnalité du système et communique vers l'extérieur via des ports. In fine, ces composants sont exécutés sur des unités de commande électronique (ECU). Dans l'automobile, les ECU correspondent à des microcontrôleurs. D'un point de vue abstrait, ils peuvent être vus comme des unités de calcul telles que les processeurs.

Runnable-to-task mapping

Dans AUTOSAR, l'étape qui consiste à associer les *runnables* à des threads est appelée *runnable-to-task mapping*. Il est question de former des threads (OS tasks) qui seront ensuite exécutés sur les unités de calcul (ECU). Cette phase permet donc déployer les nombreuses fonctionnalités vers un ensemble de tâches plus réduit (au niveau du système d'exploitation, c'est-à-dire des threads). Cette étape est décrite comme cruciale dans l'article de Scheickl et Rudorfer [94] du fait du nombre limité de threads supporté par les systèmes d'exploitation temps réel et des coûts des changements de contexte.

Dans le cadre de l'ordonnancement à priorité fixe sur monoprocesseur, Zeng et Di Natale [109] présentent une formulation OLNE très complète du problème d'association des runnables vers des tâches. L'ordre d'exécution des runnables à l'intérieur des tâches respecte l'ordre partiel initial défini sur les runnables. Ils proposent de regrouper des runnables avec des périodes multiples de celle de la tâche qu'ils forment et l'échéance des tâches est fixée sur la période. Leur formulation a pour objectif de minimiser la consommation de mémoire vive consommée par la pile en appliquant des seuils de préemption (cf. 3.2.1) tout en assurant la cohérence des données dans les communications. L'ordonnançabilité du système est vérifiée sur les runnables. Les paramètres des runnables dépendent de leur association avec les threads.

Monot et al. [76] proposent la combinaison de deux heuristiques afin d'appliquer la phase de *runnable-to-task mapping* pour des architectures multicœurs dans le cadre d'AUTOSAR. La première alloue les runnables sur les cœurs à partir des contraintes de localité et de la charge des cœurs. La méthode de partitionnement se base une heuristique inspirée du problème de *bin-packing* [58]. Pour plus d'informations sur les heuristiques de partitionnement, le lecteur pourra se référer au chapitre 11 qui a trait à l'ordonnancement multiprocesseur. La seconde construit un ordonnancement cyclique statique des runnables. Il s'agit de définir statiquement un plan d'exécution des tâches qui sera exécuté par une *tâche séquenceur* (sequencer task). Il existe une unique tâche séquenceur par cœur. La durée (taille) T_{cycle} de la tâche séquenceur est un multiple du plus grand commun diviseur (pgcd) des périodes des runnables qu'il contient. Le plus petit commun multiple (ppcm) des périodes des runnables est un multiple de la période d'activation de la tâche séquenceur T_{tic}. T_{cycle} est donc composée de $\frac{T_{cycle}}{T_{tic}}$ tranches (slots). L'activation des runnables au sein de la tâche séquenceur est obtenue par modification de leurs dates de réveil (offsets) de manière à répartir uniformément le taux d'utilisation par cycle. Si les périodes des tâches sont arbitraires, le plan d'exécution des tâches peut être très grand.

Leur travail revient à regrouper toutes les tâches d'un même cœur ensemble et à gérer statiquement l'ordonnancement à l'intérieur du regroupement, ce qui rapproche cette technique de l'ordonnancement hiérarchique [22]. Dans ce travail, il n'y a ni contrainte de précédence entre des

tâches allouées sur des cœurs différents (conformément aux recommandations d'AUTOSAR), ni de contrainte de précédence entre des tâches allouées sur le même cœur. Par ailleurs, les auteurs se restreignent aux tâches à échéances implicites.

Wozniak et al. [106] proposent une approche qui couvre les phases d'allocation des runnables vers des threads et d'assignation de priorité fixe par tâche. Le problème est formulé comme un problème d'optimisation prenant les délais de bout-en-bout et la consommation mémoire comme critères. Deux méthodes de résolution du problème d'optimisation posé sont proposées. La première est basée sur l'optimisation linéaire mixte en nombres entiers. Cette technique correspond à l'OLNE défini dans la section 6.4 à cela près que la première accepte également des nombres réels. Ces méthodes permettent d'obtenir des résultats exacts mais elles sont généralement limitées à des instances de problèmes de petites tailles. La seconde est basée sur un *algorithme génétique* (GA). Les algorithmes génétiques sont des méthodes approchées qui permettent d'obtenir des résultats corrects mais généralement sous-optimaux en un temps raisonnable. Cette contribution se situe dans le contexte monoprocesseur.

Le runnable-to-task mapping est une étape du travail de [111] dans le contexte monoprocesseur. Les auteurs proposent une variante des algorithmes génétiques. Le regroupement est restreint à des runnables qui ont des échéances égales aux périodes et seules les tâches de même période sont regroupées ensemble.

Dans le contexte multicœur, Faragardi et al. [47] proposent une méthode qui couvre les phases d'allocation des runnables vers les threads et d'ordonnancement de ceux-ci dans le but de minimiser les coûts de communication. Les auteurs affirment prendre en compte les contraintes de précédence et présentent une méthode approchée basée sur une variante du recuit simulé. A posteriori, une étape de raffinement consistant à regrouper les tâches de même période allouées sur les mêmes cœurs est appliquée. Dans leur modèle, chacun des runnables est inclus dans une transaction. Chacune des transactions est caractérisée par une échéance et les auteurs font l'hypothèse que les périodes sont égales aux échéances. Concrètement, chaque transaction correspond à une tâche qui sera allouée sur un des cœurs et il n'existe pas de contrainte de précédence entre les transactions, mais uniquement entre les runnables d'une même transaction. Ainsi, il s'agit d'une allocation de transactions sur des processeurs plutôt que d'une allocation de runnables sur ces derniers. Dans notre travail, nous considérons des contraintes de précédence entre les tâches et nous utilisons le modèle de tâche périodique qui diffère du modèle des transactions.

3.4 Langages synchrones

Les langages synchrones sont des langages dédiés à la programmation de systèmes réactifs. Ils se basent sur l'hypothèse synchrone [15] qui énonce qu'un système idéal produit ses sorties synchroniquement (en même temps) que ses entrées. Autrement dit, il est considéré que le temps de réaction du système est nul. Dans le modèle synchrone, le traitement d'une donnée acquise en entrée doit être terminé avant la nouvelle acquisition. Les langages synchrones sont très présents dans les systèmes embarqués munis de capteurs et d'actionneurs mais ne permettent pas d'assigner des contraintes de temps réel car ils font abstraction du temps. Les travaux de Curic [35] visent à intégrer des contraintes de temps dans le langage synchrone équationnel Lustre[56]. Lors de l'implantation du langage étendu, le code Lustre exprimé par des nœuds est décomposé sous forme de tâches. Curic [35] exprime plusieurs stratégies de décomposition où différents sous-nœuds peuvent être regroupés dans des tâches s'ils ont des activations communes, mais séparés s'ils sont parallélisables. Ces heuristiques sont néanmoins très spécifiques à la structure du langage étudié et au paradigme synchrone telles que les recommandations d'Alras et al. dans [3].

Les travaux de Pagetti et al. [80] ciblent l'implantation multitâches de programmes synchrones multipériodiques. Il est question d'associer les différents éléments d'un programme vers des tâches temps réel mais le regroupement de tâches n'est pas considéré.

Dans [93], Santinelli et al. proposent de réduire le nombre de tâches en contexte monoprocesseur. Les auteurs regroupent des tâches avec des contraintes de précédence et leur objectif principal est de faire face à la complexité importante des tests d'ordonnançabilité lorsque le nombre de tâches est important. Plusieurs méthodes de regroupement se basant sur la topologie d'un graphe de tâches sont présentées. Les contraintes de temps considérées sont limitées à des ensembles de tâches monopériodiques et leur modèle se base sur les contraintes de latence alors que nous visons des ensembles multipériodiques se basant sur des échéances.

Conclusion

Les travaux cités ayant trait à la conception de systèmes embarqués sont les plus proches des objectifs de cette thèse. Ils ont généralement pour objectif d'allouer des tâches à grain fin vers des threads et de minimiser les communications tout en garantissant l'ordonnançabilité du système, dans les contextes monoprocesseur ou multiprocesseur pour des ordonnancements à priorité fixe par tâche (requis par AUTOSAR par exemple). Dans notre recherche, nous avons également investigué l'ordonnancement à priorité fixe par travail. De plus, certains travaux se limitent à des échéances implicites ce qui a pour effet de faciliter grandement les analyses d'ordonnançabilité. Souvent, le modèle utilisé est celui des transactions ou celui des tâches indépendantes et les contraintes de précédences entre threads ne sont pas considérées. Aussi, il est fréquent que le problème soit entièrement traité comme un problème d'optimisation. Soit à l'aide de méthodes approchées telles que les algorithmes génétiques ou le recuit simulé, soit à l'aide de méthodes exactes basées sur l'optimisation linéaire. À notre connaissance, il n'y pas d'autres travaux qui s'intéressent à l'impact direct du regroupement sur l'ordonnançabilité.

Chapitre 4

Prototype

Dans ce chapitre, nous présentons le prototype programmé dans le cadre de nos travaux. Tous les algorithmes proposés dans ce travail ont été implantés dans un outil disponible en ligne[1]. Cet outil a été programmé en Scala [79]. Scala est un langage de programmation moderne multi-paradigmes qui combine notamment les aspects objets et fonctionnels. Scala est traduit vers du « bytecode Java » exécuté sur une machine virtuelle Java (JVM). Nous profitons de ce chapitre pour annoncer le plan du manuscrit.

4.1 Vue d'ensemble

Le prototype couvre les aspects suivants :

- l'implantation d'algorithmes de tests d'ordonnançabilité ;
- l'implantation des méthodes de regroupement pour tâches avec ou sans contrainte de précédence en monoprocesseur et multiprocesseur ;
- des générateurs d'ensembles de tâches indépendantes et dépendantes ;
- des méthodes d'import d'ensembles de tâches et d'export, pour la visualisation ou l'utilisation dans des simulateurs d'ordonnancement.

Les différentes parties du prototype sont indépendantes. Par exemple, les tests d'ordonnançabilité, ainsi que les fonctions d'import et d'export d'ensembles de tâches ont été mises au point pour les besoins du regroupement de tâches mais ils peuvent être utilisés indépendamment de celui-ci.

4.2 Génération de tâches

Dans la littérature temps réel, il est courant de générer de nombreux ensembles de tâches aléatoirement, de façon à étudier empiriquement les résultats d'algorithmes d'ordonnancement. À la manière de Buttazo [29], nous pouvons par exemple chercher à comparer le nombre de préemptions engendrées par Rate Monotonic ou EDF en fonction du facteur d'utilisation processeur. En plus du paramètre du facteur d'utilisation processeur, il est possible de faire varier l'ensemble des périodes utilisées ainsi que le placement des échéances. Généralement, la procédure de génération de tâches suit les étapes suivantes :

[1] `https://github.com/abertout/ttc`

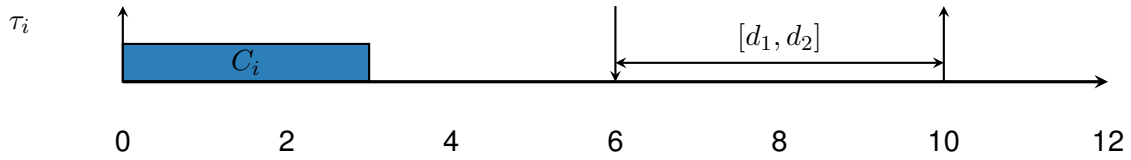


Fig. 4.1 : Exemple de génération d'une tâche avec $u_i = 0.3$, $T_i = 10$, $d_1 = 0$ et $d_2 = 1$

Méthode	Contexte
UUnifast [22]	Monoprocesseur
UUnifast-Discard [36]	Multiprocesseur
RandFixedSum [100, 45]	Multiprocesseur

Tab. 4.1 : Méthodes de génération de taux d'utilisation par tâche implantées

Test	Classe	Condition	Type	Complexité
Test d'Audsley [4]	FPT (DM)	Suffisante	Analyse de temps de réponse	$\mathcal{O}(n)$
Test de Devi [40]	FJP (EDF)	Suffisante	Booléen	$\mathcal{O}(n)$
RTA [59]	FPT (DM)	Exacte	Analyse de temps de réponse	$\mathcal{O}(n)$
Test de Spuri [99]	FJP (EDF)	Exacte	analyse de temps de réponse	$\mathcal{O}(n\mathcal{T}\mathcal{L}^2)$
Test de Guan [54]	FJP (EDF)	Exacte	analyse de temps de réponse	$\mathcal{O}(n + \mathcal{L}^2)$
Offset analysis [83]	FJP (EDF)	Suffisante	Booléen (pour tâches asynchrones)	PP

Tab. 4.2 : Tests d'ordonnancement implantés

1. nous générons un ensemble $\{u_i, ..., u_n\}$ de taux d'utilisation (méthodes détaillées en sections 7.1.1 et 12.3.1) ;
2. nous choisissons un ensemble de périodes $\{T_i, ..., T_n\}$ (cf. section 7.1.2) ;
3. nous établissons le pire temps d'exécution par tâche à partir des paramètres de taux d'utilisation et de période puisque $u_i \times T_i = C_i$;
4. nous plaçons aléatoirement l'échéance entre le pire temps d'exécution C_i et la période T_i (méthode décrite en section 7.1.3).

La figure 4.1 illustre la génération d'une tâche suivant la procédure décrite plus haut. Dans la figure 4.1, les flèches vers le haut définissent des activations de la tâche τ_i tandis que les flèches vers le bas correspondent aux échéances. Ces symboles conservent la même signification dans tous les diagrammes de tâches de ce manuscrit.

La génération des contraintes de temps pour les tâches peut être combinée avec la génération de contraintes de précédence. Plusieurs techniques de génération des contraintes de précédence sont implantées dans le prototype, nous reviendrons plus en détail sur ce cas en section 10.1.

Le tableau 4.1 récapitule les méthodes de génération de taux d'utilisation développées dans ce travail. Elles seront détaillées lorsqu'elles seront employées dans les différentes sections du manuscrit relatives aux expérimentations.

4.3 Tests d'ordonnançabilité

Les tests implantés dans notre outil sont décrits dans le tableau 4.2. Dans la colonne « Complexité », PP signifie pseudo-polynomial. Notons qu'il est aisé d'ajouter de nouveaux tests dans le prototype. La procédure est décrite dans la documentation associée.

```
taskSetDecl = taskSetStmt {sep taskSetStmt};
taskSetStmt = taskDecl | precedDecl | comment;
sep = ("," | ";"| "\t" | "\r" | "\n" | "\s" | "\f")
{"," | ";"| "\t" | "\r" | "\n" | "\s" | "\f"};
precedDecl = simplePrecedDecl | nPrecedDecl;
taskDecl = taskId "(" digit {digit} "," digit {digit} "," digit {digit}
["," digit {digit}] ")";
taskId = ( letter | digit) {letter | digit};
onePrecedDecl = taskId "->" taskId;
nPrecedDecl = taskId "->" "(" listTaskId ")";
listTaskId = taskId {"," taskId};
comment = "\*" {anyCharacter |"\n" | "\r"} "*\";
letter = "a".."z" | "A".."Z";
digit = "0".."9";
```

FIG. 4.2 : Description d'un ensemble de tâches au format BNF

Dans le tableau 4.2, FTP correspond à la catégorie des ordonnancements à priorité fixe par tâche (*Fixed-Task Priority*) et FJP à celle des ordonnancements à priorité fixe par travail (*Fixed-Job Priority*). n est le nombre de tâches de l'ensemble, $\mathcal{T}$ la période maximale de l'ensemble et $\mathcal{L}$ est la taille maximale de la *période d'activité*. Nous reviendrons sur cette notion en chapitre 5.

4.4 Fonctions d'import et d'export

4.4.1 Import d'ensembles de tâches

Le prototype inclue un analyseur lexical de manière à spécifier un ensemble de tâches. Le Listing 4.1 illustre une spécification de l'ensemble de tâches de la figure 1.3 dans le format d'entrée décrit en figure 4.2.

```
tau1(24, 500, 600),tau2(65, 700, 700, 5),tau4(682, 2771, 10500, 0, -1),
tau1 -> tau2,
tau1 -> tau3,
tau5(136, 600, 600, 0, -1),tau3(41, 450, 450, 0, -1),
tau6(24, 600, 600, 0, -1), tau7(65, 700, 700, 0, -1)
tau8(682, 2771, 10500, 0, -1),
tau9(136, 600, 600, 0, -1)
tau3 -> tau5
tau5 -> (tau6,tau7)
tau2 -> tau6
tau8 -> (tau4)
```

Listing 4.1 : Spécification possible de l'ensemble de tâches représenté en figure 1.3

4.4.2 Fonctions d'export

Simulateurs d'ordonnancement

La simulation est un outil essentiel pour évaluer le comportement d'un ensemble de tâches associé à un algorithme d'ordonnancement. D'aucuns voudront déterminer si le système est

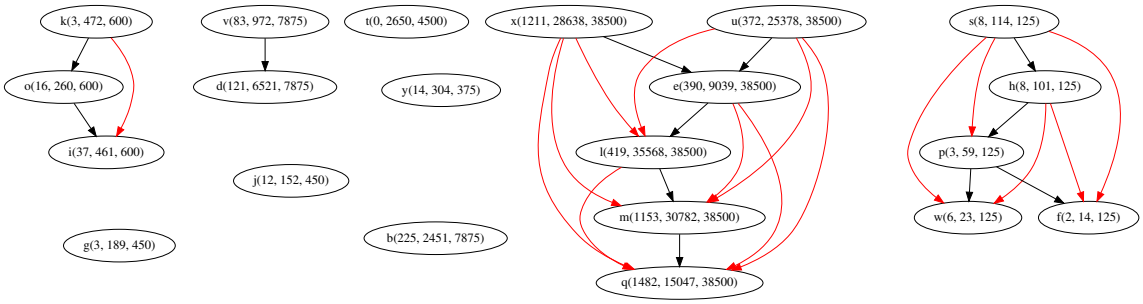


Fig. 4.3 : Export d'un graphe de tâches depuis Graphviz

ordonnançable, d'autres souhaiteront compter le nombre de préemptions ou encore mesurer le pire temps réponse d'une tâche. Pour ce faire, de nombreux simulateurs d'ordonnancement existent et autorisent la visualisation des diagrammes d'exécution des tâches. Le prototype implante des exportations d'ensembles de tâches vers les simulateurs SimSo [31] et Cheddar [97].

Notons que SimSo a également été utilisé pour évaluer les écarts du nombre de préemptions et de changements de contexte avant et après regroupement, lors d'expérimentations présentées en chapitre 7.

Visualisation de graphe de tâches

Il est pratique de pouvoir visualiser la topologie d'un graphe généré aléatoirement ou tout simplement de vérifier le comportement des algorithmes. Le prototype est doté d'une fonction de génération vers le langage de description de graphe dot [65]. Il fait partie de la suite Graphviz [2] qui fournit les outils pour obtenir des représentations graphiques de ces spécifications de graphe.

Formulation en OLNE

À titre de comparaison avec les méthodes approchées développées plus tard dans notre travail, nous proposons une formulation du problème dans les termes de l'optimisation linéaire en nombre entiers (OLNE). Pour un ensemble de tâches et un algorithme d'ordonnancement donnés (DM ou EDF), il est possible à l'aide du prototype d'exporter une instance du problème de regroupement pour les solveurs lpsolve [16] et CPLEX [34]. Ces exports consistent en la génération de séries de variables et de contraintes linéaires compatibles avec les solveurs cités.

Paquet `rtsched` pour LaTeX

La plupart des diagrammes d'exécution de tâches de ce manuscrit ont été conçus à l'aide du paquet LaTeX `rtsched` [73]. Ce paquet permet de dessiner facilement des diagrammes de tâches pour des articles, cours ou présentations à l'aide de simples primitives. `rtsched` dans sa version 1.0 utilisait la bibliothèque graphique `PSTricks` qui produit des documents de type `postscript`. Or `PSTricks` n'est pas compatible avec les outils plus récents pour LaTeX à la différence du tandem de bibliothèques `TikZ/PGF`. En effet, ces derniers produisent essentiellement des documents au format `pdf` (`pdfLaTeX`, `luaLaTeX`, etc.). J'ai donc proposé une réécriture du paquet pour qu'il fonctionne désormais avec `TikZ/PGF`. Cette nouvelle version (2.0) a été validée par l'auteur du paquet et mise en ligne. J'espère que cette contribution profitera à la communauté temps réel.

[2] `http://www.graphviz.org/`

4.5 Annonce du plan

Nous avons abordé dans ce chapitre le prototype développé au cours de notre travail et présenté d'un point de vue pratique les fonctionnalités implantées. Ce prototype recouvre les techniques de regroupement de tâches exposées à travers le plan suivant et fait office de banc d'essai pour les éprouver.

4.5.1 Partie 2 : Regroupement de tâches indépendantes en monoprocesseur

La partie 2 constitue la base du travail sur le regroupement de tâches. Nous y présentons en premier lieu le modèle de regroupement de tâches indépendantes en monoprocesseur. Ainsi, nous présentons comment les tâches sont regroupées et quel est l'impact de ce regroupement sur l'ensemble de tâches, en particulier sur son ordonnançabilité. Sous certaines conditions, il est possible de regrouper des tâches sans porter atteinte à l'ordonnançabilité de l'ensemble visé. Sinon, il est nécessaire de vérifier à nouveau l'ordonnançabilité. À cet effet, nous rappelons des tests d'ordonnançabilité de la littérature pour les algorithmes d'ordonnancement à priorité fixe par tâche (DM) et à priorité fixe par travail (EDF).

Nous décrivons ensuite la complexité du regroupement de tâche. Le regroupement de tâches est un problème difficile, raison pour laquelle nous nous orientons vers une solution approchée dite heuristique. Cette heuristique est de type best-first, c'est-à-dire qu'elle utilise un critère de décision pour décider localement vers quelle solution orienter la recherche. Ce critère de décision est également appelé fonction de coût. La méthode de regroupement implantée tente de générer un ensemble de tâches minimal à partir d'un ensemble initial, d'un test d'ordonnançabilité et d'une fonction de coût. Les fonctions de coût disponibles sont détaillées dans le chapitre 6.

De nombreux problèmes d'optimisation sont formulés au moyen de l'optimisation linéaire en nombre entier (OLNE). À titre de comparaison, nous formulons en OLNE le problème de regroupement de tâches avec des tests exacts pour DM (RTA) et EDF (PDA) dans la section 6.4.

Les expérimentations mettent en valeur l'efficacité du regroupement en DM et en EDF. Nous les avons produites en générant des tâches aléatoirement à l'aide du prototype. La procédure de génération de tâches et les méthodes implantées pour y parvenir sont décrites dans la section 4.2 du présent chapitre. Les paramètres choisis sont indiqués dans la section des expérimentations. Les différents résultats ayant trait au regroupement de tâches indépendantes en monoprocesseur ont été publiés dans les travaux [20, 18, 19].

4.5.2 Partie 3 : Regroupement de tâches avec contraintes de précédence en monoprocesseur

En partie 3, nous étudions l'évolution du regroupement de tâches lorsque des contraintes de précédence sont spécifiées entre les tâches. Tout d'abord, l'ajout des contraintes de précédence dans le modèle des tâches modifie les conditions de regroupement. En particulier, il apparaît que certains regroupements ne sont pas faisables au regard de la topologie du graphe. Nous présentons ensuite une méthode de la littérature qui permet de vérifier l'ordonnançabilité d'un ensemble sous contraintes de précédence à l'aide des tests destinés aux tâches indépendantes. Il est alors possible d'adapter l'heuristique de la partie relative aux tâches indépendantes en ne considérant que les regroupements qui respectent l'ordre des contraintes de précédence. Nous présentons finalement des expérimentations qui mettent en valeur l'efficacité du regroupement en tâches dépendantes, notamment en variant la densité des contraintes de précédence ainsi que la profondeur maximale du graphe de tâches.

4.5.3 Partie 4 : Regroupement de tâches avec contraintes de précédence en multiprocesseur

Les architectures modernes sont en grande majorité dotées de plusieurs unités de calcul que nous nommerons multiprocesseur par simplification. En premier lieu, nous présentons les différentes stratégies d'ordonnancement des tâches sur des architectures multiprocesseur. Deuxièmement, nous décrivons une solution de regroupement de tâches pour la stratégie d'ordonnancement partitionné. Celle-ci se base sur trois principes : le partitionnement de tâches sur plusieurs unités de calcul, l'encodage des contraintes de précédence pour vérifier l'ordonnançabilité et le regroupement de tâches sur chacune des partitions. Enfin, nous expérimentons les performances du regroupement de tâches dépendantes en multiprocesseur.

Dans ce chapitre, nous avons présenté le prototype développé pour expérimenter nos techniques de regroupement de tâches ainsi que le fil conducteur de ce manuscrit de thèse. Dans le chapitre suivant nous traitons des premiers travaux réalisés qui concernent le regroupement de tâches indépendantes en contexte monoprocesseur.

Deuxième partie

Regroupement de tâches indépendantes en contexte monoprocesseur

Chapitre 5

Modèle et ordonnançabilité d'un regroupement

En premier lieu dans ce chapitre, nous définissons les modèles utilisés pour le regroupement de tâches en monoprocesseur. Il s'agit de choisir les paramètres de manière à ce que les tâches continuent de respecter leur échéance et leur comportement initiaux à l'intérieur du regroupement. Nous justifions les choix effectués pour chacun des paramètres.

Dans un second temps, nous présentons une partie de l'état de l'art de l'ordonnancement monoprocesseur relatif aux ordonnancements à priorité fixe par tâche et à priorité fixe par travail. En particulier, nous passons en revue les algorithmes d'ordonnancement utilisés dans le cadre du regroupement de tâches ainsi que les tests d'ordonnançabilité associés et leurs complexités. Nous étudions ensuite l'impact du regroupement sur l'ordonnançabilité d'un ensemble de tâches dans les conditions énoncées. En effet, quel que soit l'algorithme d'ordonnancement choisi, l'ordonnançabilité de l'ensemble de tâches doit être préservée après regroupement. Finalement, nous examinons les conditions dans lesquelles il n'est pas nécessaire de vérifier de nouveau l'ordonnançabilité d'un ensemble de tâches après regroupement.

5.1 Modèle de tâche temps réel

Le modèle utilisé ici est le modèle de tâche périodique de Liu et Layland défini au chapitre 1 sans contrainte de précédence. Un système consiste donc en un ensemble de tâches temps réel synchrone $\mathcal{S} = (\{\tau_i(C_i, D_i, T_i)\}_{1 \leq i \leq n})$. Nous notons $\tau_{i.k}$ la $(k+1)^{i\text{-ème}}$ $(k \geq 0)$ instance, ou travail de τ_i. Le travail $\tau_{i.k}$ est réveillé à la date $o_{i.k} = kT_i$. Chaque travail $\tau_{i.k}$ doit être terminé avant son échéance absolue $d_{i.k} = o_{i.k} + D_i$.

Un ensemble de tâches est ordonnançable si et seulement si le pire temps de réponse R_i de chaque tâche est inférieur ou égal à son échéance relative.

5.2 Modèle de regroupement

5.2.1 Sémantique du regroupement

Regrouper dans cet ordre des tâches τ_i et τ_j consiste à exécuter en séquence, dans une même tâche τ_{ij}, le comportement de τ_i suivi du comportement de τ_j. τ_{ij} représente une seule et unique tâche, il n'y a donc pas de préemption possible entre les tâches τ_i et τ_j dont elle est issue. Le squelette de code 5.1 pourrait être celui d'un thread implantant la tâche τ_{ij} dans un langage de type C.

```
void task(){

  while(true){
  /* Comportement de la tâche τi */
  /* Comportement de la tâche τj */
    waitForNextPeriod();
  }

}
```

Listing 5.1 : Squelette de code d'un regroupement de tâches

5.2.2 Paramètres du regroupement

Regrouper des tâches τ_i et τ_j, où $D_i \leq D_j$, produit un regroupement τ_{ij} avec les paramètres suivants :

$$C_{ij} = C_i + C_j \tag{5.1}$$

$$T_{ij} = T_i = T_j \tag{5.2}$$

$$D_{ij} = \begin{cases} D_j & \text{si } (D_j - C_j \leq D_i) \vee (R_j - C_j, \leq D_i) \quad (5.3a) \\ D_i & \text{sinon.} \quad (5.3b) \end{cases}$$

Sans perte de généralité, $D_i \leq D_j$ n'est pas une contrainte mais une simplification pour exprimer les paramètres de regroupement par rapport à la tâche de plus grande échéance.

Ordre dans le regroupement

Notons que dans le cas (5.3a), exécuter τ_{ij} nécessite d'exécuter séquentiellement τ_i puis τ_j. Ci-après, $\tau_{i'}$ et $\tau_{j'}$ désignent respectivement les parts de τ_i et τ_j dans τ_{ij}. À contrario, dans le cas (5.3b), l'ordre dans le regroupement n'a pas d'importance puisque nous verrons plus loin que les deux échéances initiales sont respectées dans l'ordonnancement produit.

Pire temps d'exécution du regroupement

Dans l'équation (5.1), nous considérons que le pire temps d'exécution (WCET) du regroupement correspond à la somme des temps d'exécution. La somme des temps d'exécution est une borne supérieure sur le pire temps d'exécution du regroupement. En effet, le WCET de chaque tâche couvre le cas où les deux tâches sont exécutées en séquence et quel que soit leur ordre d'exécution. En pratique, le WCET du regroupement peut être moindre que la somme des deux WCET initiaux car les codes des deux tâches partagent le même thread. Il est ainsi possible d'éviter certaines opérations logicielles et matérielles (vider le pipeline du processeur par exemple). Notons qu'il est nécessaire que la somme des temps d'exécution C_{ij} soit inférieure ou égale à l'échéance du regroupement et donc supérieure ou égale à s*a laxité* L. La laxité correspond à la différence (ou marge) restante entre le pire temps d'exécution et l'échéance. ($L_i = D_i - C_i$). En effet, un système contenant une tâche dont le pire temps d'exécution est supérieur à son échéance est trivialement non ordonnançable.

Période du regroupement

Nous regroupons uniquement des tâches de même période (cf. équation (5.2)). Dans l'industrie, des tâches de périodes différentes sont parfois regroupées ensemble, particulièrement lorsque les fonctionnalités qu'elles traduisent interagissent considérablement, afin de réduire les

communications comme expliqué dans [95]. Lever cette limitation rendrait le regroupement plus complexe puisque cela imposerait de produire un ordonnancement à deux niveaux, le premier entre les tâches et le second à l'intérieur d'un regroupement. Cette technique fait référence aux travaux sur l'ordonnancement hiérarchique [22] et sur la planification de tâches hors-ligne abordée dans le chapitre 3 sur les travaux connexes. Nous n'avons pas étudié cette possibilité mais nos expérimentations ont montré que le nombre de périodes différentes (si raisonnable) n'a pas d'impact majeur sur l'efficacité des techniques de regroupement en monoprocesseur.

Échéance du regroupement

Dans la suite, le cas (5.3a) fait référence au regroupement utilisant (nous dirons regroupement sur) l'échéance D_j et le cas (5.3b) au regroupement sur l'échéance D_i. Nous choisissons l'échéance D_{ij} (cf. équations (5.3a) et (5.3b)) de manière à ce que les deux tâches du regroupement respectent leurs échéances initiales respectives après regroupement. Plus formellement :

Théorème 1
Soit $\mathcal{S} = (\{\tau_x(C_x, D_x, T_x)\}_{1 \leq x \leq n})$ *et* $\mathcal{S}' = (\mathcal{S} \setminus \{\tau_i, \tau_j\} \cup \{\tau_{ij}\})$ *deux ensembles de tâches synchrones ainsi que deux tâches* τ_i *et* τ_j *avec* $D_i \leq D_j$ *et* $T_i = T_j$. *Soit* τ_{ij} *la tâche issue du regroupement de* τ_i *et* τ_j *et* Φ *une assignation de priorités.*

$$\mathcal{S}' \text{ est ordonnançable sous } \Phi \Rightarrow \forall \tau_{ij.k}, \begin{cases} e(\tau_{i'.k}) \leq d_{i.k} \\ s(\tau_{i'.k}) \geq o_{i.k} \end{cases}$$

Preuve

Respect des dates de réveil
(Cas (5.3a) *et* (5.3b)*)*

$$\begin{gathered} \mathcal{S}' \text{ ordonnançable sous } \Phi \Rightarrow \forall \tau_{ij.k}, s(\tau_{ij.k}) \geq o_{ij.k} \\ \text{or } o_{ij.k} = o_{i.k} = o_{j.k} \\ \text{tandis que } s(\tau_{ij.k}) \geq s(\tau_{i'.k}) \text{ et } s(\tau_{ij.k}) \geq s(\tau_{j'.k}) \\ \text{donc } \begin{cases} \forall \tau_{ij.k}, s(\tau_{i'.k}) \geq o_{i.k} \\ \forall \tau_{ij.k}, s(\tau_{j'.k}) \geq o_{j.k} \end{cases} \end{gathered}$$

Respect des échéances
(Cas (5.3a)*)*
Trivialement, les travaux de $\tau_{j'}$ *respectent leurs échéances absolues. Les travaux de* $\tau_{i'}$ *respectent leurs échéances absolues lorsque* $D_j - C_j \leq D_i$:

$$\begin{gathered} \mathcal{S}' \text{ ordonnançable sous } \Phi \Rightarrow \forall \tau_{ij.k}, e(\tau_{ij.k}) \leq d_{ij.k} \\ \tau_i \text{ précède } \tau_j \text{ dans } \tau_{ij} \Rightarrow e(\tau_{i'.k}) + C_j \leq d_{ij.k} \\ \text{Comme } D_j \leq D_i + C_j,\ e(\tau_{i'.k}) + C_j \leq d_{i.k} + C_j \\ \text{En conséquence } e(\tau_{i'.k}) \leq d_{i.k} \end{gathered}$$

Si $R_j - C_j \leq D_i$ *:*

$$\begin{aligned} \mathcal{S}' \text{ ordonnançable sous } \Phi \Rightarrow \forall \tau_{ij.k}, e(\tau_{ij.k}) &\leq o_{ij.k} + R_{ij} \\ e(\tau_{i'.k}) + C_j &\leq o_{ij.k} + R_{ij} \\ \text{Comme } R_{ij} = R_j,\ e(\tau_{i'.k}) + C_j &\leq o_{ij.k} + R_j \\ \text{Comme } R_j \leq D_i + C_j,\ e(\tau_{i'.k}) + C_j &\leq o_{ij.k} + D_i + C_j \\ e(\tau_{i'.k}) + C_j &\leq d_{i.k} + C_j \\ \text{En conséquence } e(\tau_{i'.k}) &\leq d_{i.k} \end{aligned}$$

(Cas (5.3b)*)* $D_i \leq D_j$ *et* $D_{ij} = D_i$ *donc les travaux* $\tau_{i'}$ *et* $\tau_{j'}$ *respectent leurs échéances absolues initiales.*

Généralisation

Le modèle présenté est appliqué au regroupement de deux tâches puisque nous travaillons de manière incrémentale. Néanmoins, les définitions énoncées plus haut peuvent être étendues au regroupement de plus de deux tâches.

Dans le cas (5.3b), l'échéance minimale des tâches considérées est retenue. Sinon, les tâches triées dans l'ordre croissant des échéances respectent transitivement la condition de l'équation (5.3a). Par exemple, soit trois tâches τ_i, τ_j et τ_k avec $D_i \leq D_j \leq D_k$ (indistinctement sur D_k ou R_k, ici D_k a été choisie), nous avons :

$$\begin{aligned} D_k - C_k &\leq D_j \\ D_k - (C_k + C_j) &\leq D_i \end{aligned}$$

Dans ce chapitre, nous avons présenté ce que constituait le regroupement de tâches en fonction du comportement et des contraintes de temps de chacune. Nous assurons à l'aide du théorème 1 que le respect des conditions (5.3a) et (5.3b) garantit les contraintes de temps initiales des tâches.

5.3 Ordonnancement monoprocesseur

Dans cette section, nous détaillons les différents algorithmes d'ordonnancement préemptif à assignation de priorité utilisés dans ce travail. Pour chacun d'eux, nous passons en revue plusieurs analyses (ou tests) d'ordonnançabilité de la littérature. L'objectif de cette section est aussi de les distinguer afin d'utiliser les tests les plus adéquats pour vérifier l'ordonnançabilité d'un ensemble de tâches après regroupement. Nous nous restreignons aux systèmes synchrones avec échéances contraintes. Un test d'ordonnançabilité est dit *suffisant* si tous les ensembles de tâches qu'il juge ordonnançables le sont de fait. Certains ensembles jugés non ordonnançables par des tests suffisants peuvent donc l'être en réalité. Un test d'ordonnançabilité est dit *nécessaire* si tous les ensembles de tâches qu'il juge non ordonnançables sont de fait non ordonnançables. Les tests nécessaires ne permettent donc pas de déterminer le caractère ordonnançable d'un ensemble. Un test à la fois *nécessaire* et *suffisant* est dit exact. Dans ce travail, nous nous reposons uniquement sur des tests suffisants ou exacts puisqu'ils garantissent que les ensembles jugés ordonnançables le sont bel et bien. Ainsi, l'utilisation d'un test suffisant assure un regroupement de tâches correct mais éventuellement moins efficace qu'avec un test exact. Par regroupement « moins efficace », nous entendons un regroupement qui contient un nombre plus élevé de tâches.

Les tests les plus simples sont basés sur la notion de taux d'utilisation processeur.

Tâche	C_i	D_i	T_i
τ_1	2	6	6
τ_2	3	4	7
τ_3	3	15	15

TAB. 5.1 : Ensemble de tâches utilisé en exemple

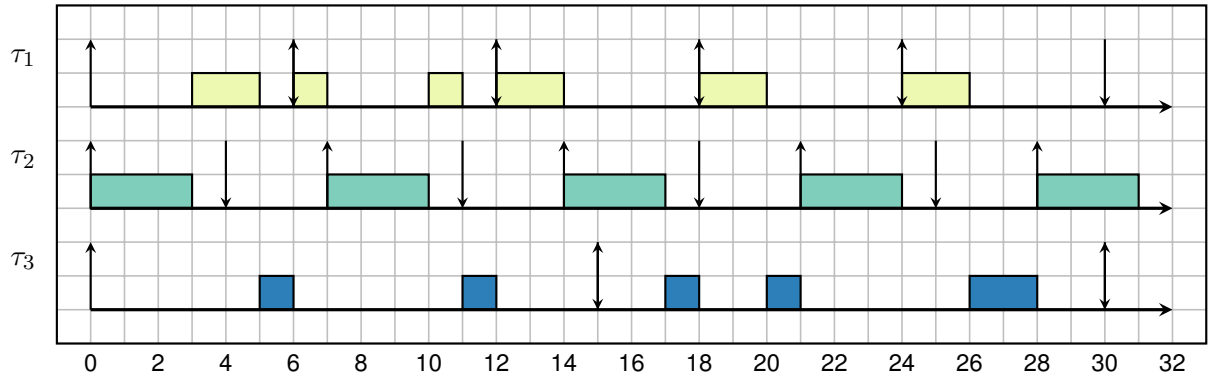


FIG. 5.1 : Diagramme d'exécution des tâches de l'ensemble du tableau 5.1 sous DM

Définition 10 (Taux d'utilisation processeur)
Le taux d'utilisation processeur u_i correspond à la charge de travail d'un processeur pour un ensemble de tâches donné :

$$u_i = \sum_{i=1}^{n} \left(\frac{C_i}{T_i} \right)$$

Chacun des algorithmes est illustré par un diagramme d'exécution des tâches basé sur l'ensemble de tâches du tableau 5.1.

5.3.1 Deadline Monotonic

Deadline Monotonic (DM) est un algorithme d'ordonnancement à priorité fixe par tâche introduit en 1982 par Leung et Whitehead [71]. DM dérive de Rate Monotonic (RM) [74], algorithme suivant lequel la priorité d'une tâche est inversement proportionnelle à sa période. À la différence de RM, où les échéances sont égales aux périodes ($D_i = T_i$), DM considère des échéances contraintes ($D_i \leq T_i$) et la priorité d'une tâche est fixée comme il suit : plus l'échéance d'une tâche est courte, plus sa priorité est élevée.

DM est un algorithme d'ordonnancement optimal [71] dans la catégorie des ordonnancements à priorité fixe par tâche. Le diagramme de la figure 5.1 décrit l'ordre d'exécution des tâches de l'ensemble du tableau 5.1 sous DM. Le lecteur observera que l'ensemble de tâches n'est pas ordonnançable sous DM puisque la tâche τ_3 ne parvient pas à exécuter la totalité de son pire temps d'exécution, avant ou à la date de son échéance.

Analyse d'ordonnançabilité

Dans le cas des algorithmes d'ordonnancement à priorité fixe par tâche, les approches analytiques de test d'ordonnançabilité sont basées sur le concept de l'*interférence*. Puisque les priorités sont fixées pour toutes les instances des tâches, il est aisé de savoir quelles sont les tâches susceptibles d'interférer sur les autres : ce sont les tâches de priorité supérieure. L'interférence correspond au temps d'exécution passé à exécuter des tâches de plus haute priorité qu'une autre

tâche prête à s'exécuter. Par exemple, dans le diagramme de la figure 5.1, la tâche τ_3 ne respecte pas son échéance à cause de l'interférence que les tâches τ_1 et τ_2, plus prioritaires, exercent sur elle. Une analyse d'ordonnançabilité aurait permis de détecter que cet ensemble n'était pas ordonnançable sans avoir à visualiser le diagramme d'exécution des tâches.

Audsley et al. proposent un test suffisant [4] basé sur l'interférence I sous l'hypothèse que les tâches soient triées par ordre décroissant des échéances relatives :

$$\forall i, (1 \leq i \leq n) : \frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1 \text{ où } I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil \cdot C_j \tag{5.4}$$

Ce test suffisant a une complexité en $\mathcal{O}(n)$ et peut être considéré comme une analyse pessimiste du temps de réponse puisqu'il le surestime. Pour obtenir sa valeur exacte, nous nous tournerons vers des tests exacts d'analyse de temps de réponse.

Analyse de temps de réponse

L'analyse de temps de réponse repose sur la notion d'instant critique.

Théorème 2 (Instant critique)

L'instant critique d'une tâche de priorité i intervient lorsque toutes les tâches de priorité supérieure ou égale à i s'activent simultanément avec la tâche étudiée.

En ordonnancement à priorité fixe par tâche pour tâches synchrones, l'instant critique survient à $t = 0$. Il s'agit alors du pire scénario d'exécution des tâches puisque toutes les tâches sont réveillées simultanément.

Dans le cas des algorithmes d'ordonnancement à priorité fixe par tâche, Joseph et Pandya [59] ont proposé une technique d'analyse de temps de réponse (RTA pour Response Time Analysis en anglais). Le pire temps de réponse R_i d'une tâche τ_i est basé sur le concept de *level-i busy period*. Intuitivement, la « level-i busy period » est l'intervalle de temps pendant lequel un processeur exécute des tâches de priorité supérieure ou égale à la tâche τ_i considérée. RTA calcule pour chaque tâche le pire temps de réponse désigné par R_i. L'équation pour trouver R_i est basée sur le principe suivant : si une tâche τ_i est réveillée à la date t où t est un instant critique, alors il doit y avoir assez de temps pour que τ_i ainsi que tous les travaux de priorité supérieure se terminent dans l'intervalle $(t, Ri]$. Le test calculant le pire temps de réponse R_i d'une tâche consiste en une recherche de point fixe pour :

$$R_i = C_i + \sum_{j=\text{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{5.5}$$

Ce test se calcule en considérant l'équation (5.5) comme une suite définie par récurrence où $R_i^0 = C_i$ [6]. Bien qu'en théorie la complexité de ce test soit pseudo-polynomiale, l'analyse de temps de réponse est en pratique un test efficace pour les ordonnancements à priorité fixe par tâche. Notons que l'équation (5.5) converge uniquement si $u_i \leq 1$.

5.3.2 Earliest Deadline First

Earliest Deadline First (EDF) est un algorithme d'ordonnancement à priorité fixe par travail. Contrairement à DM où les priorités sont assignées statiquement à chaque tâche (donc identiques à toutes les instances (travaux) de la tâche), EDF attribue la plus grande priorité au travail dont l'échéance absolue est la plus proche. La priorité d'une tâche va donc varier au cours du temps.

EDF a été introduit par le papier fondateur de Liu et Layland [74] en 1973 et son optimalité démontrée par Dertouzos [39] en 1974. Le diagramme de la figure 5.2 décrit l'ordre d'exécution des tâches de l'ensemble du tableau 5.1 sous EDF. Le lecteur constatera que l'ensemble est ordonnançable sur l'intervalle représenté par le diagramme de la figure 5.2.

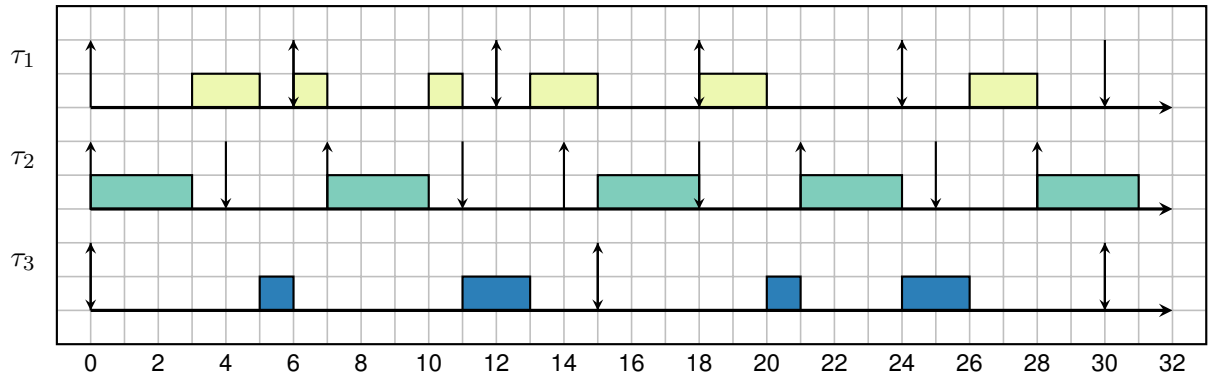


FIG. 5.2 : Diagramme d'exécution des tâches de l'ensemble du tableau 5.1 sous EDF

Analyse d'ordonnançabilité

EDF admet une condition nécessaire d'ordonnançabilité basée sur l'utilisation processeur :

$$\forall i, (1 \leq i \leq n) : u_i \leq 1 \tag{5.6}$$

Dans le cas des échéances implicites (où $D_i = T_i$), l'équation 5.6 constitue un test exact.

Le test de Devi [40] constitue une condition suffisante d'ordonnançabilité pour EDF avec une complexité en $\mathcal{O}(n)$. Compte tenu d'un ensemble initialement trié par ordre croissant des échéances, il vérifie la condition suivante :

$$\forall i, (1 \leq i \leq n) : \sum_{j=1}^{i} \frac{C_j}{T_j} + \frac{1}{D_i} \cdot \sum_{j=1}^{i} \left(\frac{T_j - min(T_j, D_j)}{T_j} \right) \cdot C_j \leq 1 \tag{5.7}$$

Le test vérifie pour chaque tâche qu'une sur-approximation de l'interférence des autres tâches ne dépasse pas l'intervalle compris entre le pire temps d'exécution et l'échéance de celle considérée. Cette borne supérieure est calculée au moyen de la demande processeur que nous détaillons plus loin.

Analyse de temps de réponse

L'analyse de temps de réponse sous EDF est plus complexe puisque le pire temps de réponse de chaque tâche n'intervient pas lorsque les tâches sont toutes réveillées à un instant critique. Spuri [99] a proposé un algorithme de calcul du temps de réponse pour EDF de complexité exponentielle. Plus récemment, Guan [54] a avancé une analyse de temps de réponse de complexité légèrement moins élevée et plus efficace pour les hauts taux d'utilisation. L'algorithme est également moins âpre à implémenter et repose sur l'analyse de la demande processeur (PDA pour *Processor Demand Analysis* en anglais).

Analyse de la demande processeur

La complexité élevée de l'analyse de temps de réponse favorise l'utilisation de tests exacts booléens sous EDF basés sur la *demande processeur*. La demande processeur a été introduite par Baruah et al. [13]. Deux fonctions permettent de déterminer la demande processeur sur un intervalle de temps $[t_1, t_2]$:

- La demande processeur des tâches dont les travaux sont réveillés dans l'intervalle $[t_1, t_2]$ nommée *request function* (rf) en anglais.
- La demande processeur des tâches dont les travaux sont réveillés dans l'intervalle $[t_1, t_2]$ et qui ont leurs échéances absolues dans $[t_1, t_2]$, nommée *demand function* (df) en anglais.

Formellement, la df d'un ensemble de tâches sur un intervalle $[t_1, t_2]$ est définie par :

$$\mathrm{df}(t_1, t_2) = \sum_{j=1}^{i} \left\{ \max\left(\left\lfloor \frac{t_2 - O_i - D_i}{T_i} \right\rfloor - \left\lceil \frac{t_1 - O_i}{T_i} \right\rceil + 1, 0 \right) \right\} \cdot C_i \tag{5.8}$$

C'est la df qui est principalement utilisée dans les tests pour algorithmes d'ordonnancement à priorité fixe par travail. En résumé, la df permet de connaître le temps cumulé nécessaire à exécuter les travaux réveillés dans $[t_1, t_2]$ qui doivent être terminés avant ou à t_2.

Intuitivement, un ensemble de tâches est faisable si et seulement si :

$$\forall\ 0 \leq t_1 < t_2 : \mathrm{df}(t_1, t_2) \leq t_2 - t_1 \tag{5.9}$$

EDF est un algorithme d'ordonnancement optimal donc être faisable en assignation à priorité fixe par travail signifie ordonnançable sous EDF.

En d'autres termes, le temps d'exécution cumulé demandé par l'ensemble de tâches ne peut jamais être supérieur au temps disponible qui correspond à la longueur de l'intervalle $[t_1, t_2]$.

Il a été prouvé par Baruah et al. [14] qu'un ensemble de tâches est faisable si et seulement si :

$$U = 1 \text{ et } \forall\ 0 \leq t_1 < t_2 \leq O_{max} + 2 \cdot H : \mathrm{df}(t_1, t_2) \leq t_2 - t_1 \tag{5.10}$$

où O_{max} est la date de réveil maximal et H l'hyper-période de l'ensemble de tâches. L'hyper-période est égale au plus grand commun diviseur (pgcd) des périodes, c'est-à-dire $H = \mathrm{pgcd}(T_1, ..., T_n)$.

Baruah et al. ont montré dans le même travail qu'un ensemble de tâches synchrones est faisable si et seulement si :

$$\forall\ L \leq L^*, \mathrm{df}(0, L) \leq L \tag{5.11}$$

où L est une échéance absolue et L^* le premier temps creux dans l'ordonnancement. Un temps creux correspond à un moment où le processeur n'est occupé par aucune exécution de tâches. La df correspond alors à la *demand bound function* (dbf) et $\mathrm{df}(0, L) = \mathrm{dbf}(L)$. Pour l'exemple, la figure 5.3 représente le tracé des dbf de chacune des tâches de l'ensemble du tableau 5.1 ainsi que leur cumul. L'axe des abscisses correspond aux échéances absolues de l'intervalle observé et l'axe des ordonnées aux valeurs de la dbf. Remarquons que sur l'intervalle observé, la dbf ne dépasse jamais le temps disponible t, l'ensemble de tâches est donc ordonnançable sous EDF sur l'intervalle concerné. Le challenge des tests basés sur la dbf comme la méthode QPA [110] consiste à réduire la taille de la fenêtre d'observation et le nombre des dates auxquelles la dbf devra être calculée.

Pour rappel, dans notre travail, nous nous basons uniquement sur les tests suffisants ou exacts car ils assurent qu'un ensemble jugé ordonnançable l'est de fait.

5.4 Impact du regroupement

Remarque 5.4.1 *Dans le cas* (5.3b)*, $\mathcal{S}$ ordonnançable sous $\Phi \not\Rightarrow \mathcal{S}'$ ordonnançable suivant l'algorithme d'ordonnancement qui a produit Φ.*

Autrement dit, un ensemble de tâches peut devenir non ordonnançable après regroupement. En effet, nous remarquons en figure 5.4b que la tâche τ_x manque sa première échéance à la date 9 après regroupement des tâches τ_i et τ_j, alors qu'il était initialement ordonnançable comme l'illustre la figure 5.4a. La tâche τ_x a été regroupée sur l'échéance de τ_i. L'échéance de τ_i étant plus courte que celle de τ_x, τ_j est devenue plus prioritaire que τ_x. L'exécution de la tâche τ_x est donc repoussée et elle n'a pas plus le temps nécessaire pour s'exécuter avant son échéance.

En conséquence, il est impératif de vérifier l'ordonnançabilité de l'ensemble de tâches après regroupement dans le cas (5.3b).

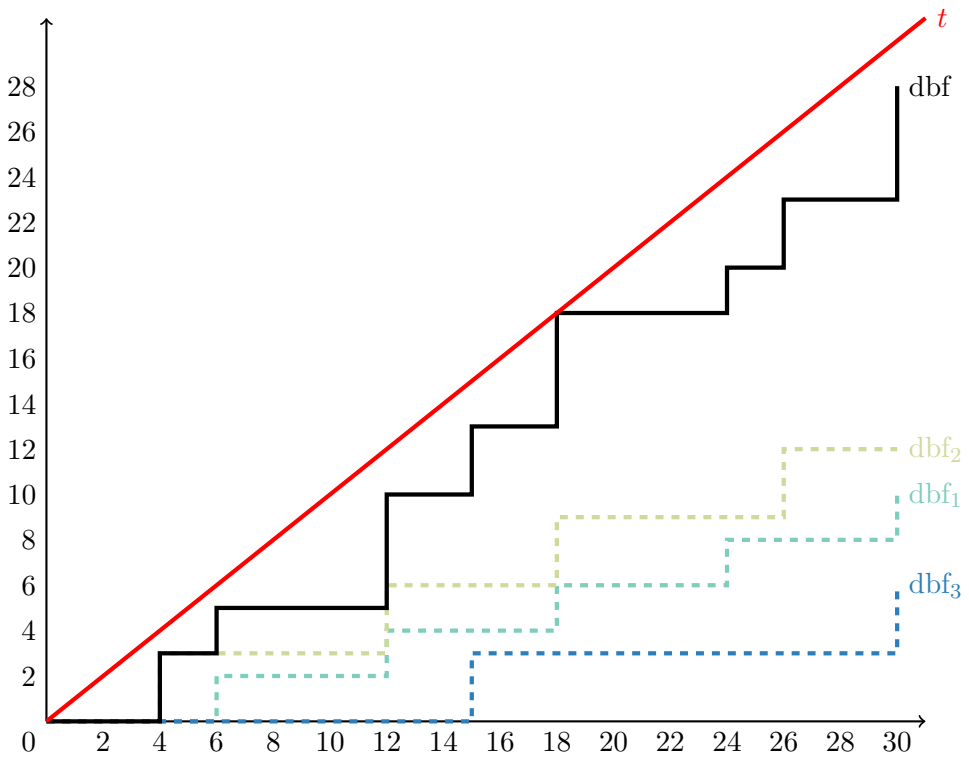
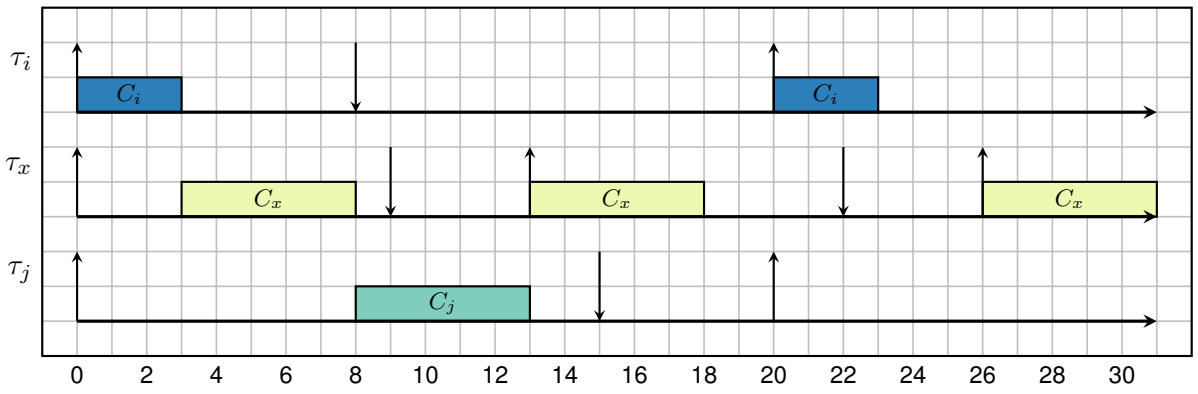
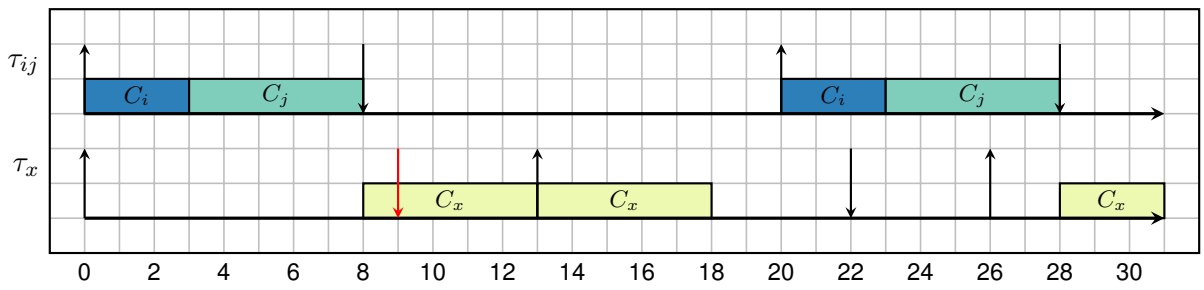


FIG. 5.3 : Représentation graphique des dbf de chaque tâche de l'ensemble du tableau 5.1 et de leur cumul

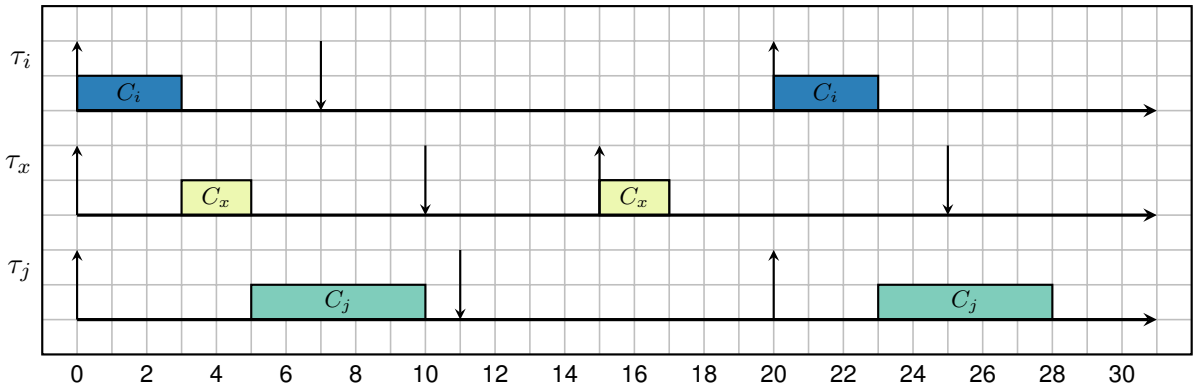


(a) Ensemble des tâches τ_i, τ_x et τ_j

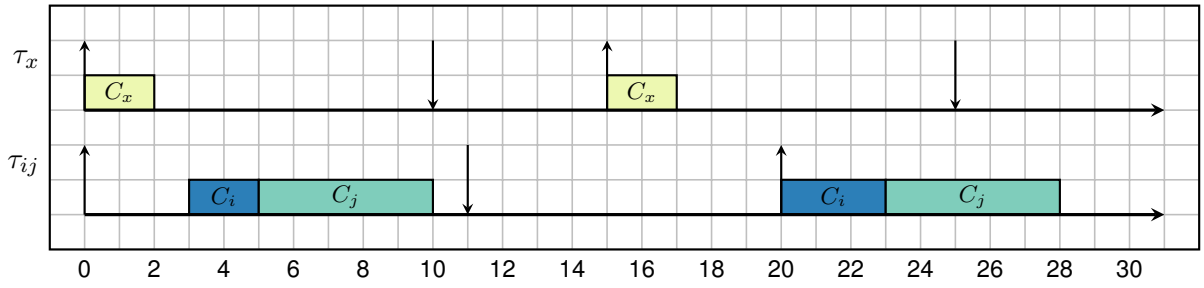


(b) Ensemble des tâches après regroupement de τ_i et τ_j

FIG. 5.4 : Impact du regroupement de tâches dans le cas (5.3b)



(a) Ensemble des tâches τ_i, τ_x et τ_j



(b) Ensemble des tâches après regroupement de τ_i et τ_j

Fig. 5.5 : Regroupement de tâches à coût nul (cas (5.3a))

5.5 Regroupement à coût nul

Dans cette section, nous prouvons que dans le cas (5.3a), le regroupement préserve l'ordonnançabilité de l'ensemble de tâches. Ainsi, il n'est pas nécessaire d'appliquer de nouveau un test d'ordonnançabilité, parfois coûteux. Nous nommons ce type de regroupement, *regroupement à coût nul*.

Théorème 3
Dans le cas (5.3a), $\mathcal{S}$ *ordonnançable sous* $\Phi \Rightarrow \mathcal{S}'$ *ordonnançable suivant l'algorithme d'ordonnancement qui a produit* Φ.

Preuve *Regrouper* τ_i *et* τ_j *sous l'échéance de* τ_j *revient à agrandir l'échéance de* τ_i *pour être égale à* D_j. *D'après la viabilité [8] (sustainability en anglais) des tests RTA et PDA, nous savons qu'agrandir une échéance préserve l'ordonnançabilité.* τ_i *et* τ_j *ont alors la même échéance et la même période, tous leurs travaux ont les mêmes priorités. Le système est donc ordonnançable quel que soit l'ordre d'exécution de* τ_i *et* τ_j *y compris lorsque l'exécution de* τ_i *précède systématiquement celle de* τ_j. *Or, ce cas correspond aux conditions du regroupement du cas* (5.3a). *En conséquence, si* $\mathcal{S}$ *est ordonnançable alors* $\mathcal{S}'$ *l'est aussi.*

Pour exemple, nous observons dans la figure 5.5 qu'avant regroupement la condition $D_j - C_j \leq D_i$ est respectée puisque $11 - 5 \leq 7$. Après regroupement, les tâches τ_x et τ_{ij} respectent leurs échéances respectives. En particulier, la part de τ_i dans τ_{ij} respecte toujours son échéance d'origine $D_i = 7$.

5.6 Coût du regroupement

Dans ce chapitre, nous avons montré que lorsque l'échéance la plus grande est choisie et que la contrainte (5.3a) est vérifiée, il est possible de regrouper les tâches sans diminuer aucune échéance, tout en garantissant le respect de toutes les échéances initiales des tâches qui composent le regroupement. Il est également prouvé que dans ce cas, il n'est pas nécessaire de réaliser une nouvelle analyse d'ordonnançabilité. Lorsqu'il n'est pas possible d'opérer de regroupement à coût nul, c'est sur l'échéance minimale (cas (5.3b)) que les tâches sont regroupées. Nous assurons certes que toutes les échéances initiales sont respectées mais en pratique, nous diminuons l'échéance de la tâche ayant l'échéance initiale la plus grande. En diminuant des échéances, nous réduisons les marges qui existent entre les dates de fin d'exécution des tâches et leurs échéances. D'une part, nous augmentons la « pression » sur la tâche elle-même et nous rendons le système plus dur à ordonnancer. D'autre part, nous réduisons nos chances de réaliser de futurs regroupements puisque la tâche en question aura moins de marge pour s'exécuter conjointement avec une autre tâche candidate au regroupement. Il apparaît donc que nous devons utiliser les regroupements à coût nul autant que faire se peut au détriment des regroupements sur les échéances les plus courtes. De surcroît, le regroupement à coût nul évite l'application coûteuse d'un nouveau test d'ordonnançabilité. Dans la suite, le regroupement de tâches sera guidé par l'objectif de maximisation des marges à chaque étape.

Chapitre 6

Minimisation du nombre de tâches

Après avoir défini comment nous regroupons des tâches, nous entrons dans le vif du sujet : trouver l'ensemble de tâches minimal en utilisant le regroupement de tâches. Dans ce chapitre, nous étudions la complexité du problème, une heuristique qui permet d'y apporter une solution et des résultats relatifs à la viabilité de la non-ordonnançabilité qui servent cette heuristique. Enfin, nous présentons une technique de résolution alternative à notre heuristique basée sur une formulation du problème en optimisation linéaire en nombres entiers. Le regroupement de tâches est un problème d'optimisation combinatoire et ce type de problème est classiquement traité par cette technique.

6.1 Complexité du problème de regroupement de tâches

6.1.1 Espace des solutions

Le problème consiste à trouver une partition de tâches ordonnançable avec un nombre minimal de sous-ensembles. Une partition d'un ensemble $\mathcal{X}$ est un ensemble de sous-ensembles non vides de $\mathcal{X}$ construit de manière à ce que chaque élément n de $\mathcal{X}$ est exactement dans un de ces sous-ensembles. Le nombre de partitions possible d'un ensemble correspond au nombre de Bell [88]. Le nombre de Bell est exponentiel sur la cardinalité de $\mathcal{X}$ et peut être calculé par la relation de récurrence suivante :

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k \text{ avec } B_0 = 1 \tag{6.1}$$

Comme nous ne regroupons que des tâches de même période, l'espace des solutions peut être restreint à $\prod_{i=0}^{m} B_{n_i}$ où B_n est le nombre de Bell d'un ensemble de n tâches avec périodes égales et m le nombre d'ensembles. Néanmoins, ce nombre reste exponentiel. Pour donner une intuition de la taille de l'espace de recherche, notons que $B_{500} \simeq 10^{844}$.

6.1.2 Le regroupement de tâches est un problème difficile

La taille de l'espace de recherche laisse présager une complexité du problème élevée. Cependant, exprimer une combinatoire élevée n'est pas suffisant pour juger de la dureté du problème dans les termes de la théorie de la complexité. En ce sens, nous soutenons dans cette section que le regroupement de tâches synchrones indépendantes est un problème difficile (le terme diffère ici de NP-difficile) du point de vue de la théorie de la complexité [51] et qu'il justifie l'utilisation de méthodes approchées.

Définissons premièrement le problème de regroupement de tâches plus formellement. Soit $\mathcal{S} = (\{\tau_i(C_i, D_i, T_i)\}_{1 \leq i \leq n})$ un ensemble de tâches synchrones. Soit $\mathcal{C}(\mathcal{S})$ qui dénote l'ensemble des ensembles de tâches ordonnançables obtenus par des regroupements de tâches tels que définis dans le chapitre 5 et appliqués récursivement sur $\mathcal{S}$. Le problème que nous adressons est le suivant :

Soit $\mathcal{S}$, trouver l'élément de $\mathcal{C}(\mathcal{S})$ de cardinalité minimale.

En préambule, nous rappelons un théorème sur la faisabilité d'un ensemble de tâches issu des travaux de Leung et Merril [70] puis de Baruah et al. [14] et Ekberg and Yi [44].

Théorème 4
Le problème de faisabilité d'un ensemble de tâches périodiques et indépendantes sur un processeur est co-NP-complet au sens fort.

Nous distinguons deux cas. Le premier correspond à la situation où le caractère ordonnançable de l'ensemble de départ $\mathcal{S}$ n'est pas connu. Le second fait l'hypothèse que l'ensemble de départ est ordonnançable.

Cas 1 : le caractère ordonnançable de $\mathcal{S}$ n'est pas connu

Ici, aucune hypothèse n'est faite quant à l'ordonnançabilité de $\mathcal{S}$. Considérons alors notre problème formulé dans sa version décisionnelle de la manière suivante :

Soit un ensemble $\mathcal{S}$ de n tâches, existe-t-il un ensemble $\mathcal{S}'$ de k tâches formé par des regroupements de tâches à partir de l'ensemble $\mathcal{S}$ faisable sur un processeur avec $k \leq n$?

Nous relevons trivialement que le problème de faisabilité d'un ensemble sur un processeur est un cas particulier du (et donc se réduit au) problème de regroupement de tâches lorsque $k = n$. Néanmoins, cette piste n'est pas pleinement satisfaisante puisqu'elle masque la complexité du problème de partitionnement, inhérente au problème de regroupement de tâches.

Cas 2 : $\mathcal{S}$ est ordonnançable

Dans ce second cas, nous partons du principe que l'ensemble de tâches initial $\mathcal{S}$ est ordonnançable. Il faut désormais montrer qu'il n'est pas possible de trouver l'élément minimal appartenant à $\mathcal{C}(\mathcal{S})$ obtenu par regroupement à partir de $\mathcal{S}$ sans avoir à aucun moment besoin de juger l'ordonnançabilité d'un ensemble. Le regroupement à coût nul défini en section 5.5 permet sous certaines conditions de réaliser un regroupement sans avoir à décider de l'ordonnançabilité de l'ensemble après coup. Il est en fait trivial de construire un ensemble de tâches appartenant $\mathcal{C}(\mathcal{S})$, minimal, tel que la condition de regroupement à coût nul ne soit pas applicable.

En conséquence, dans le cas général, résoudre le problème de regroupement de tâches nécessite d'être en mesure de décider de l'ordonnançabilité d'un ensemble de tâches, problème connu comme étant co-NP-complet au sens fort. Cette indication ne permet pas de définir précisément la complexité du problème, cependant elle indique que le problème est au moins aussi difficile qu'un problème co-NP-complet et justifie pleinement l'utilisation de méthodes approchées dans la suite du travail.

6.2 Non-ordonnançabilité viable

Dans cette section, nous prouvons qu'un ensemble qui est jugé non ordonnançable par les tests exacts utilisés le demeure après regroupement. Cela nous permet de réduire l'espace de recherche des solutions lorsque nous recherchons l'ensemble de tâches minimal.

Baruah et Burns [11] ont défini la notion d'*ordonnançabilité viable* (*sustainable schedulability* en anglais).

Définition 11 (Viabilité)
Un test d'ordonnançabilité pour une politique d'ordonnancement donnée est viable si n'importe quel ensemble de tâches jugé ordonnançable par le test d'ordonnançabilité reste ordonnançable lorsque les paramètres d'une ou de plusieurs tâches sont changés de une, de plusieurs ou de toutes les manières suivantes : (i) diminution des pires temps d'exécution ; (ii) agrandissement des périodes ; (iii) diminution de la gigue ; et (iv) agrandissement des échéances relatives.

De manière similaire, nous étudions comment évolue un ensemble de tâches non ordonnançable dans de « moins bonnes » conditions : avec diminution des échéances relatives, des périodes ou augmentation des pires temps d'exécution.

Nous examinons la non-ordonnançabilité viable dans le contexte de l'ordonnancement préemptif monoprocesseur avec des tâches synchrones sous assignation de priorités fixes par tâche et sous assignation de priorités fixes par travail. Notons que dans cette section, $\mathcal{S}'$ ne fait pas référence à l'ensemble $\mathcal{S}$ après regroupement.

6.2.1 Assignation de priorité fixe par tâche

Dans le cas d'un ordonnancement à priorité fixe par tâche, nous démontrons la viabilité de la non-ordonnançabilité du test exact RTA défini dans l'équation 5.5. La viabilité est considérée dans la littérature comme une caractéristique d'un test d'ordonnançabilité au même titre qu'un test est suffisant ou nécessaire. Nous appliquons également le principe de la viabilité de manière plus générale à toutes les assignations à priorité fixe par tâche en ce qui concerne les échéances.

Théorème 5 (Non-ordonnançabilité viable de RTA)
Un ensemble de tâches jugé non ordonnançable par RTA le demeure avec des échéances relatives plus courtes.

Preuve *Par contraposée. Soit $\mathcal{S}$ un ensemble de tâches jugé non ordonnançable par RTA. Soit $\mathcal{S}'$ un ensemble de tâches identique à $\mathcal{S}$ à l'exception que certaines tâches ont des échéances relatives plus courtes. Supposons que $\mathcal{S}'$ est ordonnançable. Comme RTA est viable, cela signifie que $\mathcal{S}$ est ordonnançable (parce que $\mathcal{S}$ est obtenu en augmentant certaines échéances de $\mathcal{S}'$). En conséquence, nous avons une contradiction.*

Lemme 1
Un ensemble de tâches qui est non ordonnançable avec RTA le demeure avec des périodes ou des temps d'exécution plus grands.

Preuve *Considérant RTA, nous pouvons observer trivialement dans l'équation* (5.5) *que diminuer une période T_i et augmenter un pire temps d'exécution C_i augmentera le pire temps de réponse R_i.*

Le théorème précédent se restreint à la viabilité de la non-ordonnançabilité selon un test d'ordonnancement particulier, RTA, pour la classe des ordonnancements à priorité fixe par tâche. Dans le théorème suivant, nous traitons une propriété plus forte qui concerne la classe des ordonnancements à priorité fixe par tâche.

Théorème 6 (Non-ordonnançabilité viable de l'ordonnancement FTP)
Un ensemble de tâches périodiques à échéances contraintes, non faisable en ordonnancement à priorité fixe par tâche, le demeure avec des échéances relatives plus courtes.

Preuve *En préalable, rappelons que DM étant un algorithme d'ordonnancement optimal, être ordonnançable sous une assignation Φ respectant l'ordre des priorités selon DM est équivalent à être faisable en assignation de priorité fixe par tâche.*

Soit un ensemble $\mathcal{S}$ non ordonnançable sous Φ, alors il existe au moins une tâche τ_k dont le temps de réponse est supérieur à son échéance :

$$\exists \tau_k \mid R_k > D_k \tag{6.2}$$

Supposons maintenant que nous réduisions l'échéance D_k de τ_k, alors deux cas se présentent.

Dans le premier cas, $D_{p+1} < D_{k'} < D_k$ où D_{p+1} est l'échéance de la tâche de priorité directement supérieure à celle de τ_k si une telle tâche existe et $D_{k'}$ la nouvelle échéance de τ_k dans $\mathcal{S}$ devenu $\mathcal{S}'$. Alors, l'équation (6.2) *tient toujours et S' reste non ordonnançable. Φ correspond toujours à une assignation de priorité selon DM. En conséquence, $\mathcal{S}'$ ne peut être faisable puisque DM est optimal.*

Dans le second cas, soit $D_{k''} < D_{p+1} < D_k$ où $D_{k''}$ est la nouvelle échéance de τ_k dans $\mathcal{S}$ devenu $\mathcal{S}''$. Alors l'équation (6.2) *tient toujours. $\mathcal{S}''$ reste non ordonnançable mais Φ ne correspond plus à une assignation de priorité respectant DM (c'est-à-dire que les priorités ne sont plus ordonnées selon les échéances décroissantes).*

$\mathcal{S}''$ serait-il toujours non ordonnançable sous une assignation Φ' qui correspondrait à l'ordre des priorités défini par DM ?

Par construction
Nous savons que $\mathcal{S}$ n'est pas ordonnançable sous Φ. DM étant optimal, $\mathcal{S}$ n'est pas non plus ordonnançable sous Φ' donc $\exists \tau_x \in \mathcal{S} \mid R_x > D_x$. Réduisons maintenant l'échéance de τ_k à $D_{k''}$ dans $\mathcal{S}$ devenu $\mathcal{S}'''$. L'équation $R_x > D_x$ reste vraie puisque le pire temps de réponse d'une tâche est insensible aux modifications des échéances. Conséquemment, S'' est non ordonnançable sous Φ' qui respecte DM. $\mathcal{S}''$ n'est donc pas faisable en assignation de priorité fixe par tâche. CQFD

6.2.2 Assignation de priorité fixe par travail

Dans cette section, nous utilisons le test exact basé sur la demande processeur pour un ensemble de tâches ordonnancées sous EDF pour prouver la *non-ordonnançabilité viable* pour l'assignation de priorité fixe par travail. L'analyse de la demande processeur PDA repose sur la dbf présentée en chapitre 5.

Théorème 7 (Non-ordonnançabilité viable sous PDA)
Un ensemble de tâches jugé non ordonnançable par la demande processeur le demeure avec des échéances relatives plus courtes, des périodes plus courtes et des temps d'exécution plus longs.

Preuve *Observer simplement dans l'équation* (5.10) *que la fonction* dbf *est croissante lorsque D_i diminue, T_i diminue et C_i augmente.*

6.3 Heuristique

Nous avons montré que notre problème est au moins aussi difficile qu'un problème co-NP-complet. De plus, nos expérimentations ont montré qu'un parcours exhaustif de l'espace des solutions n'était pas faisable. C'est pourquoi nous nous intéressons dans cette section à l'utilisation de méthodes approchées : les heuristiques.

6.3.1 Méthodes heuristiques

La complexité de certains problèmes est telle qui n'est parfois pas possible d'obtenir des solutions exactes en un temps raisonnable. Les méthodes heuristiques sont des algorithmes approchés qui apportent des solutions non optimales mais proches de celles qui sont exactes en un temps réduit.

Heuristiques de type *best-first*

Parmi ces algorithmes approchés, les heuristiques *best-first* (le *meilleur en premier* en anglais) avancent dans l'espace des solutions en ne choisissant que les solutions qui apparaissent localement les meilleures. Évidemment, choisir successivement des optimums locaux ne garantit pas d'atteindre nécessairement l'optimum global mais permet d'obtenir de bonnes performances. Les optimums locaux sont sélectionnés sur certains critères à travers une fonction d'évaluation ou fonction de coût. Par exemple, l'algorithme A* est un algorithme de recherche de chemin dans un graphe de type best-first qui estime localement le nœud se rapprochant le plus de la destination à atteindre. Lorsque le choix effectué localement est définitif, c'est-à-dire qu'il n'est pas possible de revenir en arrière dans l'espace de solution, il est question d'algorithmes *gloutons* (*greedy* en anglais).

Le choix de l'heuristique n'est pas un élément central dans ce travail. Notre objectif est de proposer et d'éprouver une méthode approchée efficace. Le principe élémentaire de notre méthode est l'utilisation d'une heuristique basée sur une fonction de coût. Comme expliqué ci-dessous, d'autres heuristiques reposent sur ce principe et pourraient être utilisées en lieu et place de l'algorithme glouton de type best-first que nous exposons.

6.3.2 Principe

Nous partons d'un ensemble initial de tâches où chaque tâche est considérée comme un regroupement à un élément puis nous tentons successivement de regrouper de plus en plus de regroupements ensemble de manière à minimiser la cardinalité totale. À chaque étape, nous tentons de regrouper un regroupement avec chacun des autres regroupements. Plusieurs candidats respectant les contraintes de validité d'un regroupement se présentent, il est donc nécessaire de choisir le candidat le plus prometteur. Une fonction de coût estime alors localement le candidat qui est le plus à même de conduire au meilleur regroupement. La figure 6.1 présente un exemple du principe de regroupement de l'heuristique. Nous partons d'un ensemble ordonnançable initial de 4 tâches A, B, C et D. Nous générons toutes les partitions possibles de 3 tâches de l'ensemble initial au niveau du dessous. Parmi celles-ci, l'ensemble $\{\{A\},\{BC\},\{D\}\}$ n'est pas ordonnançable. Grâce à la viabilité de la non-ordonnançabilité présentée en section 6.2, nous savons qu'il n'est pas nécessaire de parcourir les partitions générées à partir de cet ensemble puisqu'elles ne pourront pas aboutir à des ensembles ordonnançables. Toutes les autres partitions représentées de ce niveau de l'arbre représentent des ensembles ordonnançables. Elles constituent donc toutes des solutions valables de regroupement mais nous ne souhaitons en parcourir qu'une seule. Nous choisirons de parcourir la solution locale la plus prometteuse selon la fonction de coût sélectionnée. C'est ici la partition $\{\{A\},\{BC\},\{D\}\}$ qui est élue. Nous poursuivons la génération d'ensembles à deux tâches à partir de cette dernière.

Remarque 6.3.1 *Cette méthode de génération récursive génère de nombreux doublons. Nous observons par exemple que la partition $\{\{A\},\{BDC\}\}$ jugée non ordonnançable au niveau le plus bas de l'arbre est une partition fille possible de $\{\{A\},\{BC\},\{D\}\}$, déjà considérée non ordonnançable plus tôt. D'après la non-ordonnançabilité viable, il n'était pas nécessaire d'évaluer cette partition. Néanmoins, l'heuristique n'explore qu'une partition par niveau, il n'est donc pas possible de parcourir deux fois la même partition.*

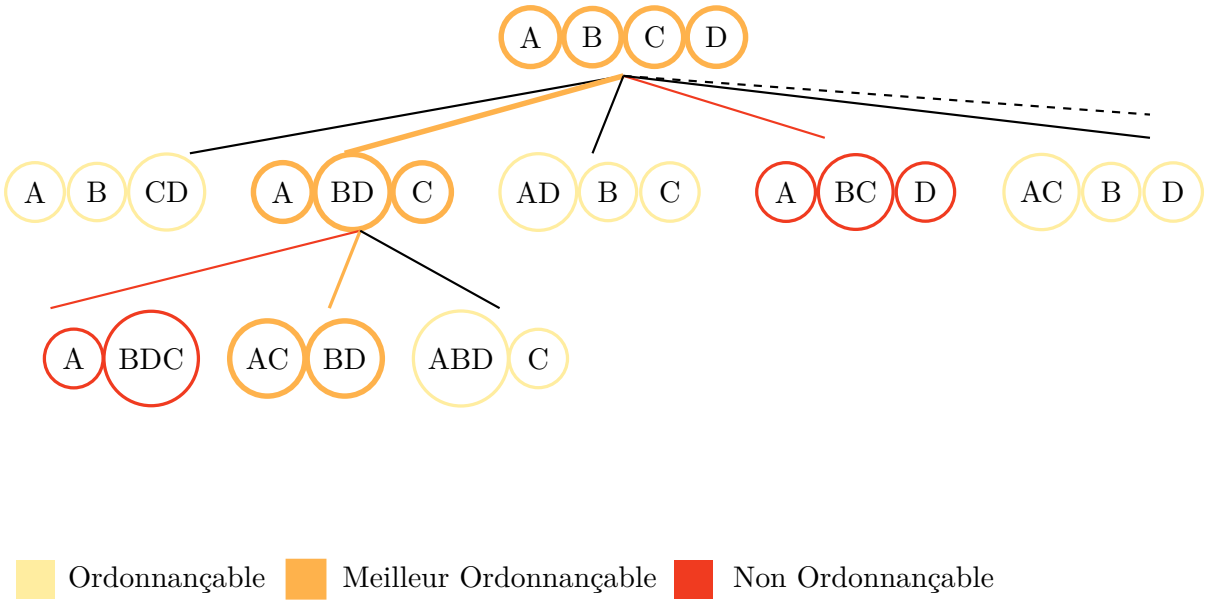


Fig. 6.1 : Illustration du principe de l'heuristique : regroupements successifs

Remarque 6.3.2 *Les résultats liés à la non-ordonnançabilité (cf. section 6.2) ne sont pas utilisés directement dans l'heuristique. En effet, dans l'algorithme présenté, une seule solution par étage de l'arbre est explorée et nous ne considérons que les candidats ordonnançables. Néanmoins, dans le cadre de méthode de recherche exacte de type séparation et évaluation [68] (branch-and-bound en anglais), ces résultats pourraient être utilisés comme des conditions d'élagages, en vue de diminuer l'espace des solutions.*

6.3.3 Fonction de coût

Motivation

À chaque étape de parcours de l'heuristique, plusieurs partitions filles peuvent s'avérer ordonnançables. Les méthodes classiques d'optimisation utilisent généralement des fonctions de coût puis sélectionnent localement la solution qui minimise ou maximise cette fonction de coût par rapport à l'objectif à atteindre. Ici, notre intérêt est de choisir la partition qui, localement, permettra de réaliser le plus grand nombre de regroupements.

Choix de la fonction de coût

Dans la littérature du domaine temps réel, la notion de fonction de coût correspond à des indicateurs utilisés pour mesurer la performance des algorithmes d'ordonnancement sur des ensembles de tâches. Il est parfois préférable de privilégier un algorithme diminuant les latences maximales, ou un autre minimisant en priorité le nombre de préemptions si les changements de contexte sont particulièrement coûteux dans le système visé. La fonction de coût va donc permettre de connaître quel algorithme d'ordonnancement est le plus à même d'optimiser le résultat en fonction du critère choisi. Cette définition de la fonction de coût diffère donc de celle qui concerne les méthodes heuristiques

Dans notre travail, l'idée est de baser la fonction de coût de l'heuristique sur les contraintes de temps des ensembles candidats au regroupement.

Lorsque nous réalisons certains regroupements, nous diminuons des échéances. Nous réduisons donc par la même les marges qui existent entre le pire temps d'exécution et les échéances. En ce sens, la densité (cf. équation (6.4)) de l'ensemble de tâches peut par exemple être un

critère à minimiser puisque maximiser la marge d'une tâche revient à minimiser la densité d'une tâche. Aussi, certains tests d'ordonnançabilité donnent des informations sur les caractéristiques temporelles d'une tâche dans l'ordonnancement produit telles que le pire temps de réponse. C'est pourquoi nous étudions dans la suite la possibilité d'utiliser les résultats d'analyse d'ordonnancement comme critère de sélection.

Utilisation des tests d'ordonnançabilité comme fonctions de coût

Dans le chapitre 5 ayant trait à l'ordonnançabilité, nous avons classifié les tests d'ordonnançabilité en fonction de leur caractère suffisant, nécessaire ou nécessaire et suffisant, c'est-à-dire exact.

Les tests exacts peuvent également être classés en deux catégories. D'une part, les tests exacts *booléens* permettent de savoir si oui ou non un ensemble est ordonnançable. D'autre part, les analyses de temps de réponse indiquent le pire temps de réponse de chaque tâche. Il suffit alors ensuite de vérifier que les temps de réponse sont bien inférieurs ou égaux aux échéances relatives des tâches pour considérer que l'ensemble est ordonnançable. Les seconds, généralement plus complexes, donnent une estimation plus précise du comportement de chaque tâche à l'exécution. On observe par exemple sur la figure 5.3 que la fonction cumulative dbf ne donne pas d'indice sur la où les tâche(s) susceptible(s) de rendre un ensemble non ordonnançable.

C'est la raison pour laquelle nous privilégions les tests basés sur l'analyse de temps de réponse dans le regroupement de tâches. Toutefois, nous verrons que des fonctions de coûts non basées sur les temps de réponses pourront être appliquées.

Cette catégorisation s'applique en général aux tests exacts, pour autant les tests suffisants cités peuvent être également considérés comme des bornes supérieures sur les pires temps de réponses. Par exemple, le test suffisant pour DM de l'équation (5.4) équivaut à

$$\forall i, (1 \leq i \leq n) : C_i + I_i \leq D_i \tag{6.3}$$

où I est une surestimation de l'interférence. En conséquence, $C_i + I_i$ est une borne supérieure sur le pire temps de réponse de la tâche τ_i.

Nous dressons ci-dessous une liste de fonctions de coût que nous avons appliquées. Les résultats des expérimentations sont consultables dans le chapitre 7.

- Densité de l'ensemble de tâches :

$$\min \sum_{i=1}^{n} \frac{C_i}{D_i} \tag{6.4}$$

  Comme expliqué plus tôt, nos regroupements ont pour effet de diminuer les échéances d'un ensemble. Plus les échéances sont réduites, plus il est délicat de réaliser des regroupements supplémentaires. Intuitivement, nous avons donc tout intérêt à sélectionner un candidat qui maximise la laxité globale ou de manière équivalente, qui minimise la somme des $\frac{C_i}{D_i}$

- Densité sur le pire temps de réponse :

$$\min \sum_{i=1}^{n} \frac{R_i}{D_i} \tag{6.5}$$

  Le pire temps d'exécution C_i est une estimation du plus long temps d'exécution de la tâche. En pratique, elle peut terminer son exécution plus tôt ou plus tard, si elle est retardée par l'exécution d'autres tâches plus prioritaires. Sa date de fin d'exécution effective dans le pire cas est caractérisée par la notion de pire temps de réponse R_i. Prendre ce critère est donc plus précis que de se baser sur le pire temps d'exécution mais il nécessite de connaître le comportement de la tâche à l'exécution, à l'aide d'une analyse de temps de réponse. Ce test est donc plus coûteux mais plus précis que la fonction de densité basée uniquement le pire temps d'exécution.

- Laxité sur l'hyper-période :

$$\max \sum_{i=1}^{n} (D_i - C_i) \cdot \frac{H}{T_i} \tag{6.6}$$

  L'hyper-période (ppcm des périodes) correspond à un intervalle de temps périodique à l'issue duquel le plan d'exécution des tâches se répète. Il est souvent utilisé en ordonnancement puisque, prouver un ensemble ordonnançable sur cet intervalle revient à prouver son ordonnançabilité de manière générale. La laxité d'une tâche aura un poids différent en fonction du nombre de répétitions de la tâche sur l'hyper-période. Pour cette fonction, nous proposons de pondérer la marge disponible pour le regroupement par tâche (sa laxité) par son nombre de répétitions sur l'hyper-période ($\frac{H}{T_i}$).

- Aléatoirement. Le critère aléatoire ne sélectionne pas nécessairement le plus mauvais candidat mais il est utilisé à titre de comparaison.

6.3.4 Algorithme

L'algorithme développé ici est de type *greedy best-first*. Comme décrit dans l'algorithme 1, nous énumérons récursivement les partitions. À chaque appel récursif, nous essayons prioritairement d'appliquer les regroupements à coût nul sur chacun des fils générés. Si la condition du regroupement à coût nul est remplie, nous réalisons un appel récursif sur le nouveau regroupement, sinon, nous accumulons un triplet contenant l'ensemble des tâches et les indices des deux tâches que nous souhaitons regrouper dans une liste.

Enfin, si aucun regroupement à coût nul n'a été effectué pendant cette étape de génération, nous choisissons l'ensemble de tâches ordonnançable le plus prometteur de la liste des ensembles non concernés par le regroupement à coût nul. Celui-ci est sélectionné par une fonction de coût (cf. section 6.3.3). Dans l'algorithme 1, il s'agit de la fonction `meilleurCoût`. Notons qu'il est possible arrêter l'algorithme si un nombre requis de tâches est atteint plutôt que de chercher l'ensemble de tâches de cardinalité minimale. En pratique, il s'agit juste d'introduire un compteur des appels récursifs et d'arrêter la récursion aussi tôt que le nombre de tâches requis est atteint. Le nombre maximal de threads autorisé dans les systèmes d'exploitation temps réel est souvent limité. Dans cette configuration, la modification mineure proposée prend tout son intérêt.

La condition du regroupement à coût se base en partie sur l'échéance de la tâche qui a la plus grande échéance, ou sur son pire temps de réponse. Si le test d'ordonnançabilité utilisé dans l'algorithme est une analyse de temps de réponse, alors le temps de réponse peut être utilisé comme une valeur moins pessimiste que l'échéance. Il apparaît alors que si le test utilisé n'est pas une analyse de temps de réponse ou son approximation, aucun regroupement à coût nul ne sera réalisé sur la condition moins pessimiste puisque l'information n'est pas disponible. C'est la raison pour laquelle nous conseillons d'effectuer une analyse de temps de réponse sur le fils le plus prometteur, afin d'obtenir le temps de réponse et de pouvoir effectuer des regroupements à coût nul sur la condition plus favorable.

Lemme 2
n étant le nombre de tâches initial, la complexité de l'algorithme 1 avec des tests linéaires est $\mathcal{O}(n^4)$, pseudo-polynomiale avec des tests pseudo-polynomiaux (RTA pour DM) et exponentielle avec des tests exponentiels (RTA pour EDF).

Preuve *Le nombre de fils générés à partir d'une partition de i élément(s) est égal à $(i \times (i-1)/2)$. Nous explorons uniquement un fils parmi tous les fils visités à chaque étape du parcours à l'aide de notre heuristique gloutonne. En conséquence, le nombre maximal de partitions visitées à chaque étape est égal à $\sum_{i=0}^{n} \frac{i \times (i-1)}{2}$. Cette somme correspond à la somme des n premiers nombres triangulaires (aussi appelés nombres tétraédriques) et son expression analytique est de la forme*

Algorithme 1 Algorithme de l'heuristique de regroupement
Fonction regroupement($\mathcal{S}$)

Entrée : $\mathcal{S} = (\{\tau_i\}_{1 \leq i \leq n})$: ensemble initial de tâches triées par ordre croissant des échéances

fils $\leftarrow \varnothing$
/*Tentative de regroupement à coût nul.*/
Pour $j = n - 1$ à 0 **Faire**
  Pour $i = j - 1$ à 0 **Faire**
    Si $T_i == T_j$ **Alors**
      Si $(C_i + C_j \leq D_j) \wedge ((D_j - C_j \leq D_i) \vee (R_j - C_j \leq D_i))$ **Alors**
        $\mathcal{S}' \leftarrow \mathcal{S} \setminus \{\tau_i, \tau_j\} \cup \{\tau_{ij}\}$
        Retourne regroupement($\mathcal{S}'$)
      Sinon
        fils $\leftarrow$ fils $\cup \{(\mathcal{S}, i, j)\}$
      Fin Si
    Fin Si
  Fin Pour
Fin Pour

filsOrdo $\leftarrow \varnothing$
/*Si aucun regroupement à coût nul n'est possible, recherche du fils le plus prometteur.*/

Pour Tout $(M, x, y) \in$ fils **Faire**
  Si $C_x + C_y \leq D_x$ **Alors** /*Laxité*/
    $M' \leftarrow M \setminus \{\tau_x, \tau_y\} \cup \{\tau_{xy}\}$
    Si ordonnançable(M') **Alors**
      filsOrdo $\leftarrow$ filsOrdo $\cup M'$
    Fin Si
  Fin Si
Fin Pour
Si filsOrdo $\neq \varnothing$ **Alors**
  Retourne regroupement(meilleurCoût(filsOrdo)) /*Poursuite avec le fils le plus prometteur*/
Sinon
  Retourne $\mathcal{S}$
Fin Si

$f(n) = \frac{n(n+1)(n+2)}{6}$ [98]. En conséquence, la complexité de cette séquence est $\mathcal{O}(n^3)$. Nous appliquons un test d'ordonnançabilité suffisant en $\mathcal{O}(n)$ (qu'il s'agisse de DM ou de EDF) sur chaque partition visitée, donc la complexité de l'heuristique est de $\mathcal{O}(n^3) \times \mathcal{O}(n) = \mathcal{O}(n^4)$. Similairement, appliquer des tests d'ordonnançabilité de complexité pseudo-polynomiale ou exponentielle entraîne une complexité également polynomiale ou exponentielle pour l'ensemble de l'algorithme.

6.4 Formulation en un problème d'optimisation linéaire en nombres entiers

Dans cette section, nous présentons une formulation du problème de regroupement de tâches en optimisation linéaire en nombres entiers. Ces techniques permettent de résoudre des problèmes combinatoires de manière exacte. Nous verrons que cette solution ne passe pas à l'échelle, raison pour laquelle elle n'a pas été retenue. Néanmoins, sa présentation nous a semblé digne d'intérêt puisqu'elle pourrait servir de base à la résolution de problèmes proches, mais de taille plus réduite.

6.4.1 Optimisation linéaire en nombres entiers

L'optimisation linéaire (ou programmation linéaire) est un problème d'optimisation mathématique dans lequel nous cherchons à minimiser ou maximiser une fonction objectif sur un polyèdre convexe. Le problème est spécifié au moyen de contraintes linéaires sur des variables et des constantes. Lorsque les variables sont contraintes à ne prendre que des valeurs entières, il est question d'optimisation linéaire en nombres entiers (OLNE).

La forme standard d'un problème OLNE est :

$$\begin{aligned} \text{minimiser/maximiser} \quad & c^T x \\ \text{tel que} \quad & Ax + s = b \\ & s \geq 0 \\ & x \in \mathbb{Z} \end{aligned}$$

où b et c sont des vecteurs et A une matrice de valeurs entières.

6.4.2 Formulation du regroupement de tâches en OLNE

La recherche exhaustive de solution au problème de regroupement de tâches n'a pas donné de résultats satisfaisants. La plupart des problèmes d'optimisation combinatoire de complexité importante (les problèmes NP-difficiles par exemple) sont traités par des heuristiques. Néanmoins, les techniques basées sur l'optimisation linéaire en nombres entiers permettent d'obtenir des résultats exacts pour certains problèmes de taille moyenne. La technique consiste d'abord à formuler le problème sous forme de variables et de contraintes linéaires puis à confier cette modélisation à un programme qui applique des algorithmes adaptés à ce type de résolution. Nous séparons la formulation en deux parties. La première est dédiée aux variables et aux contraintes générales au problème de regroupement indépendamment du test d'ordonnançabilité choisi. Dans la seconde partie, nous formulons les tests RTA et PDA (basé sur la dbf).

Variables et contraintes générales au regroupement

Nous définissons premièrement la variable binaire y_i qui correspond à l'utilisation d'un regroupement de tâches. Nous cherchons à minimiser le nombre de tâches obtenues par regroupement,

la fonction objectif est la suivante :

$$\min \sum_{i=1}^{n} y_i \tag{6.7}$$

La variable binaire x_{ij} indique que la tâche j est assignée au regroupement i.

$$x_{ij} = \begin{cases} 1 & \text{si } j \text{ est dans le regroupement } i \\ 0 & \text{sinon.} \end{cases} \tag{6.8}$$

La contrainte suivante impose que le regroupement i soit utilisé si la tâche j lui est assignée.

$$x_{ij} \leq y_i \quad \forall i, j \in [1, n] \tag{6.9}$$

Aussi, une tâche ne peut être assignée qu'à un et un seul regroupement. Cette condition est encodée dans la contrainte :

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \in [1, n] \tag{6.10}$$

Le temps d'exécution w_i d'un regroupement est égal à la somme des temps d'exécution des tâches qui lui sont assignées.

$$w_i = \sum_{j=1}^{n} (C_i x_{ij}) \quad \forall i \in [1, n] \tag{6.11}$$

Seules les tâches de même période peuvent être regroupées ensemble. Le regroupement a donc la même période que les tâches qu'il contient. Nous introduisons donc deux variables $TMIN_i$ et $TMAX_i$ qui représentent la période d'un regroupement. Ces variables sont associées aux contraintes suivantes :

$$M(1 - x_{ij}) + TMAX_i \geq T_j \qquad \forall i, j \in [1, n] \tag{6.12}$$

$$TMIN_i \leq T_j + M(1 - x_{ij}) \qquad \forall i, j \in [1, n] \tag{6.13}$$

$$TMIN_i = TMAX_i \qquad \forall i \in [1, n] \tag{6.14}$$

où M est une valeur suffisamment grande pour que la contrainte soit trivialement vérifiée si x_{ij} vaut 0 et qu'elle contraigne la période du regroupement sinon. Cette technique, appelée « méthode du grand M » est fréquemment utilisée en OLNE pour exprimer des conditions.

L'échéance δ_i d'un regroupement de tâches est égale à l'échéance minimale des tâches qui s'y trouvent. Nous n'encodons pas la contrainte de regroupement à coût nul pour simplifier la formulation du problème. Nous faisons de nouveau appel à la méthode du grand M.

$$\delta_i \leq D_j + M(1 - x_{ij}) \quad \forall i, j \in [1, n] \tag{6.15}$$

Variables et contraintes propres à l'ordonnançabilité

L'ensemble des tâches regroupées doit être ordonnançable. En conséquence, nous utilisons une formulation du test RTA pour des ensembles de tâches à priorité fixe par tâche ou une formulation du test PDA basé sur la *demand bound function* pour les ensembles à priorité fixe par travail. Des formulations similaires ont été appliquées dans [108, 21].

Formulation du test RTA

Pour rappel, le test RTA considère un ensemble de tâches ordonnançable si et seulement si :

$$R_i = C_i + \sum_{j \in hp(i)} I_{ji} \cdot C_j \leq D_j \quad \forall i \in [1, n]$$

où la variable I_{ji} correspond au nombre d'interférences de la tâche (regroupement dans notre cas) i sur la tâche (regroupement dans notre cas) j et vaut

$$I_{ji} = \left\lceil \frac{R_i}{TMAX_j} \right\rceil \quad \forall i, j \in [1, n] \tag{6.16}$$

L'opérateur de partie entière par excès peut être défini en OLNE par la contrainte suivante :

$$\frac{R_i}{TMAX_j} \leq I_{ji} < \frac{R_i}{TMAX_j} + 1 \quad \forall i, j \in [1, n] \tag{6.17}$$

Nous constatons que la contrainte de l'équation (6.17) n'est pas linéaire. Bien que certains solveurs (la notion de solveur est détaillée dans la section suivante) de contraintes tels que CPLEX [34] soient capables de résoudre des équations quadratiques, nous proposons ici une formulation linéaire. Pour ce faire, nous utilisons une formulation grand M pour prendre en compte la période de la tâche j si elle est présente dans le regroupement i. Ceci est possible car toutes les périodes des tâches d'un même regroupement sont égales et définissent la période du regroupement. La variable T_{kj} correspond à la période de la tâche k dans le regroupement j.

$$\frac{R_i}{T_{kj}} \leq I_{ji} + M\,(1 - x_{kj}) \qquad \forall i, j, k \in [1, n] \tag{6.18}$$

$$I_{ji} < \frac{R_i}{T_{kj}} + 1 + M\,(1 - x_{kj}) \qquad \forall i, j, k \in [1, n] \tag{6.19}$$

Le test RTA est valable quelle que soit l'assignation de priorités choisie. Dans notre problématique, nous ne nous intéressons qu'à l'assignation selon DM, qui est optimale pour les assignations de priorités fixes par tâche. L'algorithme d'ordonnancement DM assigne les priorités en se basant sur l'échéance des tâches. Plus l'échéance relative d'une tâche est courte, plus sa priorité est élevée. Dans notre cas, nous ne connaissons pas statiquement les priorités des regroupements de tâches puisqu'elles vont dépendre du partitionnement des tâches. Cependant, nous pouvons ajouter une contrainte telle que les regroupements de tâches soit triés par ordre décroissant des échéances, selon DM.

$$\delta_i \leq \delta_{i+1} \quad \forall i \in [1, n-1] \tag{6.20}$$

Ainsi, nous connaissons par avance les regroupements susceptibles d'interférer les uns avec les autres. RTA peut alors être formulé de la sorte :

$$R_i = w_i \sum_{j>i}^{n} I_{ji} w_j \leq \delta_j \quad \forall i \in [1, n] \tag{6.21}$$

La contrainte de l'équation (6.21) n'est pas linéaire. Nous contournons cette limite en introduisant une variable Ψ_{ijk} qui correspond à l'interférence du regroupement i sur le regroupement k si la tâche j y participe.

$$\Psi_{ijk} = \begin{cases} I_{ik} & \text{si } x_{ij} = 1 \\ 0 & \text{sinon.} \end{cases} \tag{6.22}$$

Les valeurs possibles de Ψ_{ijk} sont définies par rapport à l'interférence en utilisant la méthode du grand M grâce aux contraintes suivantes :

$$I_{ik} - M(1 - x_{ij}) \leq \Psi_{ijk} \leq I_{ik} \qquad \forall j, k, i > k \in [1, n] \tag{6.23}$$

$$0 \leq \Psi_{ijk} \leq M x_{ij} \qquad \forall j, k, i > k \in [1, n] \tag{6.24}$$

Le test RTA peut alors être limité à la contrainte suivante :

$$r_k = w_k + \sum_{j=1}^{n} C_j \Psi_{ijk} \quad \forall k, i > k \in [1, n] \tag{6.25}$$

Si contrairement à notre cas, les assignations de priorités fixes aux regroupements sont des variables du problème, il est possible d'utiliser la formulation tirée de [108].

Formulation du test PDA
Nous rappelons que l'analyse de demande processeur se base sur la *demand bound function*. Il est nécessaire que :

$$\forall t \in \mathcal{S}, \sum_{i=1}^{n} \mathrm{dbf}\,(i,t) \leq t \tag{6.26}$$

où $\mathcal{S}$ correspond à l'ensemble des dates où vérifier la dbf, c'est-à-dire, à toutes les échéances absolues inférieures à l'hyper-période :

$$\mathcal{S} = \bigcup_{i=1}^{n} \Big\{ t | t = D_i + T_i \cdot k \leq H \Big\} \tag{6.27}$$

$\mathcal{S}$ peut être statiquement calculé sur l'ensemble de départ puisque le regroupement ne modifie pas l'hyper-période et ne peut que supprimer certaines échéances absolues auxquelles calculer la dbf. En effet, nous ne modifions pas les périodes des tâches et nous regroupons soit sur l'échéance maximale, soit sur l'échéance minimale donc nous n'ajoutons pas de nouvelle échéance. Le seul risque est ici de calculer inutilement la dbf à certaines dates où ses valeurs sont redondantes.

Nous définissons la variable $FLOOR_{it}$ qui correspond à la partie entière par défaut de l'équation de la dbf du regroupement i à la date t :

$$\frac{t + TMAX_i - \delta_i}{TMAX_i} - 1 < FLOOR_{it} \leq \frac{t + TMAX_i - \delta_i}{TMAX_i} \tag{6.28}$$

La contrainte (6.28) est quadratique. En conséquence, nous utilisons la même technique que dans l'équation (6.21) en introduisant une variable Ω_{ijt} qui correspond à la valeur de la dbf du regroupement i à la date t si la tâche j y participe.

$$\Omega_{ijt} = \begin{cases} FLOOR_{it} & \text{si } x_{ij} = 1 \\ 0 & \text{sinon.} \end{cases} \tag{6.29}$$

Les contraintes suivantes fixent la valeur de Ω_{ijt} :

$$FLOOR_{it} - M(1 - x_{ij}) \leq \Omega_{ijt} \leq FLOOR_{it} \qquad \forall t \in \mathcal{S}, \forall i, j \in [1, n] \tag{6.30}$$

$$0 \leq \Omega_{ijt} \leq M x_{ij} \qquad \forall t \in \mathcal{S}, \forall i, j \in [1, n] \tag{6.31}$$

Nous introduisons la variable DBF_{it} qui correspond à la valeur de la dbf du regroupement i à la date t. Celle-ci ne doit en aucun cas dépasser le temps disponible de longueur t :

$$DBF_{it} = C_j\, \Omega_{jt} \qquad \forall i, j \in [1, n]\,, \forall t \in \mathcal{S} \tag{6.32}$$

$$\sum_{i=1}^{n} DBF_{it} \leq t \qquad \forall t \in \mathcal{S} \tag{6.33}$$

6.4.3 Résolution

Une fois le problème correctement formulé, la résolution peut être confiée à des solveurs. Les solveurs sont des programmes qui appliquent des méthodes telles que l'*algorithme du simplexe* ou le *branch-and-bound* de manière à trouver des solutions optimales au problème d'optimisation linéaire formulé. Nos essais ont été réalisés avec les solveurs *lpsolve* [16] et CPLEX [34]. CPLEX a la réputation d'être plus efficace que lpsolve, ce qu'ont confirmé nos essais bien que les résultats n'aient tout de même pas été satisfaisants. Avec un test RTA, à partir d'une vingtaine de tâches, il n'est pas possible d'obtenir de solution en un temps raisonnable (après plusieurs heures de calcul). La limite se place même en deçà pour PDA. Le test repose sur l'évaluation de la dbf à un ensemble de dates sur un intervalle borné supérieurement par l'hyper-période. Le nombre de contraintes et de variables générées est donc de l'ordre de l'hyper-période et augmente considérablement avec le nombre de tâches. Or les ensembles que nous visons disposent de plusieurs centaines de tâches ce qui nous encourage à privilégier l'utilisation de l'heuristique. Ces résultats préliminaires expliquent aussi pourquoi nous n'avons pas poussé plus loin les expérimentations de comparaison.

Bini et Baruah [21] proposent également des formulations OLNE approchées de tests exacts pour le problème de partitionnement de tâches sur différentes processeurs. Nous évinçons cette alternative puisque les résultats de notre heuristique affichent des résultats satisfaisants en se basant sur des tests exacts (cf. chapitre 7). Il n'y a donc pas lieu de chercher une solution optimale qui se baserait sur un test qui est approché.

Je tiens à remercier Clément Ballabriga pour l'aide qu'il m'a apportée dans la formulation du problème en OLNE. Dans le chapitre suivant, nous exposons les résultats des expérimentations menées à partir de l'heuristique présentée plus tôt dans ce chapitre.

Chapitre 7

Expérimentations

Tout d'abord, nous présentons les méthodes de génération d'ensemble de tâches utilisées pour nos expérimentations sur les tâches indépendantes en mono processeur. Ensuite, à partir de multiples expérimentations, nous mettons en exergue les paramètres qui influent sur les performances du regroupement de tâches. Les essais ont été réalisés sur une machine dotée de 4 processeurs Intel Xeon cadencés à 2.4 GHz et embarquant 32Go de mémoire vive.

7.1 Génération d'ensembles de tâches

7.1.1 Génération des taux d'utilisation

Algorithmes UUnifast

L'algorithme UUnifast [22] est un algorithme mis au point pour la génération de taux d'utilisation sur monoprocesseur. Il génère une distribution uniforme de n taux d'utilisation non biaisés à partir du nombre de tâches n de l'ensemble et du taux d'utilisation processeur total souhaité u. UUnifast est un algorithme efficace de complexité $\mathcal{O}(n)$. Nous rappelons qu'un ensemble au taux d'utilisation supérieur à 1 est trivialement non ordonnançable puisque l'utilisation processeur dépasse alors le temps maximal disponible.

7.1.2 Génération des périodes

Lors de la génération de tâches, le choix des périodes est un élément sensible pour les tests d'ordonnançabilité. En effet, certains de ces tests basent leur analyse sur un intervalle de faisabilité. La longueur de cet intervalle dépend du plus petit commun multiple des périodes (ppcm) appelée l'hyper-période. Si les périodes sont grandes, premières entre elles, nombreuses, l'hyper-période explose. Le défi consiste donc à générer des périodes aléatoirement tout en limitant la taille de l'hyper-période et c'est l'objet de la méthode de Goossens et Macq [52] implantée dans notre prototype. Certains travaux choisissent des périodes harmoniques, c'est-à-dire des périodes multiples entre elles pour limiter l'hyper-période. Précisons qu'en pratique, les différentes périodes sont dictées par la dynamique du système et en ce sens, elles ne sont pas nécessairement très nombreuses. Notons également que le nombre de tâches que nous considérons (plusieurs centaines) nous oblige à choisir des périodes suffisamment grandes pour diminuer le risque d'obtenir des tâches au pire temps d'exécution nul. En effet, plus le nombre de tâches est grand, plus la valeur moyenne du taux d'utilisation par tâche a de chance d'être réduite. Pour ce faire, nous multiplions les périodes générées par un facteur entier obtenu empiriquement. De facto, cette contrainte a tendance à augmenter l'hyper-période des ensembles générés.

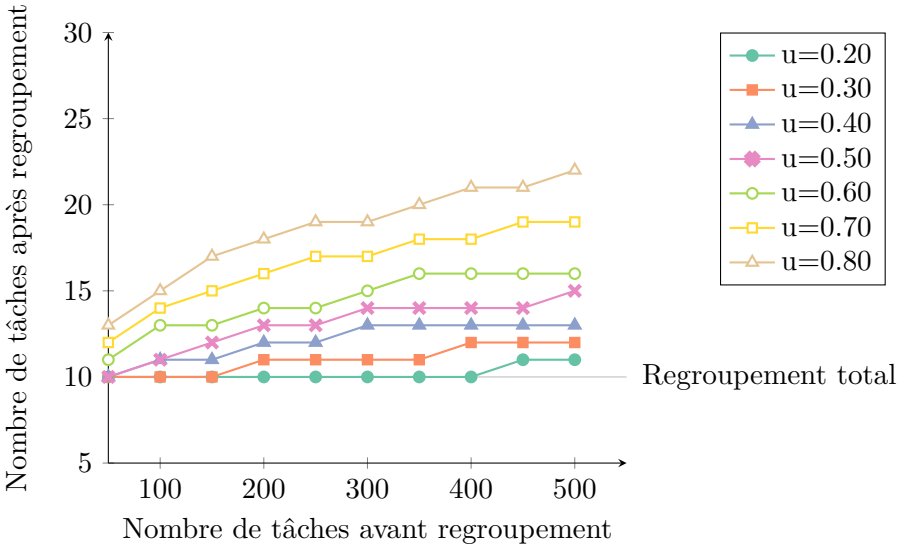


Fig. 7.1 : Impact du regroupement sur le nombre de tâches en DM ($d_{min} = 0$, $d_{max} = 1$)

7.1.3 Génération des échéances

Nous générons les échéances similairement à Goossens et Macq dans [52]. Nous déterminons aléatoirement l'échéance dans un intervalle $[d_{min}, d_{max}]$ tel que $C_i \leq d_{min} \leq D_i \leq d_{max} \leq T_i$. Notons qu'un système avec une échéance inférieure au pire temps d'exécution est trivialement non ordonnançable. En résumé, $D_i = arrondi((T_i - C_i) \times aleatoire(d_{min}, d_{max})) + C_i$ avec $0 \leq d_{min} \leq d_{max}$ où la fonction $aleatoire(d_{min}, d_{max})$ retourne un nombre réel pseudo-aléatoire uniformément distribué sur l'intervalle $[d_{min}, d_{max}]$ et où la fonction $arrondi(x)$ retourne l'entier le plus proche de x. Rappelons que $D_i = T_i$ correspond à une échéance implicite et $D_i \leq T_i$ à une échéance contrainte.

7.2 Évolution du nombre de tâches

Lorsque les échéances sont aléatoires, nous pouvons nous attendre à ce que le nombre de tâches augmente avec la hausse du taux d'utilisation puisque sa hausse rend l'ordonnançabilité plus délicate. Cette hypothèse est confirmée par la figure 7.1 pour DM et la figure 7.2 pour EDF. Nous avons mené les expérimentations sur des échantillons de 1000 ensembles initialement ordonnançables par nombre de tâches initial et par taux d'utilisation. Les deux figures affichent les résultats du regroupement de tâches en fonction du nombre de tâches initial et du taux d'utilisation. La fonction de coût utilisée est basée sur l'analyse de temps de réponse. Ce choix impose d'utiliser un test d'ordonnançabilité basé sur l'analyse de temps de réponse. Ce test étant particulièrement complexe pour EDF, nous avons limité le nombre de tâches maximal à 300 pour la figure 7.2. Nous observons que, quel que soit le taux d'utilisation, la technique de regroupement permet de diviser le nombre de tâches par un facteur de 10 au moins. Aussi, nous remarquons que la hausse du taux d'utilisation rend le regroupement moins efficace. Finalement, il est clair que le regroupement est moins efficace sous EDF que DM.

7.3 Comparaisons des fonctions de coût

La fonction de coût permet de sélectionner le meilleur candidat local au regroupement dans l'heuristique. Cette section a pour objectif de comparer l'efficacité du regroupement de tâches par rapport à la fonction de coût utilisée, parmi celles présentées en section 6.3.3. A priori,

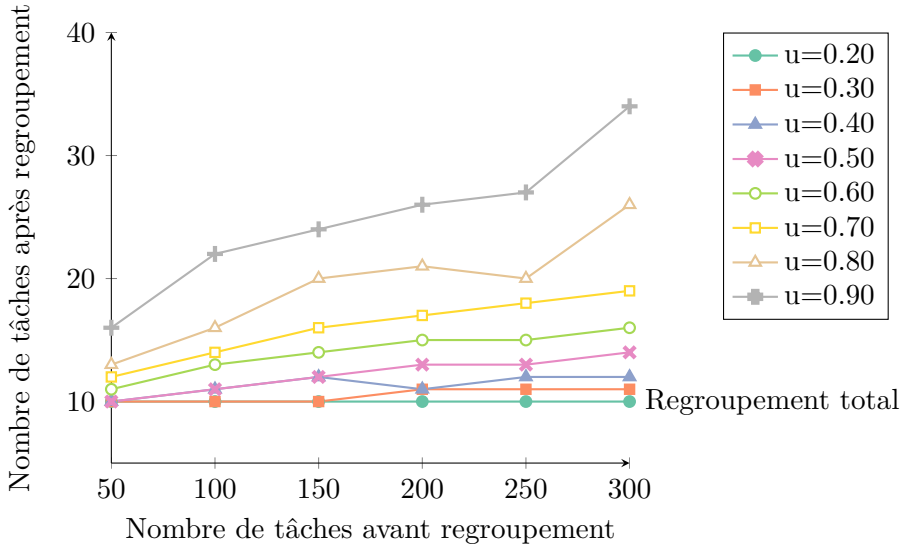


FIG. 7.2 : Impact du regroupement sur le nombre de tâches en EDF ($d_{min} = 0$, $d_{max} = 1$)

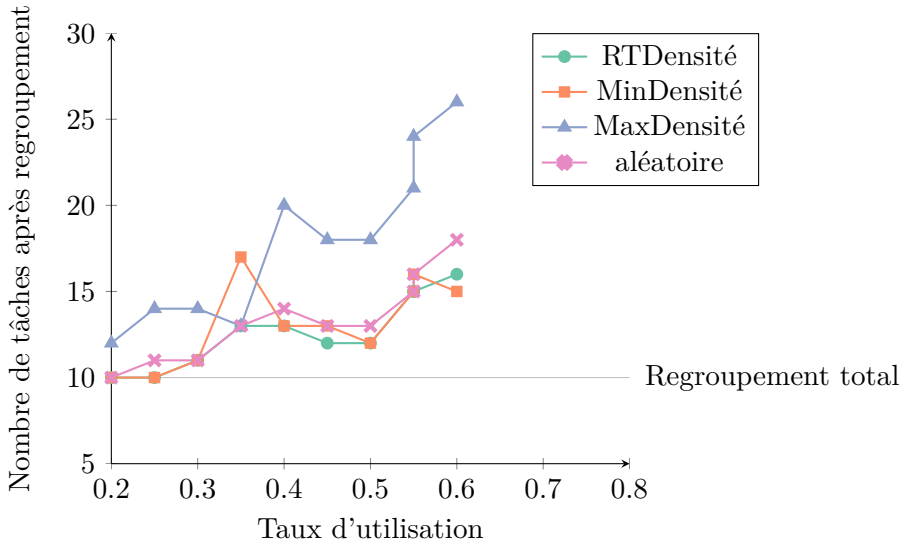


FIG. 7.3 : Comparaison des fonctions de coût en DM

la fonction de coût de l'équation 6.5 basée sur l'analyse de temps de réponse peut paraître la fonction la plus précise.

7.3.1 En fonction du taux d'utilisation

Les figures 7.3 et 7.4 comparent l'impact du choix de la fonction de coût sur le regroupement de tâches par rapport au taux d'utilisation en DM puis en EDF. La fonction de coût `MaxDensité` correspond au choix de l'ensemble qui maximise la densité. En théorie, elle correspond au pire choix possible en se basant sur la densité, elle est utilisée à des fins de comparaison. Bien que le regroupement paraisse globalement meilleur lorsque les taux d'utilisation sont bas, il est difficile de tirer une tendance claire de ces figures. Le taux d'utilisation ne semble pas être le critère de comparaison le plus adéquat, c'est pourquoi nous comparons dans la suite les performances du regroupement vis-à-vis des fonctions de coût choisies par rapport à la densité de l'ensemble initial. Puisque regrouper des tâches peut conduire à réduire des échéances, un ensemble à la densité élevée laisse une marge plus réduite pour effectuer de nouveaux regroupements.

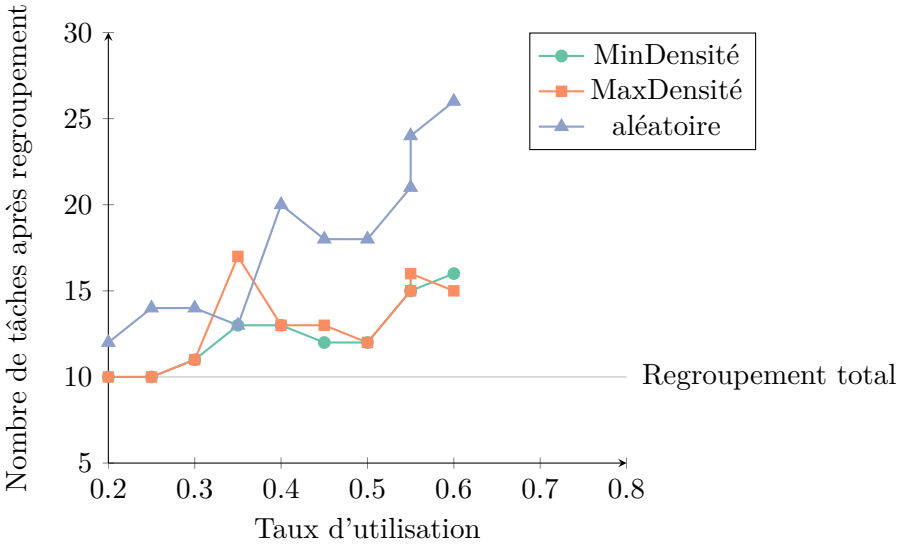


Fig. 7.4 : Comparaison des fonctions de coût en EDF

7.3.2 En fonction de la densité

À notre connaissance, il n'existe pas de technique qui permette de générer un ensemble de tâches avec un taux d'utilisation borné uniformément distribué, un nombre de périodes limité, et une densité particulière. Nous générons donc un grand nombre d'ensembles de 300 tâches, puis nous comparons le nombre de tâches obtenues selon chaque fonction de coût avec la densité de l'ensemble initial. Seul le nombre de tâches est fixé, le taux d'utilisation u est tiré aléatoirement comme les échéances pour 1000 exécutions. Dans ces expérimentions, nous avons utilisé l'optimisation qui consiste à calculer le pire temps de réponse après chaque regroupement réussi pour augmenter les chances de futurs regroupements à coût nul.

Les résultats de celles-ci sont dépeints dans la figure 7.5 pour DM et pour EDF dans la figure 7.6. Les fonctions de coût évaluées en EDF sont moins nombreuses qu'en DM. Par exemple, les fonctions de coût basées sur le pire temps de réponse sont absentes. Le test de pire temps de réponse étant beaucoup plus coûteux en EDF qu'en DM, il ne nous a pas été possible d'expérimenter cette fonction. Dans la figure 7.5, nous remarquons que choix de la fonction de coût n'a pas d'importance significative sous DM. Les fonctions de coût basées sur la densité et sur la densité vis-à-vis du pire temps de réponse affichent des performances équivalentes et sont à peine meilleures que la fonction aléatoire. En EDF, la fonction de coût basée sur la densité est la meilleure.

Les expérimentations développées ici tendent à montrer que le choix de la fonction de coût n'a qu'un impact faible sur la qualité du regroupement. En effet, nous observons dans la figure 7.5 et dans la figure 7.6 que l'écart entre la fonction de coût qui maximise la densité et celle qui minimise la densité n'est que d'environ 10 tâches dans la moyenne. Nous constatons que les fonctions de coût a priori pertinentes, telles que celles basées sur le pire temps de réponse ou la densité minimale, offrent des performances très similaires. Par ailleurs, les résultats de la fonction de coût aléatoire sont très proches de celle qui minimise la densité, bien qu'ils soient très légèrement moins bons.

L'utilisation d'une fonction de coût basée sur le pire temps de réponse implique l'emploi d'un test d'ordonnançabilité de type RTA. En ordonnancement à priorité fixe par travail tel que EDF, ce test est coûteux. Puisque les résultats ne démontrent pas d'intérêt significatif à utiliser une fonction de coût basée sur le temps de réponse, nous pouvons à première vue nous interroger sur la pertinence d'utiliser un tel test. Toutefois, comme indiqué dans la description de l'algorithme (cf. 6.3.4), l'information de pire temps de réponse permet d'effectuer les regroupements à coût

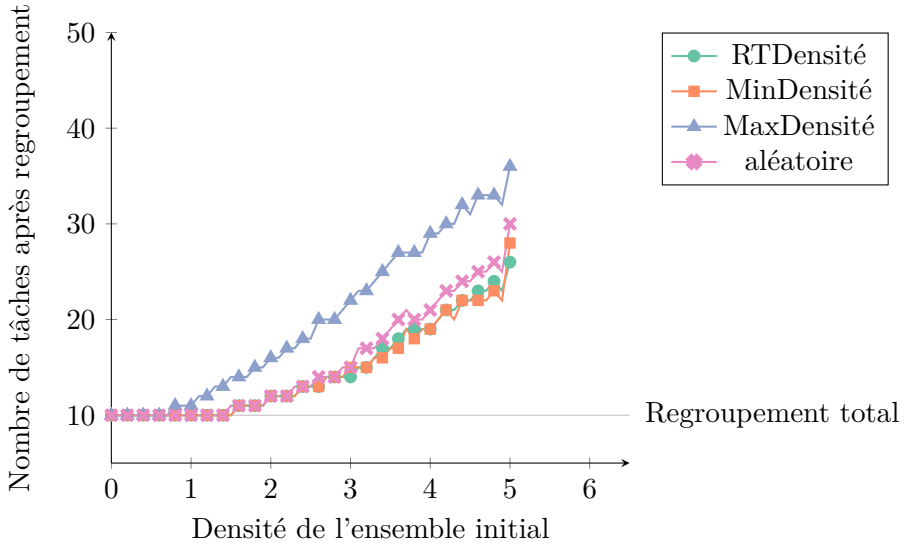


FIG. 7.5 : Comparaison des fonctions de coût en DM en fonction de la densité (n = 300)

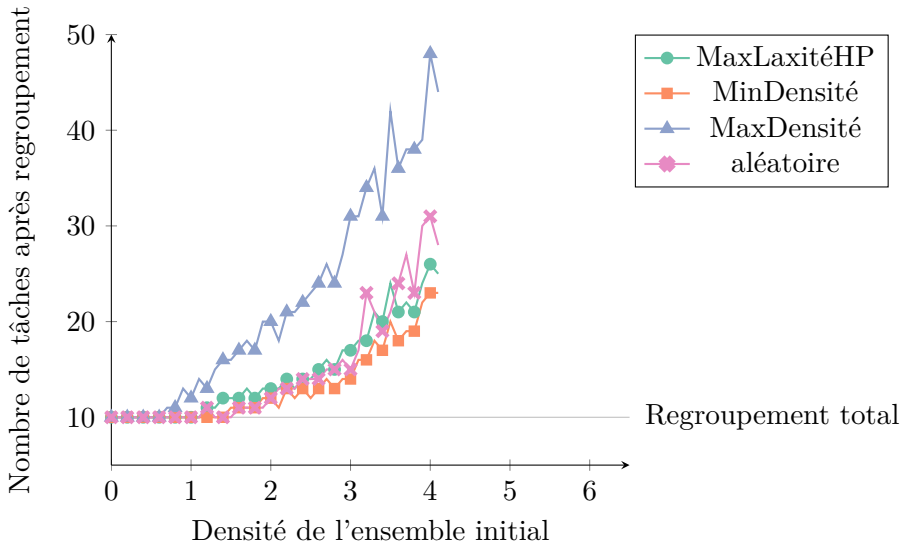


FIG. 7.6 : Comparaison des fonctions de coût en EDF en fonction de la densité (n = 300)

nul sur une condition plus favorable, il n'y a donc pas lieu de discréditer l'utilisation de l'analyse de temps de réponse dans le regroupement de tâches.

De manière générale, nous observons que la fonction de coût la plus performante est celle qui minimise la densité. De surcroît, il apparaît clairement que plus la densité est élevée dans l'ensemble initial, plus le nombre de tâches après regroupement est élevé et donc moins le regroupement est efficace. Les expérimentations menées en section 7.2 suggéraient que le taux d'utilisation avait un impact sur les performances du regroupement, nous découvrons ici que c'est la densité qui est la caractéristique la plus influente. Les résultats de la section 7.2 s'expliquent par le fait que plus le taux d'utilisation est élevé, plus il y a de la chance que la densité le soit aussi.

Dans la suite, nous baserons nos expérimentations sur la fonction de coût qui minimise la densité des ensembles candidats au regroupement.

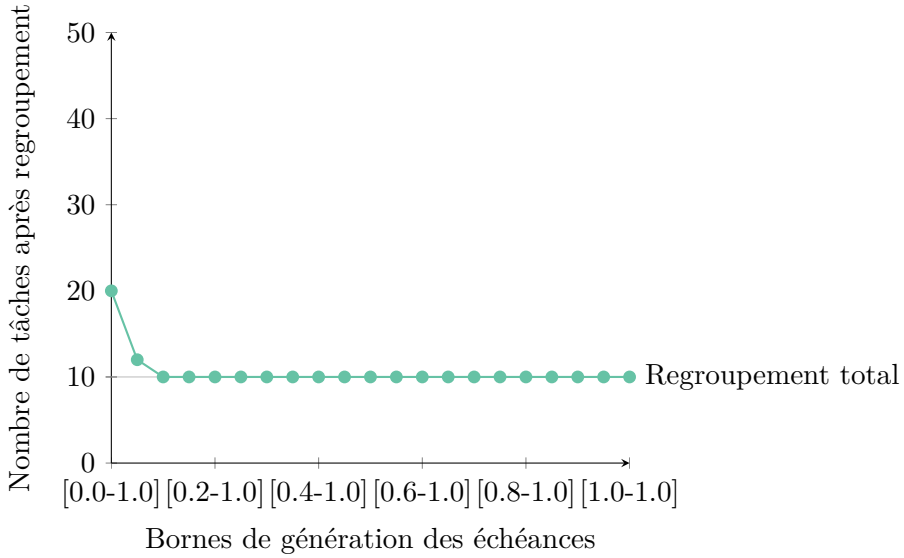


Fig. 7.7 : Variation de la borne inférieure de génération des échéances sur le nombre de tâches après regroupement en DM (u = 0.80, n = 300)

7.4 Variation des bornes de génération des échéances

Dans cette section, nous avons choisi de ne pas présenter les résultats obtenus sous EDF qui sont assez similaires à ceux produits sous DM. Les figure 7.7 et 7.8 représentent l'influence des bornes respectivement inférieure et supérieure de génération aléatoire des échéances sur le regroupement de tâches en DM. Une borne inférieure (d_{min}) à 0 correspond au pire temps d'exécution de la tâche et une borne supérieure (d_{max}) à 1 correspond à la période d'activation de la tâche. Pour ces résultats, le taux d'utilisation est fixé à 0.80, le nombre de tâches à 300 et la fonction de coût minimise la densité. Nous observons en figure 7.7 qu'à partir d'une borne inférieure de 0.20 (pour une borne supérieure constamment égale à 1), le regroupement devient total, c'est-à-dire que nous obtenons autant de tâches que de périodes. Cette indication corrobore l'hypothèse selon laquelle plus l'échéance est éloignée du pire temps d'exécution, plus il y a de marge disponible pour effectuer des regroupements de tâches.

La figure 7.8 indique que dans l'intervalle [0,0.5], le nombre de tâches obtenu après regroupement est égal à 0. Cela signifie qu'aucun ou que trop peu d'ensembles générés initialement étaient ordonnançables. Pour l'intervalle [0.5,1], nous observons que le nombre de tâches après regroupement décroît. Une nouvelle fois, ces résultats tendent à montrer que plus les échéances sont éloignées des pires temps d'exécution, plus la marge est grande et plus le regroupement est efficace.

7.5 Évolution des changements de contexte et des préemptions

En première partie, nous avons évoqué les changements de contexte et les préemptions à plusieurs titres. Nous avons distingué les changements de contexte dus aux préemptions (implicites) des changements de contexte explicites qui surviennent lorsque l'on passe de la fin d'exécution d'un travail au début d'un autre. Dans cette section, nous évaluons l'effet du regroupement de tâches sur le nombre de changements de contexte globaux (addition des changements de contexte implicites et explicites) et sur le nombre de préemptions. Les expérimentations ont été menées sur 1000 ensembles générés avec les paramètres suivants : taux d'utilisation aléatoire, échéances aléatoires et un nombre de tâches aléatoirement tiré entre 10 et 300 pour DM et pour EDF. Le nombre de changements de contexte globaux et le nombre de préemptions ont été obtenus

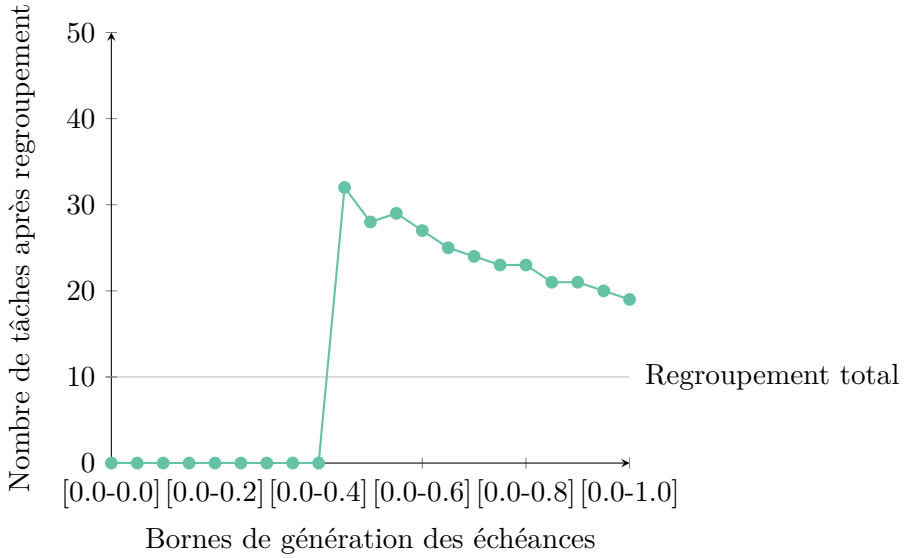


FIG. 7.8 : Variation de la borne supérieure de génération des échéances sur le nombre de tâches après regroupement en DM (u = 0.80, n = 300)

par simulation à l'aide du simulateur SimSo [31]. Le script d'exécution de SimSo en mode texte peut être facilement modifié afin de se procurer ces informations. Le script modifié est disponible avec le code du prototype. La figure 7.9 expose les pourcentages de diminution du nombre de changements de contexte globaux et du nombre de préemptions en fonction du pourcentage de diminution du nombre de tâches en DM après regroupement. Par exemple, nous observons qu'une diminution moyenne de 80% du nombre de tâches après regroupement provoque une baisse d'un peu moins de 80% du nombre de changements de contexte globaux mais aucune variation du nombre de préemptions (diminution environ égale à 0%). Plus généralement, les figures 7.9 et 7.10 nous apprennent que le regroupement de tâches diminue le nombre de changements de contexte globaux proportionnellement à la diminution du nombre de tâches. Néanmoins, il n'a aucune influence sur le nombre de préemptions. Nous précisons que les résultats ne prennent pas en compte les diminutions du nombre de tâches inférieures à 60%. En raison du nombre trop faible d'instances obtenues pour ces cas, nous ne les avons pas jugés assez significatifs pour les représenter. Dans la même veine, les résultats des changements de contexte globaux sont moins réguliers lorsque le taux de regroupement est inférieur à 80%, du fait du nombre plus faible d'échantillons obtenus.

Nous observons ici que le nombre de préemptions n'a pas diminué après le regroupement de tâches alors que le nombre de changements de contexte est constamment réduit d'environ 90%. Les travaux sur la diminution du nombre de préemptions présentés en partie une, telle que les *non preemption threshold* ou les *non preemption group* pouvaient apparaître comme des travaux concurrents. En effet, les systèmes d'exploitation temps réel contraignent le nombre de tâches utilisables dans le but de limiter les préemptions coûteuses. Or ces travaux n'ont pas d'impact sur le nombre de changements de contexte explicites. Il est donc possible de combiner les approches en appliquant d'abord le regroupement de tâches puis les techniques de limitation des préemptions pour diminuer conjointement les changements de contextes implicites et explicites.

Conclusion de la deuxième partie

Nous avons posé les bases du regroupement de tâches dans cette deuxième partie. Le regroupement de tâches est un problème difficile qui combine des problématiques d'optimisation et d'ordonnançabilité. Partant de ce constant, nous avons étudié des conditions qui permettent

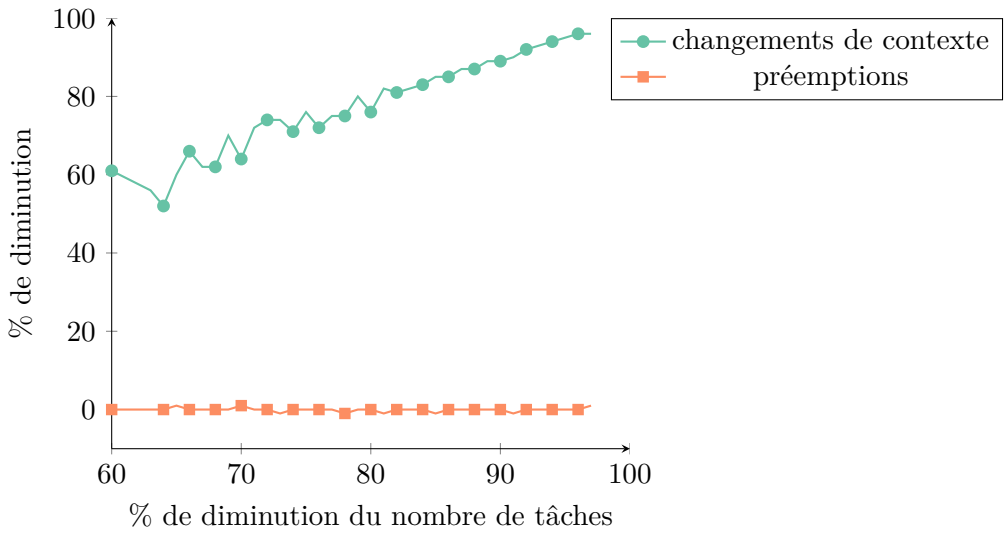


Fig. 7.9 : Impact de la diminution du nombre de tâches sur les changements de contexte et les préemptions en DM (n max = 300, u = aléatoire, $d_{min} = 0$, $d_{max} = 1$)

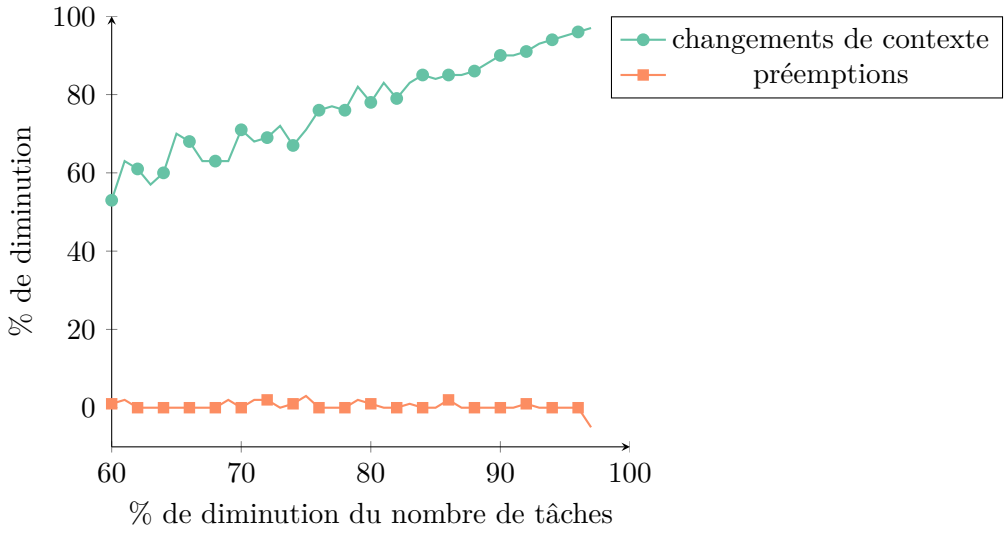


Fig. 7.10 : Impact de la diminution du nombre de tâches sur les changements de contexte et les préemptions en EDF (n max = 300, u = aléatoire, $d_{min} = 0$, $d_{max} = 1$)

de réduire l'espace de recherche et une méthode approchée de résolution. Le type d'heuristique choisi n'est pas fondamental dans ce travail. Nous avons cherché à dégager des principes pouvant être appliqués plus largement à d'autres méthodes approchées. Les principes de regroupement à coût nul et de la non-ordonnançabilité viable en font partie. Motivés également par les questions d'applicabilité des principes énoncés, nous avons éprouvé l'efficacité du regroupement de tâches dans plusieurs expérimentations sous des configurations différentes. Ces expérimentations avaient pour objectif d'identifier quels étaient les paramètres les plus déterminants sur la qualité des résultats obtenus.

Troisième partie

Regroupement de tâches avec contraintes de précédence en contexte monoprocesseur

Chapitre 8

Modèle et ordonnançabilité du regroupement de tâches

En pratique, les systèmes temps réel sont constitués de tâches qui communiquent. Ces communications ont lieu entre une tâche émettrice et une tâche réceptrice. Cette dernière ne peut pas commencer son exécution tant que la première n'a pas terminé la sienne. Cette exigence est représentée par une contrainte de précédence.

Premièrement dans ce chapitre, nous nous intéressons à l'ajout de contraintes de précédence entre les tâches dans le modèle des tâches indépendantes puis à l'adaptation des techniques pour tâches indépendantes. Dans le chapitre 5, nous avons présenté plusieurs algorithmes d'ordonnancement pour les tâches indépendantes. En second lieu, nous étudions l'ordonnancement de tâches avec contraintes de précédence. Ensuite, nous analysons l'impact du regroupement sur l'ordonnançabilité d'un ensemble de tâches avec contraintes de précédence.

Du point de vue temporel, nous conservons le modèle de tâche de la partie concernant les tâches dépendantes. Nous ajoutons au modèle des contraintes de précédence et notre ensemble de tâches devient un graphe acyclique dirigé de tâches où les sommets représentent les tâches et les arcs, les contraintes de précédence.

Ce travail vise les contraintes de précédence simples, c'est-à-dire que des dépendances ne peuvent exister qu'entre des tâches de même périodes.

8.1 Modèle de tâche dépendante

Cette section est dédiée à la définition du modèle des tâches avec contraintes de précédence et à l'introduction du vocabulaire nécessaire à la description des relations de dépendance entre les tâches.

Soit un ensemble de tâches $\mathcal{S} = (\{\tau_i(C_i, D_i, T_i)\}_{1 \leq i \leq n})$ tel que défini dans le chapitre 1. Nous modélisons un ensemble de tâches avec contraintes de précédence par un graphe acyclique dirigé $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ où $\mathbb{V}$ est l'ensemble de sommets correspondant à l'ensemble des tâches $\mathcal{S}$, $\mathbb{V} \equiv \mathcal{S}$ et $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ l'ensemble d'arcs qui représente les contraintes de précédence entre les tâches. Nous emploierons le terme *graphe* pour qualifier les graphes acycliques dirigés.

Du point de vue des contraintes de temps, nous nous basons sur le modèle des tâches indépendantes présenté dans le chapitre 5. Une tâche est caractérisée par son pire temps d'exécution C_i, son échéance relative D_i et sa période T_i. Les tâches sont synchrones donc $O_i = 0$ et les échéances contraintes donc $D_i \leq T_i$.

Dans la section 1.2.1, nous avons évoqué la contrainte de précédence entre deux tâches. Nous aurons besoin, pour la suite, de définir plus formellement les concepts liés à la notion de précédence. L'enchaînement de ces définitions est repris de [93].

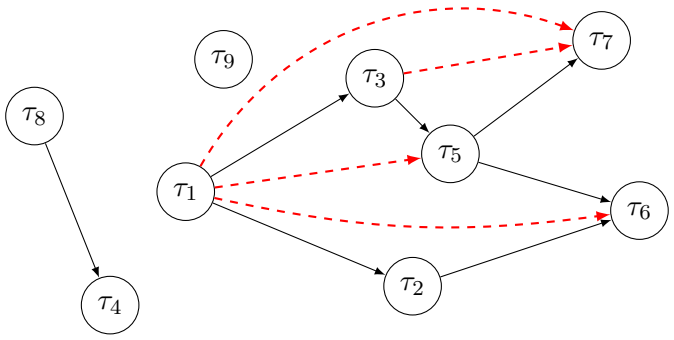


Fig. 8.1 : Regroupements infaisables représentés par des arcs rouges en pointillés ajoutés au graphe

Définition 12 (Contrainte de précédence directe $\prec$)
Soit un graphe $\mathcal{G} = (\mathbb{V}, \mathbb{E})$, un arc partant d'une tâche τ_i vers une tâche τ_j correspond à une contrainte de précédence directe $\tau_i \prec \tau_j$.

À titre d'exemple, dans la figure 8.1, il existe une précédence directe (représentée par un arc en trait continu) entre τ_1 et τ_3 mais pas entre τ_1 et τ_6. La contrainte de précédence est une extension transitive de la contrainte de précédence directe.

Définition 13 (Contrainte de précédence $\rightarrow$)
Soit un graphe $\mathcal{G} = (\mathbb{V}, \mathbb{E})$, il existe une contrainte de précédence entre τ_i et τ_j ($\tau_i \rightarrow \tau_j$) si :
$$\begin{cases} \tau_i \prec \tau_j \\ \tau_i \prec \tau_k \wedge \tau_k \prec \tau_j \end{cases}$$

En d'autres termes, il existe une contrainte de précédence entre τ_i et τ_j s'il existe un *chemin* dans le graphe menant de τ_i à τ_j.

Définition 14 (Chemin)
Dans un graphe orienté, il existe un chemin allant d'un sommet u à un sommet v si et seulement s'il existe une suite finie d'arcs consécutifs reliant u à v.

Par exemple, dans la figure 8.1, il existe une contrainte de précédence (et par la même un chemin) entre τ_3 et τ_7. Cette définition nous amène naturellement à celles qui concernent la dépendance de deux tâches.

Définition 15 (Dépendance $\triangleright$)
Soit un graphe $\mathcal{G} = (\mathbb{V}, \mathbb{E})$, τ_i et τ_j sont dépendantes ($\tau_i \triangleright \tau_j$), si et seulement si $\tau_i \rightarrow \tau_j$ ou $\tau_j \rightarrow \tau_i$.

La dépendance est une relation commutative donc $\tau_i \triangleright \tau_j \Leftrightarrow \tau_j \triangleright \tau_i$. L' *indépendance* $\not\triangleright$ est l'opposée de la dépendance $\triangleright$. C'est également une relation commutative donc $\tau_i \not\triangleright \tau_j \Leftrightarrow \tau_j \not\triangleright \tau_i$ Dans la figure 8.1, les tâches τ_1 et τ_7 sont dépendantes mais pas τ_9 et τ_4.

8.2 Modèle de regroupement

Dans cette section, nous définissons le modèle de regroupement de tâches dépendantes.

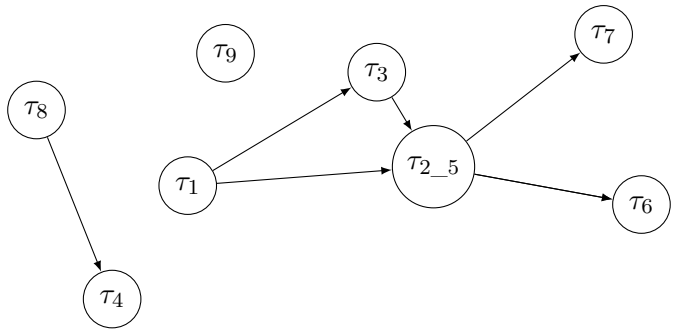


FIG. 8.2 : Graphe de la figure 8.1 après regroupement des sommets (tâches) τ_2 et τ_5 en τ_{2_5}

8.2.1 Contraintes de précédence

Les contraintes de précédence sont représentées dans un graphe. Le regroupement de tâches correspond à une fusion de sommets dans le graphe.

Définition 16 (Fusion de sommets dans un graphe)
Une paire de sommets u et v sont dits fusionnés si u et v sont remplacés par un nouvel unique sommet w tel que chaque arc entrant de u et v est un arc entrant de w et que chaque arc sortant de u et v est un arc sortant de w.

En conséquence, la fusion ne modifie pas le nombre d'arcs du graphe mais diminue le nombre de sommets de un. Dans la figure 8.2, les sommets τ_2 et τ_5 sont fusionnés en un sommet τ_{2_5}.

Remarque 8.2.1 *Dans notre modèle de tâches, nous pouvons identifier trois types de relations possibles entre deux tâches qui peuvent, relativement à la topologie du graphe, prétendre au regroupement :*

- *Il existe une précédence directe entre τ_i et τ_j, $\tau_i \prec \tau_j$ et donc $\tau_i \rightarrow \tau_j$,*
- *Les tâches sont « isolées », c'est-à-dire qu'aucune des deux tâches n'a de prédécesseur ou de successeur. Ce cas correspond au cas des tâches indépendantes,*
- *Les tâches sont indépendantes et une seule des deux tâches est isolée ($\tau_i \not\triangleright \tau_j$).*

Dans les autres cas, les tâches ne sont pas « regroupables ».

Ordre à l'intérieur d'un regroupement

Si les tâches regroupées sont dépendantes, l'ordre à l'intérieur du regroupement est déterminé par l'ordre des contraintes de précédence. Dans le cas contraire, l'ordre est imposé par les contraintes de temps de manière similaire au cas des tâches indépendantes du chapitre 5.

Il n'y pas de gestion de précédence avec d'autres tâches à l'intérieur d'un regroupement. Toutes les tâches qui précédaient les tâches concernées par le regroupement précèdent désormais le regroupement en entier, c'est-à-dire la première tâche ordonnée du regroupement. Semblablement, toutes les tâches qui étaient précédées par une tâche concernée par le regroupement sont maintenant précédées par le regroupement dans son ensemble. Cette restriction peut mener à des regroupements dits « infaisables » dans le sens où ils conduiraient au final à un ordonnancement infaisable.

Remarque 8.2.2 *Du point de vue des communications, un regroupement peut retarder la production d'une donnée. Si une tâche τ_i est regroupée avec une tâche τ_j et que τ_i produisait une donnée pour une tierce tâche, celle-ci doit désormais attendre la terminaison complète de la tâche regroupée pour consommer la donnée. Par ailleurs, regrouper deux tâches reliées par une contrainte de précédence permet de faire l'économie d'une communication.*

(a) $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$

(b) Dépendance circulaire avec τ_2 créée par le regroupement τ_{1_3}

Fig. 8.3 : Regroupement infaisable de τ_1 et τ_3

Regroupements infaisables

Au regard des contraintes de précédence et du modèle de regroupement, certains regroupements sont infaisables en inspectant le graphe. Dans la figure 8.3a, il existe des contraintes de précédence entre τ_1 et τ_2 et entre τ_2 et τ_3 ($\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$). τ_1 et τ_3 ne peuvent pas être regroupées puisque cela impose d'exécuter τ_2 avant ou après τ_1 et τ_3 ce qui va à l'encontre des contraintes de précédence établies. En effet, si τ_1 et τ_3 sont regroupées alors τ_2 ne peut plus à la fois précéder τ_3 et être précédée par τ_1. Au regard de la topologie du graphe ce regroupement formerait une dépendance circulaire entre τ_{1_3} et τ_2, représentée en figure 8.3b. Cette dépendance circulaire conduit à un ordonnancement trivialement infaisable puisque les contraintes de précédence ne seront jamais satisfaites.

Définition 17 (Regroupement infaisable)
Le regroupement de deux tâches τ_i et τ_j est fonctionnellement infaisable si τ_i et τ_i sont dépendantes ($\tau_i \triangleright \tau_j$), excepté s'il existe une contrainte de précédence directe entre τ_i et τ_j ($\tau_i \prec \tau_j$).

Remarque 8.2.3 *La définition 17 fait l'hypothèse qu'il n'existe pas d'arc entre deux tâches s'il existe déjà un chemin qui les relient entre elles. Dans notre, travail nous ne tenons pas compte des arcs redondants puisque considérés comme des contraintes de précédence directes, ils pourraient mener à des regroupements infaisables. Par exemple, s'il existait un arc supplémentaire dans la figure 8.3a entre τ_1 et τ_3 alors cet arc ne serait pas considéré car il existe déjà un chemin entre τ_1 et τ_3 passant par τ_2. En tout état de cause, le regroupement de τ_1 et τ_3 est infaisable. Par conséquent, nous ne changeons pas la nature du problème en examinant uniquement les chemins uniques existant entre chacune des tâches.*

La figure 8.1 représente les regroupements infaisables d'un graphe de tâches. Ceux-ci sont matérialisés par des arcs rouges en pointillés.

8.2.2 Contraintes de temps du regroupement

Nous ne regroupons toujours que des tâches de même période. Par ailleurs, nous traitons le cas des contraintes de précédence simples donc il n'y pas de contrainte de précédence entre des tâches de différentes périodes. Nous avons distingué trois types de relation possibles entre des tâches dites « regroupables » dans la remarque 8.2.1. Nous ne traitons ici que le cas des tâches qui entretiennent une relation de précédence directe entre elles. Si au moins l'une des deux tâches est isolée, les conditions de regroupement sont les mêmes que pour les tâches indépendantes définies en chapitre 5.

Regrouper des tâches τ_i et τ_j dans cet ordre avec $\tau_i \rightarrow \tau_j$ produit un regroupement τ_{ij} avec les paramètres suivants :

$$C_{ij} = C_i + C_j \tag{8.1}$$

$$T_{ij} = T_i = T_j \tag{8.2}$$

$$D_{ij} = \begin{cases} D_j & \text{si } D_i > D_j & \text{(8.3a)} \\ D_j & \text{sinon si } (D_i \leq D_j) \wedge ((D_j - C_j \leq D_i) \vee (R_j - C_j \leq D_i)) & \text{(8.3b)} \\ D_i & \text{sinon.} & \text{(8.3c)} \end{cases}$$

Échéance du regroupement

Comme dans le cas des tâches indépendantes, nous choisissons l'échéance du regroupement de manière à ce que chaque tâche respecte son échéance initiale après regroupement. L'ordre dans le regroupement garantit le respect de la contrainte de précédence si elle existe. Plus formellement :

Théorème 8
Soit $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ un graphe de tâches synchrones et deux tâches τ_i et τ_j avec $T_i = T_j$. Soit $\mathcal{G}' = (\mathcal{G} \setminus \{\tau_i, \tau_j\} \cup \{\tau_{ij}\})$, un graphe de tâches synchrones après regroupement des tâches τ_i et τ_j en τ_{ij} et fusion des sommets correspondants.

$$\mathcal{G}' \text{ est ordonnançable sous } \Phi \Rightarrow \begin{cases} \forall \tau_{ij.k}, e(\tau_{i'.k}) \leq d_{i.k} \\ \forall \tau_{ij.k}, s(\tau_{i'.k}) \geq o_{i.k} \\ \forall \tau_{i.k} \rightarrow \tau_{j.k}, e(\tau_{i'.k})) \leq o_{j.k} \end{cases}$$

Preuve *Dans le cas 8.3a, τ_i est placée avant τ_j dans le regroupement ce qui satisfait donc trivialement la contrainte de précédence entre τ_i et τ_j dans $\mathcal{G}$. Ensuite $D_{ij} \leq D_j < D_i$ donc si $\mathcal{G}'$ est ordonnançable alors les parts de τ_i et τ_j dans τ_{ij} respectent leur échéance initiale et ne s'exécutent pas plus tôt que leur date de réveil (nous dirons qu'elles garantissent leurs contraintes de temps initiales).*

Dans le chapitre 5 propre aux tâches isolées (originalement appelées indépendantes mais la définition diffère dans cette partie), nous avons montré que regrouper que τ_i et τ_j dans cet ordre avec $D_i \leq D_j$ sur D_j en respectant la condition de regroupement à coût nul permettait de garantir que si $\mathcal{G}'$ est ordonnançable alors les parts de τ_i et τ_j dans τ_{ij} respectent leurs contraintes de temps initiales. Regrouper les tâches dans cet ordre assure le respect de la contrainte de précédence entre τ_i et τ_j. Or la condition 8.3b correspond à la condition de regroupement à coût nul. En conséquence, comme pour le cas des tâches isolées sous la condition de regroupement à coût nul valide, $\mathcal{G}'$ ordonnançable implique que les parts de τ_i et τ_j dans τ_{ij} respectent leurs contraintes de temps initiales si la condition 8.3b est respectée. Nous verrons cependant plus loin que regrouper sur la condition de regroupement à coût nul des tâches indépendantes n'est pas synonyme de regroupement à coût nul en tâches indépendantes.

Le cas 8.3c est similaire au cas des tâches isolées du chapitre 5 où l'on contraint l'ordre des tâches dans le regroupement de façon à garantir la contrainte de précédence. Regrouper sur l'échéance de la tâche prédécesseur quand elle est la plus courte des deux assure le respect des contraintes de temps initiales de τ_i et τ_j.

8.3 Ordonnancement de tâches avec contraintes de précédence

8.3.1 Ordonnançabilité

Il est à la fois nécessaire de respecter les contraintes de temps et l'ordre partiel entre les tâches établi par les contraintes de précédence.

Un système consiste en un graphe de tâches synchrones sous contraintes de précédence $\mathcal{G} = (\mathbb{V}, \mathbb{E})$. Soit, $\rightarrow$ un ordre partiel défini sur $\mathcal{G}$.

Définition 18 (Ordonnançabilité d'un ensemble de tâches dépendantes)
Soit un graphe de tâches $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ et une assignation de priorités Φ. $\mathcal{G}$ est ordonnançable *sous Φ si et seulement si :*

$$\begin{cases} \forall \tau_{i.k}, e_\Phi(\tau_{i.k}) \leq d_{i.k} \wedge s_\Phi(\tau_{i.k}) \geq o_{i.k} \\ \forall \tau_{i.k} \rightarrow \tau_{j.k}, e_\Phi(\tau_{i.k}) \leq o_{j.k} \end{cases}$$

8.3.2 Algorithmes d'ordonnancement

Nous avons traité le regroupement de tâches indépendantes en appliquant des algorithmes d'ordonnancement à priorité fixe par tâche (DM) et à priorité fixe par travail (EDF). Ces algorithmes ne prennent pas en compte les contraintes de précédence. D'abord, nous présentons succinctement plusieurs méthodes d'ordonnancement de tâches avec contraintes de précédence. Nous détaillons ensuite la méthode d'ordonnancement choisie qui correspond au travail de Chetto et al. [32].

Bref état de l'art

L'ordonnancement de tâches dépendantes avec contraintes de précédence simple en monoprocesseur est un problème bien examiné dans la littérature temps réel. Lorsque le respect des contraintes de précédences est assuré par des sémaphores, Richard et al. [87] proposent une condition suffisante d'ordonnançabilité pour des ordonnancements à priorité fixe par tâche. Si les contraintes de précédence ne sont pas assurées par des mécanismes de synchronisation, une autre approche consiste à modifier les priorités des tâches et les dates de réveil. Dans ce cadre, pour certains modèles de tâches dépendantes avec contraintes de bout-en-bout, une échéance unique est assignée pour toutes les tâches d'une même chaîne (également appelée transaction). Il est alors nécessaire d'assigner des échéances aux tâches intermédiaires de manière à utiliser les algorithmes classiques d'ordonnancement. Le problème d'assignation optimale des échéances est NP-difficile [84]. Des travaux comme celui de Palencia et Harbour [81] proposent des heuristiques afin d'assigner les échéances des tâches intermédiaires proportionnellement à leurs temps d'exécution. Néanmoins, ces méthodes requièrent d'avoir des tâches pour lesquelles les échéances ne sont pas des contraintes mais uniquement des paramètres que l'on peut fixer librement, ce qui n'est pas notre cas.

Nous choisissons ici le technique d'encodage des contraintes de précédence de Chetto et al. [32].

Encodage des contraintes de précédence

Le principe global de l'encodage consiste à modifier les contraintes de temps de manière à ce que les contraintes de précédence soient implicitement respectées. Ainsi, nous obtenons un ensemble de tâches indépendantes qu'il est possible d'ordonnancer avec les algorithmes d'ordonnancement pour tâches indépendantes (comme DM ou EDF). Nous notons que si la méthode est originalement prouvée valide pour l'algorithme EDF, il a été prouvé par Forget et al. [48] qu'elle l'était également pour les algorithmes à priorité fixe par tâches comme DM. Par ailleurs, la méthode concerne l'encodage des contraintes de précédence simples, c'est-à-dire entre tâches de même période. Nous donnons ci-dessous une description des techniques d'ajustement des contraintes de temps ainsi qu'un exemple pour l'illustrer.

Soit deux tâches τ_i et τ_j synchrones avec $T_i = T_j$ et $\tau_i \rightarrow \tau_j$. Dans un ordonnancement valide, $\forall \tau_{i.k}, \tau_{j.k}, e(\tau_{i.k}) \leq d_{i.k}$ et $e(\tau_{j.k}) \leq d_{j.k}$. τ_i doit s'exécuter prioritairement à τ_j pour respecter la contrainte de précédence. Mais τ_i doit également laisser suffisamment de temps à τ_j pour s'exécuter avant son échéance. Or, τ_j nécessite C_j unités de temps pour terminer son exécution. La contrainte de précédence est donc assurée si $\forall \tau_{i.k}, \tau_{j.k}, e(\tau_{i.k}) \leq d_{j.k} - C_j$. En conséquence, un système de tâches indépendantes ordonnançable avec D_i fixée à $D_j - C_j$ assure la contrainte de précédence entre τ_i et τ_j.

L'algorithme d'ajustement des échéances opère en premier lieu sur les tâches sans successeur et modifie successivement les échéances selon l'équation 8.4, en suivant l'ordre topologique inverse du graphe (des successeurs vers les prédécesseurs). Dans la suite, l'échéance relative encodée est notée D_i^* et la date de réveil encodée O_i^*.

	C_i	D_i	O_i
τ_1	2	15	2
τ_2	1	10	1
τ_3	2	12	0
τ_5	3	15	0
τ_6	4	18	0
τ_7	3	22	0

TAB. 8.1 : Contraintes de temps de quelques tâches de la figure 8.1

$$D_i^* = \min(D_i, \min_{\tau_j \in succs(\tau_i)}(D_j^* - C_j)) \tag{8.4}$$

Notons que dans le cas de tâches asynchrones, il est également nécessaire d'ajuster les dates de réveil des tâches pour assurer qu'une tâche ne puisse se réveiller avant la fin d'exécution de tous ses prédécesseurs. Dans un ordonnancement valide, τ_j ne peut pas être activée avant τ_i qui s'exécute pour C_i unités de temps, donc $\forall \tau_{i.k}, \tau_{j.k}, o_{j.k} \geq o_{i.k} + C_i$. En conséquence, l'ordonnancement est valide si O_j est fixée à $O_i + C_i$ et si les échéances sont ajustées comme détaillé plus haut.

L'algorithme d'ajustement des dates de réveil opère en premier lieu sur les tâches sans prédécesseur et modifie successivement les dates de réveil suivant l'équation 8.5, en suivant l'ordre topologique du graphe (des tâches sans prédécesseur vers les successeurs).

$$O_i^* = \max(O_i, \max_{\tau_j \in preds(\tau_i)}(O_j^* + C_j)) \tag{8.5}$$

De manière générale, les auteurs ont prouvé qu'un graphe de tâches $\mathcal{G}$ avec contraintes de précédence est ordonnançable si et seulement si son équivalent encodé $\mathcal{G}^*$ par les algorithmes DM et EDF est ordonnançable. Notons que les algorithmes d'ajustement des échéances et des dates de réveil ont une complexité en $\mathcal{O}(n^2)$ où n correspond au nombre de tâches. L'exemple 8.3.1 illustre l'application des algorithmes d'encodage sur la composante connexe du graphe de la figure 8.1 formée par les tâches décrite dans le tableau 8.1.

Exemple 8.3.1

$$\begin{aligned}
D_7^* &= 22 \\
D_6^* &= 18 \\
D_5^* &= min(D_5, D_7^* - C_7, D_6^* - C_6) = min(15, 19, 14) = 14 \\
D_3^* &= min(D_3, D_5^* - C_5) = min(12, 11) = 11 \\
D_2^* &= min(D_2, D_6^* - C_6) = min(10, 14) = 10 \\
D_1^* &= min(D_1, D_2^* - C_2, D_3^* - C_3) = min(15, 9, 9) = 9
\end{aligned}$$

$$\begin{aligned}
O_1^* &= 2 \\
O_2^* &= max(O_2, O_1 + C_1) = max(2, 4) = 4 \\
O_3^* &= max(O_3, O_1^* + C_1, O_2^* + C_2) = max(0, 4, 5) = 5 \\
O_5^* &= max(O_5, O_3^* + C_3) = max(0, 6) = 6 \\
O_6^* &= max(O_6, O_2^* + C_2, O_5^* + C_5) = max(0, 5, 9) = 9 \\
O_7^* &= max(O_7, O_5^* + C_5) = max(0, 9) = 9
\end{aligned}$$

Tâche	C_i	D_i	T_i	D_i^*
τ_i	2	5	15	5
τ_j	3	8	15	8
τ_k	1	6	15	6
τ_l	1	6	15	
τ_{ij}	5	5	15	5

Tab. 8.2 : Caractéristiques des tâches dépendantes de $\mathcal{T}$ avant et après regroupement

8.4 Impact du regroupement sur l'ordonnançabilité

Dans cette section, nous mettons en exergue qu'il n'est pas possible de regrouper des tâches sans vérifier l'ordonnançabilité après regroupement. Autrement dit, contrairement au cas des tâches isolées évoqué en chapitre 5, il n'y a pas de regroupement à coût nul.

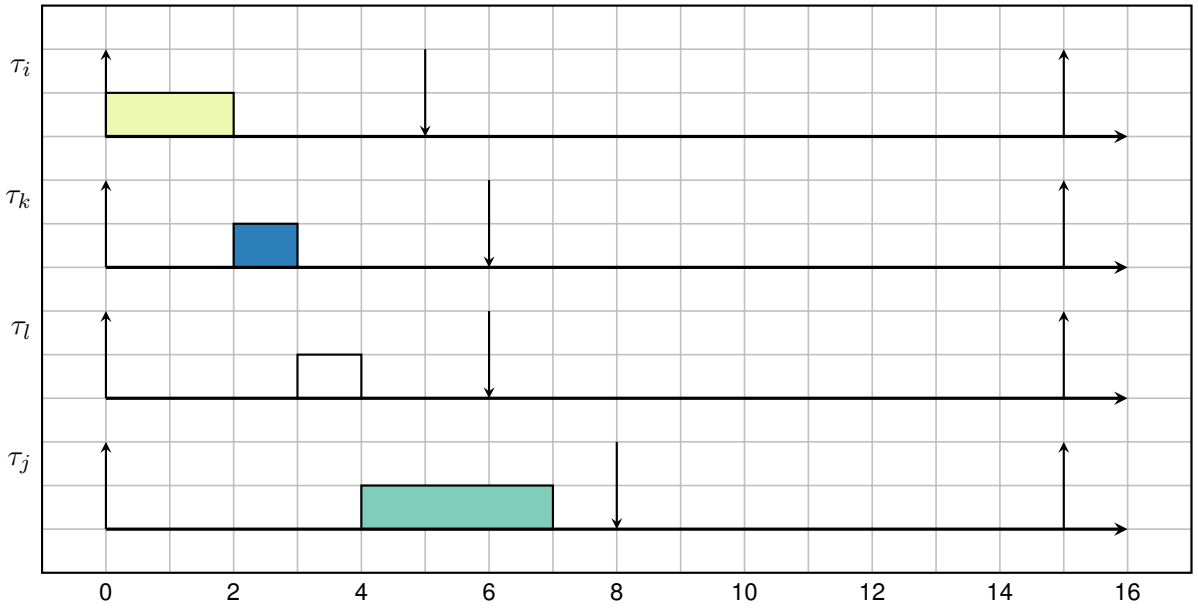
Remarque 8.4.1 *Dans les cas des tâches dépendantes, $\mathcal{G}$ ordonnançable sous $\Phi \not\Rightarrow \mathcal{G}'$ ordonnançable.*

Nous démontrons par un contre-exemple que la condition (8.3a) n'implique pas de regroupement à coût nul comme pour le cas des tâches sans contrainte de précédence.

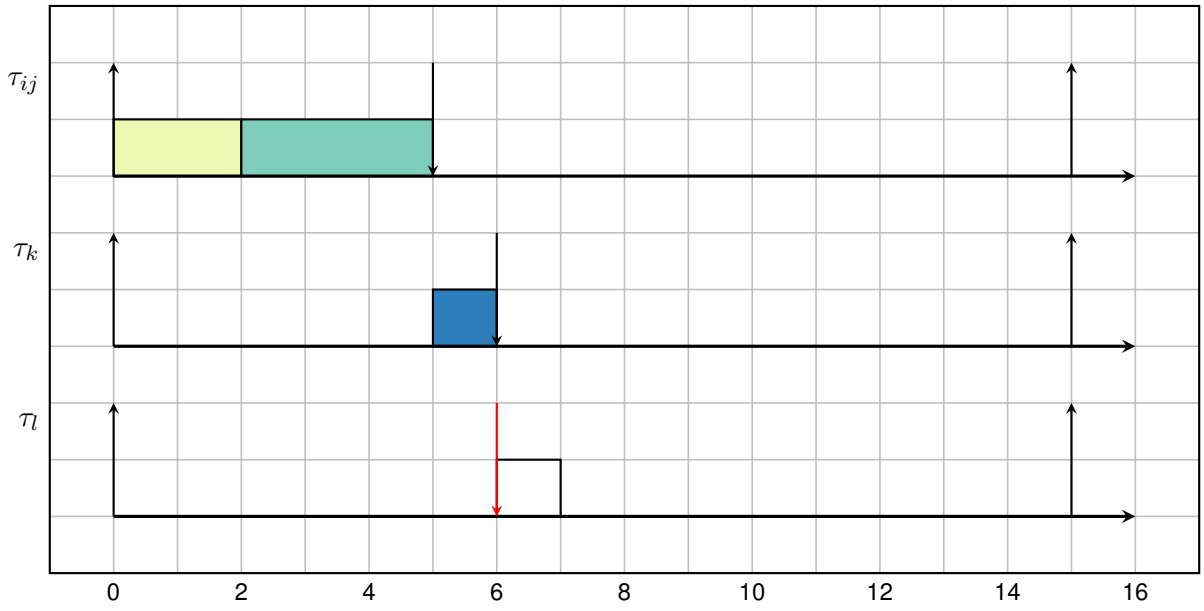
Soit un ensemble de tâches $\mathcal{T} = \{\tau_i, \tau_j, \tau_k, \tau_l\}$ ordonnançable avec $\tau_i \rightarrow \tau_j$ et $\tau_i \rightarrow \tau_k$. Les contraintes de temps sont décrites dans le tableau 8.2 et les contraintes de précédence dans la figure 8.5a.

Une période d'exécution de l'ensemble $\mathcal{T}$ est représentée dans la figure 8.4a. Notons que l'encodage ne modifie pas ici les échéances initiales, c'est-à-dire que l'ensemble initial et l'ensemble ajusté sont identiques. L'ensemble $\mathcal{T}$ est ordonnançable et la condition (8.3a) est vérifiée. Nous procédons au regroupement de τ_i et τ_j dans τ_{ij} sur l'échéance D_j, l'évolution des contraintes de précédence apparaît dans la figure 8.5b. Nous observons dans le diagramme de la figure 8.4b qu'après regroupement et encodage des contraintes de précédence, l'ensemble $\mathcal{T}$ n'est plus ordonnançable puisque la tâche τ_l manque son échéance à la date 6. Il faut noter que le regroupement a bien eu lieu sur l'échéance de D_j mais qu'après encodage, celle-ci devient égale à l'échéance initiale de D_i.

La condition du regroupement à coût nul en tâches dépendantes ne permet plus de se passer d'un test d'ordonnançabilité. Néanmoins, elle autorise le regroupement sur l'échéance maximale plutôt que sur l'échéance minimale tout en garantissant le respect des contraintes initiales. De ce fait, elle reste une condition plus favorable puisqu'elle ne rogne pas de marge disponible pour de futurs regroupements.



(a) Diagramme d'exécution de $\mathcal{T}$



(b) Diagramme d'exécution de $\mathcal{T}$ après regroupement de τ_i et τ_j

FIG. 8.4 : Effet du regroupement sur le diagramme d'exécution de $\mathcal{T}$

(a) Contraintes de précédence de $\mathcal{T}$

(b) Contraintes de précédence de $\mathcal{T}$ après regroupement de τ_i et τ_j

FIG. 8.5 : Évolution des contraintes de précédence lors du regroupement

Chapitre 9

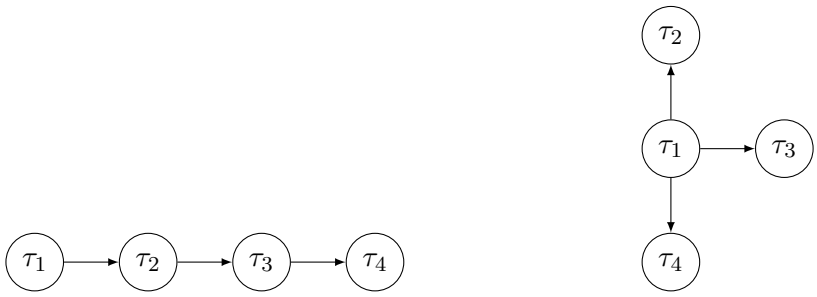
Minimisation du nombre de tâches

Dans ce chapitre, nous discutons dans un premier temps de l'évolution de l'espace de recherche et de la complexité du problème de regroupement de tâches dépendantes. Nous adaptons ensuite l'heuristique de regroupement de tâches indépendantes pour prendre en compte les contraintes de précédence.

9.1 Espace des solutions et complexité

En tâches indépendantes, l'espace des solutions de l'heuristique est de l'ordre du nombre de Bell. Dans le chapitre 8, nous avons établi que certains regroupements étaient infaisables du fait des contraires de précédence. L'espace des solutions peut être diminué du fait des regroupements infaisables. Dans le cas où toutes les tâches sont dépendantes, c'est-à-dire concernées par les contraintes de précédence, nous pouvons rencontrer deux situations extrêmes. Dans le cas d'une topologie en chaîne, le nombre de regroupements faisables est minimal au premier pas de l'heuristique. Nous observons dans la figure 9.1a que seuls les regroupements entre les prédécesseurs et leurs successeurs directs sont possibles. τ_1 et τ_2 sont par exemple éligibles au regroupement, mais pas τ_1 et τ_3 ou encore τ_2 et τ_4. La figure 9.1 représente une topologie de graphe en étoile. Cette situation s'apparente à celle des tâches indépendantes. En effet, à ce niveau tous les regroupements sont faisables du point de vue de la topologie du graphe. Les dépendances peuvent donc restreindre l'espace des solutions mais la borne supérieure sur le nombre de regroupements possibles correspond à la situation des tâches indépendantes.

Lemme 3
Le problème du regroupement de tâches avec contraintes de précédence en monoprocesseur est au moins aussi difficile que le regroupement de tâches indépendantes.



(a) Topologie en chaîne : regroupements entre successeurs/prédécesseurs directs uniquement possibles

(b) Topologie en étoile : aucun regroupement infaisable

Fig. 9.1 : Espace de recherche en fonction de la topologie du graphe

Preuve *Nous avons montré dans le chapitre 6 que le regroupement de tâches indépendantes en monoprocesseur était au moins aussi dur qu'un problème co-NP-complet. Or le cas des tâches indépendantes est un cas particulier des tâches dépendantes.*

9.2 Adaptation de l'heuristique

Nous avons choisi d'adapter l'heuristique décrite en chapitre 6 au cas des tâches dépendantes. L'adaptation réside principalement dans l'encodage. Si le regroupement est faisable compte tenu de la topologie du graphe, nous appliquons l'encodage des contraintes de précédence de manière à transformer un ensemble de tâches dépendantes en un ensemble de tâches indépendantes. De ce fait, il est possible d'appliquer les algorithmes d'ordonnancement monoprocesseur.

9.2.1 Traitement des regroupements infaisables

Nous présentons deux méthodes pour ne pas regrouper des tâches que l'on ne peut pas fusionner vis-à-vis de la topologie du graphe.

À l'aide de la fermeture transitive du graphe

La figure 8.1 représente les regroupements fonctionnellement infaisables d'un graphe par des arcs en pointillés. Nous observons que les regroupements fonctionnellement infaisables correspondent en fait aux arcs ajoutés au graphe pour réaliser sa fermeture transitive. Par exemple, un arc en pointillés a été ajouté entre les sommets τ_1 et τ_6 par construction de la fermeture transitive du graphe. Il n'est donc pas possible de regrouper les tâches τ_1 et τ_6.

Définition 19 (Fermeture transitive d'un graphe)
La fermeture transitive d'un graphe $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ est définie par un graphe $\mathcal{G}' = (\mathbb{V}, \mathbb{E}')$ tel que s'il existe un chemin entre deux sommets u et v appartenant à $\mathbb{V}$ dans $\mathcal{G}$ alors il existe un arc (u, v) appartenant à $\mathbb{E}'$ dans $\mathcal{G}$.

Généralement, la fermeture transitive d'un graphe permet d'identifier s'il existe un chemin entre un sommet u et un sommet v dans le graphe. L'algorithme de Warshall [104] calcule la fermeture transitive d'un graphe représenté sous la forme d'une *matrice d'adjacence* avec une complexité de $\mathcal{O}(n^3)$ où n correspond au nombre de sommets.

Définition 20 (Matrice d'adjacence d'un graphe)
La matrice d'adjacence $\mathcal{A}$ d'un graphe $\mathcal{G}$ à n sommets est une matrice booléenne $n \times n$ dans laquelle la valeur de $\mathcal{A}_{ij}$ dénote la présence ou l'absence d'un arc entre les sommets i et j. Formellement, soit $\mathcal{G} = (\mathbb{V}, \mathbb{E})$,

$$\mathcal{A}_{ij} = \begin{cases} 1 & si\ (v_i, v_j) \in \mathbb{E} \\ 0 & sinon. \end{cases}$$

La matrice 9.2 est la matrice d'adjacence du graphe en figure 8.1.

L'idée relevée ici consiste à construire une fermeture transitive du graphe transitivement réduit et à identifier de manière singulière tous les arcs ajoutés. Par exemple, si le regroupement de deux tâches τ_i et τ_j est fonctionnellement infaisable, il peut être représenté par un 2 dans la matrice d'adjacence ($\mathcal{A}_{ij} = 2$). Ainsi, il suffit de vérifier en temps constant dans la matrice d'adjacence qu'un regroupement est possible ou non. En théorie, il s'agit donc d'appliquer dans le pire cas n fermeture(s) transitive(s) de complexité $\mathcal{O}(n^3)$.

En théorie des graphes, la littérature est riche sur la mise à jour dynamique de fermeture transitive de graphe [62]. Généralement, elle concerne la mise à jour de la fermeture transitive après ajout ou suppression d'arcs. Elle opère en complexité quadratique et permet d'éviter de

$$\mathcal{A} = \begin{array}{c} \\ \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \\ \tau_7 \\ \tau_8 \\ \tau_9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} \tau_1 & \tau_2 & \tau_3 & \tau_4 & \tau_5 & \tau_6 & \tau_7 & \tau_8 & \tau_9 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

FIG. 9.2 : Matrice d'adjacence du graphe de la figure 1.3

calculer de nouveau l'entière fermeture, ce qui est coûteux. Dans notre travail, le regroupement de tâches produit des fusions dans le graphe. A notre connaissance, il n'y pas de travaux en théorie des graphes qui permettent de mettre à jour la fermeture transitive d'un graphe après ajout, suppression ou fusion de nœuds. En ce sens, nous effectuons une nouvelle fermeture transitive complète après chaque regroupement.

En utilisant le tri topologique de l'encodage

L'encodage des contraintes de précédence décrit en chapitre 8 permet de transformer un ensemble de tâches dépendantes en un ensemble de tâches indépendantes équivalant. Il consiste à modifier les échéances et les dates de réveil de manière à assurer le respect des contraintes de précédence. Ces algorithmes procèdent à un tri topologique des sommets du graphe. L'algorithme de tri topologique de Kahn [60] calcule une fermeture transitive du graphe si ce dernier est un DAG. Dans le cas contraire, il détecte alors que le graphe possède au moins un cycle. En conséquence, il nous est possible de déceler un regroupement infaisable dans le graphe puisque celui-ci formerait un cycle comme remarqué en figure 8.3. La technique consiste alors à essayer tous les regroupements possibles sans se préoccuper de la topologie du graphe à la manière de l'heuristique des tâches indépendantes puis de ne pas sélectionner les solutions pour lesquelles l'encodage n'est pas réalisable. La complexité d'un tri topologique est quadratique.

Le gain en complexité de l'utilisation de l'une des deux méthodes plutôt que l'autre n'est pas évident et dépend de la topologie du graphe. Dans notre prototype, c'est la méthode de la fermeture transitive qui a été privilégiée. Nous considérons que ce choix relève de l'implantation et il n'est pas détaillé dans l'algorithme.

9.2.2 Principe de l'heuristique et algorithme

Nous reprenons le parcours récursif des solutions proposé pour les tâches indépendantes dans la section 6.3.2. Nous vérifions avant toute tentative que les tâches parcourues ne peuvent pas produire un regroupement infaisable. En fonction des relations de dépendance entretenues par les deux tâches dans le graphe, nous regroupons sur l'échéance adéquate de manière à respecter les contraintes initiales de chacune d'elles. Nous utilisons la technique d'ajustement des échéances sur l'ensemble après regroupement de manière à vérifier la préservation de l'ordonnançabilité, à l'aide des tests du contexte monoprocesseur. Notons qu'excepté le cas où les tâches sont isolées, il n'est plus possible d'effectuer de regroupement à coût nul, c'est-à-dire de regrouper sans tester à nouveau l'ordonnançabilité.

Comme décrit dans l'algorithme 2, nous énumérons récursivement les partitions possibles du graphe de tâches de départ. Pendant le parcours, nous n'inspectons que les tâches qui sont dites « regroupables » suivant les principes présentés en section 9.2.1. La fonction `regroupable(`τ_i`,`τ_j`)`

retourne vrai si les tâches τ_i et τ_j ont la même période ($T_i = T_j$) et si leur fusion dans le graphe respecte les contraintes de précédence initiales, faux sinon.

Dans le cas où les tâches sont « regroupables », elles peuvent être isolées, c'est-à-dire qu'aucune d'elles n'a de contrainte de précédence avec autre tâche. La fonction `isolées(`τ_i`,`τ_j`)` retourne vrai si les tâches τ_i et τ_j sont dans cette situation, faux sinon. Dans ce cas, nous appliquons le principe des tâches indépendantes. Si la condition de regroupement à coût nul est vérifiée alors nous effectuons un appel récursif sur ce nouveau regroupement (qui inclut la fusion dans le graphe), sinon, nous accumulons un quadruplet contenant le graphe de tâches, les indices des deux tâches concernées et l'échéance choisie dans une liste `fils`.

Si les tâches ne sont pas isolées, elles peuvent entretenir une relation de précédence directe. Si c'est le cas, la fonction `précédenceDirecte(`τ_i`,`τ_j`)` retourne vrai et faux sinon. Le quadruplet contient l'échéance choisie conformément aux conditions 8.3c énoncées plus tôt. L'échéance du regroupement peut prendre les valeurs suivantes :

- $DlMin$: échéance minimale des deux tâches ;
- $DlMax$: échéance maximale des deux tâches ;
- $DlPred$: échéance de la tâche qui constitue le prédécesseur de l'autre ;
- $DlSucc$: échéance de la tâche qui constitue le successeur de l'autre.

Si les tâches ne sont ni isolées ni reliées par une relation de précédence directe alors le regroupement est effectué sur l'échéance en suivant les conditions 8.3b.

Si aucun regroupement à coût nul n'a été réalisé sur des tâches isolées, alors la liste `fils` est parcourue. Pour chacun des quadruplets, le regroupement est effectué. Si après ajustement des échéances par la méthode `encodage`, l'ensemble est ordonnançable alors il est ajouté à la liste `filsordo` des ensembles ordonnançables après regroupement.

Similairement à l'algorithme de regroupement des tâches indépendantes, l'ensemble le plus prometteur est sélectionné d'après la fonction de coût *meilleurCoût*.

Algorithme 2 Algorithme de l'heuristique de regroupement de tâches avec contraintes de précédence

Fonction regroupementDep($\mathcal{G}$)

Entrée : $\mathcal{G} = (\mathbb{V}, \mathbb{E})$: graphe de tâches initial de tâches triées par ordre croissant des échéances

fils $\leftarrow \varnothing$

Pour $j = n - 1$ à 0 **Faire**
  Pour $i = j - 1$ à 0 **Faire**
    Si regroupable(τ_i, τ_j) **Alors**
      Si isolées(τ_i, τ_j) **Alors**
        /*Tentative de regroupement à coût nul*/
        Si $(C_i + C_j \leq D_j) \wedge ((D_j - C_j \leq D_i) \vee (R_j - C_j \leq D_i))$ **Alors**
          $\mathcal{G}' \leftarrow \mathcal{G} \setminus \{\tau_i, \tau_j\} \cup \{\tau_{ij}\}$
          Retourne regroupement($\mathcal{G}'$)
        Sinon
          fils $\leftarrow$ fils $\cup \{(\mathcal{G}, i, j, DlMin)\}$
        Fin Si
      Sinon Si précédenceDirecte(τ_i, τ_j) **Alors**
        Si $D_{pred} > D_{succ}$ **Alors**
          fils $\leftarrow$ fils $\cup \{(\mathcal{G}, i, j, DlMin)\}$
        Sinon
          Si $(D_j - C_j \leq D_i) \vee (R_j - C_j \leq D_i)$ **Alors**
            fils $\leftarrow$ fils $\cup \{(\mathcal{G}, i, j, DlSucc)\}$
          Sinon
            fils $\leftarrow$ fils $\cup \{(\mathcal{G}, i, j, DlPred)\}$
          Fin Si
        Fin Si
      Sinon
        Si $(D_j - C_j \leq D_i) \vee (R_j - C_j \leq D_i)$ **Alors**
          fils $\leftarrow$ fils $\cup \{(\mathcal{G}, i, j, DlMax)\}$
        Sinon
          fils $\leftarrow$ fils $\cup \{(\mathcal{G}, i, j, DlMin)\}$
        Fin Si
      Fin Si
    Fin Si
  Fin Pour
Fin Pour

filsOrdo $\leftarrow \varnothing$

Pour Tout $(\mathcal{M}, x, y, Dl) \in$ fils **Faire**
  $\mathcal{M}' \leftarrow \mathcal{M} \setminus \{\tau_x, \tau_y\} \cup \{\tau_{xy}\}$ /* sous l'échéance Dl*/
  $\mathcal{M}'' \leftarrow encodage(\mathcal{M}')$
  Si ordonnançable($\mathcal{M}''$) **Alors**
    filsOrdo $\leftarrow$ filsOrdo $\cup \mathcal{M}'$
  Fin Si
Fin Pour
Si filsOrdo $\neq \varnothing$ **Alors**
  Retourne regroupementDep(meilleurCoût(filsOrdo)) /*Poursuite avec le fils le plus prometteur*/
Sinon
  Retourne $\mathcal{G}$
Fin Si

Chapitre 10

Expérimentations

Dans ce chapitre, nous expérimentons le regroupement de tâches sous contraintes de précédence. Nous présentons tout d'abord plusieurs techniques de génération de graphes. Puis, nous évaluons l'impact des paramètres de génération de ces techniques sur les performances du regroupement de tâches.

10.1 Génération des contraintes de précédence

Générer des contraintes de précédence entre les tâches revient à générer des graphes aléatoires. L'outil *TFGG* [42] permet de générer des graphes de tâches aléatoires multi-périodiques. Il n'est pas possible de contrôler le nombre exact de tâches par graphe, la période exacte par graphe. *TFGG* permet de spécifier des échéances uniquement sur les tâches sans successeur mais pas de pire temps d'exécution par tâche. Dans sa thèse, Ndoye [78] décrit une procédure de génération de tâches dépendantes par niveau qui se rapproche de celle que nous proposons ci-après. À notre connaissance, il n'y a pas d'autres travaux ou d'outil qui traitent de la combinaison de techniques classiques de génération de graphes de tâches [33] et de techniques de génération de tâches sous contraintes temps réel, applicables dans l'état à notre modèle. Par conséquent, nous adaptons plusieurs méthodes de génération de graphes classiques à notre modèle de graphe acyclique dirigé, dans lequel chaque sommet correspond à une tâche.

10.1.1 Méthode d'Erdös-Rényi suivant le modèle $\mathcal{G}(n, p)$

Globalement, la technique consiste à générer des contraintes de précédence aléatoirement (c'est-à-dire des arcs) sur un ensemble de tâches indépendantes (c'est-à-dire des sommets).

En premier lieu, nous appliquons la méthode de génération de tâches indépendantes selon la procédure décrite en section 4.2. Nous devons ensuite veiller à générer des contraintes de précédences en tenant compte des échéances. Il est en effet peu probable dans un système réel que l'échéance d'un prédécesseur soit plus grande que celle de son successeur. Dans un système ordonnançable, s'il existe une contrainte de précédence entre une tâche τ_i et son successeur τ_j, alors τ_i doit nécessairement terminer son exécution avant le réveil de τ_j. Irrémédiablement, c'est l'échéance de τ_j qui dictera l'exécution de τ_i dans l'ordonnancement produit. Par conséquent, afin construire des contraintes de précédence cohérentes avec les échéances, nous générons les contraintes de précédence sur un ensemble trié par échéances croissantes, à partir du modèle $\mathcal{G}(n, p)$ d'Erdös-Rényi [46] précisé ci-après. En outre, nos expérimentations montrent que lorsque les contraintes de précédence ne sont pas générées suivant l'ordre croissant des échéances, il est difficile d'obtenir un ensemble initial ordonnançable.

Soit un graphe à n sommets (tâches) et une probabilité p telle que $0 \leq p \leq 1$, nous choisissons les arcs comme il suit. Pour toutes les paires de sommets u et v, il y a une probabilité p qu'il existe

un arc entre u et v. Cette manière de générer n'assure pas de propriété particulière sur le graphe, excepté que plus p est grand, plus le nombre d'arcs dans le graphe augmente. Rappelons que nous ne générons que des contraintes de précédence entre tâches de même période. La densité des contraintes de précédence correspond donc à la probabilité qu'il existe un arc entre deux tâches de même période. De manière à ne former aucun cycle, il n'existe de contrainte de précédence entre deux tâches τ_i et τ_j que si l'indice i est strictement inférieur à l'indice j, dans la liste des tâches. La matrice d'adjacence représentative est une matrice triangulaire stricte (supérieure ou inférieure selon le sens de parcours).

Suivant cette méthode, il est possible d'obtenir des tâches dites isolées, c'est-à-dire des graphes non connexes pour une période donnée.

10.1.2 Méthode fan-in fan-out

L'algorithme *fan-in fan-out* [42] est utilisé comme technique de base de génération de graphes dans TGFF. Il permet de contrôler le nombre maximal d'arcs entrants (degré entrant), le nombre maximal d'arcs sortants (degré sortant) du graphe et le nombre approximatif de sommets du graphe. Il alterne les phases de génération d'arcs entrants (« fan-in ») et sortants (« fan-out »). Chaque phase engendre la création de sommets de manière à équilibrer le nombre des arcs des sommets existants. Le graphe produit est nécessairement connexe, il ne peut y avoir de tâche isolée.

Nous concevons un graphe par période de cette manière et nous assignons les échéances suivant le sens de création des sommets. Celui-ci suit de manière assez proche l'ordre topologique du graphe.

10.1.3 Méthode de génération par niveau

La méthode d'Erdös-Rényi a l'inconvénient de potentiellement générer des arcs redondants dans le graphe. Soit trois tâches, τ_i, τ_j et τ_k, il est probable qu'elle produise trois arcs, le premier de τ_i vers τ_j, le deuxième de τ_j vers τ_k ainsi que le troisième arc entre τ_i et τ_k. Nous soulignions dans le chapitre 8 (remarque 8.2.3) que ces arcs redondants ne sont pas considérés dans le regroupement. La technique de génération de tâches par niveau a pour objectif d'éviter toute création d'arcs redondants dans le graphe. Elle consiste pour chacune des périodes, à générer aléatoirement i niveaux de n_i tâches par niveau de manière à ce que la somme des n_i soit égal à n, le nombre de tâches de la période concernée. Autrement, le nombre de niveaux peut être fixé proportionnellement au nombre de tâches de la période concernée. Ensuite, pour toutes les paires de sommets u et v appartenant respectivement aux niveaux i et $i+1$, il existe une probabilité p avec $0 \leq p \leq 1$ qu'un arc relie u à v. De cette façon, il ne peut pas exister d'arc redondant dans le graphe puisque chacune des tâches ne peut être rattachée qu'avec des tâches du niveau suivant. Les échéances des tâches d'une même période sont produites de telle sorte que l'échéance d'une tâche de niveau i soit inférieure ou égale à celle de toutes les tâches de niveau $i+1$. Ainsi, pour une même probabilité p, le nombre d'arcs générés est globalement moins élevé avec cette méthode qu'avec celle d'Erdös-Rényi, car les tâches de plus d'un niveau de différence ne sont pas sujettes à être reliées. Similairement à la technique d'Erdös-Rényi, il est possible que certaines tâches soient isolées. La figure 10.1 illustre cette technique de génération d'un graphe pour 4 niveaux.

Il est à noter que le nombre de niveaux (exprimé en pourcentage du nombre de tâches) ne correspond pas à la profondeur (ou hauteur) exacte du graphe, mais à sa valeur maximale. La profondeur du graphe dépend également des contraintes de précédence générées. Dans la figure 10.1, la profondeur du graphe est de taille 4 ce qui correspond au nombre maximal de niveaux sélectionné. Toutefois, si les tâches τ_h et τ_k n'étaient pas liées par une contrainte de précédence, le graphe aurait une profondeur de taille 3.

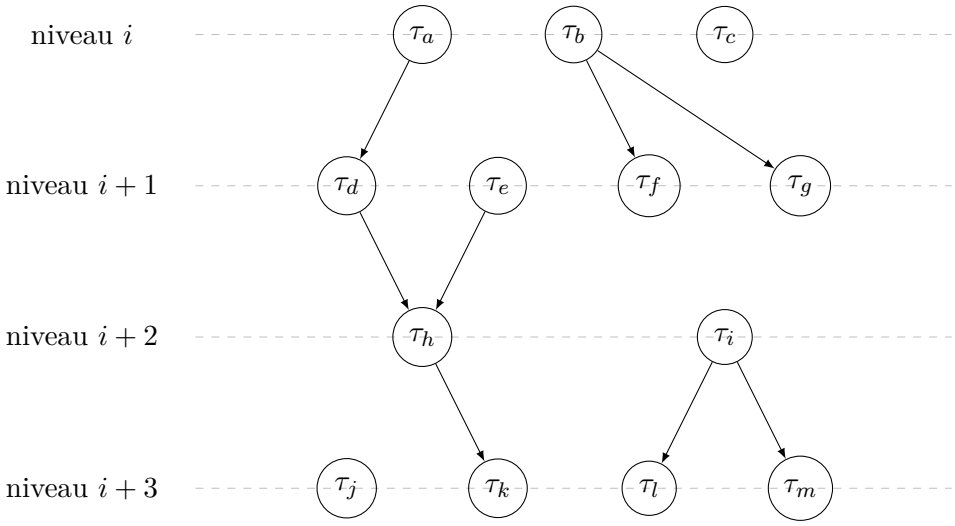


FIG. 10.1 : Exemple de génération d'un graphe de tâches par niveau

10.2 Variation des paramètres de génération du graphe

10.2.1 A partir du modèle d'Erdös-Rényi

Dans la figure 10.2, nous exposons l'effet de la densité des contraintes de précédence en abscisse sur le nombre de tâches après regroupement en ordonnée, sous DM et EDF. Le taux d'utilisation et les contraintes de précédence sont tirés aléatoirement. La densité des contraintes de précédence varie de 0.1 à 0.8. Le nombre de tâches est limité à 200 et la fonction de coût choisie minimise la densité. Nous observons que sous DM, le nombre de tâches s'accroît légèrement avec l'augmentation des contraintes de précédence. En EDF, il ne varie quasiment pas. Ces résultats tendent à montrer que la densité des contraintes de précédence n'a qu'un impact mineur sur l'efficacité du regroupement. En réalité, l'augmentation de la densité des contraintes de précédence dans les ensembles générés grandit le risque d'obtenir des arcs redondants. Par exemple, supposons qu'il existe des arcs $(u, v), (v, w), (w, x)$ et (u, x). L'arc (u, x) est redondant puisqu'il existe déjà un chemin allant de u vers x. Or, parmi les tâches dépendantes, nous ne regroupons que les tâches reliées par une précédence directe et nous ne considérons pas ces arcs redondants. Nous avons comparé le nombre de contraintes de précédence restantes, après réduction transitive, de deux graphes de densité de contraintes de précédence de 0.9 et de 0.1. La réduction transitive supprime les arcs redondants évoqués. Il s'avère que lorsque la probabilité initiale de densité des arcs est très haute, le nombre d'arcs restants n'est que très légèrement supérieur à celui d'un graphe généré avec une probabilité basse. La méthode d'Erdös-Rényi produit donc un nombre important d'arcs redondants. De manière générale, il n'y aucune garantie que la densité des contraintes de précédence après réduction transitive corresponde à la densité initiale désignée. Par conséquent, cette méthode ne permet pas de mettre en valeur l'impact de la densité des contraintes de précédence sur le regroupement de tâches.

10.2.2 Méthode fan-in fan-out

La méthode fan-in fan-out permet d'éviter les arcs redondants en fixant le paramètre du degré maximal entrant à 1. Néanmoins, cela a pour effet de limiter les graphes à des arbres et de n'autoriser que la variation du degré sortant maximal. Nos expérimentations montrent que si nous augmentons le degré maximal sortant, alors le nombre total de contraintes de précédence augmente. Aussi, le nombre de tâches de l'ensemble initial augmente alors linéairement avec le nombre de contraintes de précédence, ce qui permet difficilement d'évaluer l'effet du nombre de ces contraintes sur le regroupement. C'est la raison pour laquelle nous n'avons pas approfondi

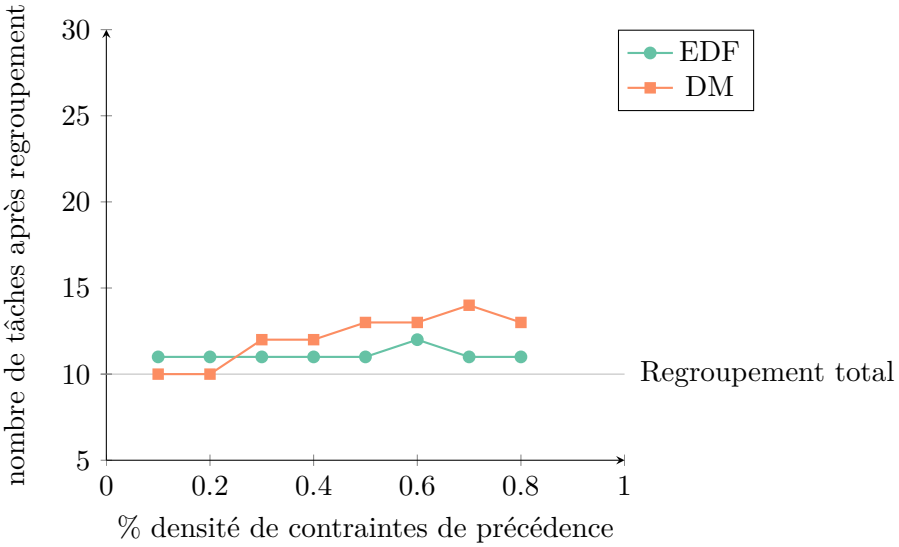


Fig. 10.2 : Impact de la densité des contraintes de précédence sur la diminution du nombre de tâches (n = 200, u aléatoire, $d_{min} = 0$, $d_{max} = 1$) à partir du modèle d'Erdös-Rényi

les expérimentations basées sur la méthode de génération fan-in fan-out.

10.2.3 Génération de tâches par niveau

La méthode de génération par niveau présente l'avantage de pouvoir contrôler le nombre de tâches en entrée. La densité des contraintes de précédence est également paramétrable en faisant variant conjointement le nombre de niveaux et la probabilité de présence des arcs, tout en évitant la présence d'arcs redondants. Intuitivement, pour une probabilité p d'avoir une contrainte de précédence liant les tâches de niveaux i et $i+1$ (niveaux adjacents), plus le nombre de niveaux est réduit, plus le nombre total de contraintes de précédence est élevé. Cette hypothèse est confirmée en pratique par la figure 10.3. Dans celle-ci, l'axe des abscisses représente le nombre de niveaux (en pourcentage du nombre de tâches de même période), l'axe des ordonnées la probabilité p et l'axe des cotes, le nombre total de contraintes de précédence. Le nombre de tâches de l'ensemble est à 200 et le nombre de périodes différentes à 10. Nous observons que le nombre de contraintes de précédence explose lorsque les valeurs en abscisse et en ordonnée sont poussées de concert vers leurs extremums.

Si le nombre de niveaux influe sur la densité des contraintes de précédences, il permet également de contrôler l'allure générale du graphe de chaque période. Choisir un nombre de niveaux faible par rapport au nombre de tâches assure d'obtenir un graphe aplati. Réciproquement, si le nombre de niveaux est élevé, les composantes du graphe prendront la forme de chaîne(s). Si l'on souhaite générer un graphe profond, il est nécessaire de choisir un nombre de niveaux élevé et une probabilité de contraintes de précédence élevée. En effet, si le nombre de niveaux est haut mais que la probabilité des contraintes de précédence est basse, le graphe ne contiendra que de multiples courtes chaînes.

Variation de la probabilité des contraintes de précédence

Nous avons fait varier la probabilité des contraintes de précédence avec un nombre de niveaux fixé à 90% du nombre de tâches par période. Au maximum, la profondeur du graphe est donc à 90% du nombre de tâches de la période concernée. Choisir un nombre maximal de niveaux élevé permet de diminuer l'espace de recherche puisqu'à chaque pas, l'heuristique a moins de candidats au regroupement. Les résultats sont illustrés dans la figure 10.4. Afin de mettre en valeur l'impact des contraintes de précédence, nous avons sélectionné des contraintes de temps moins favorables

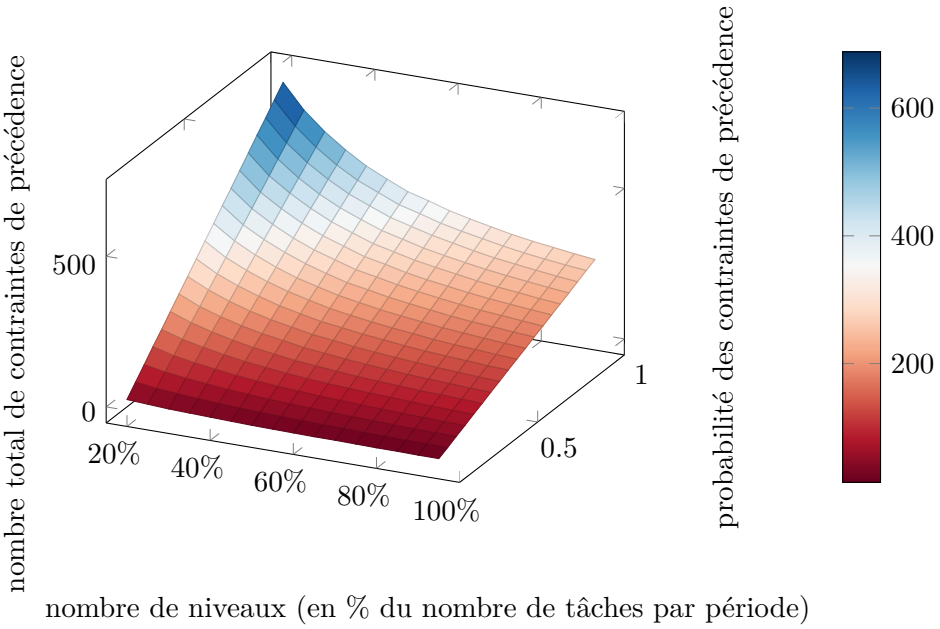


FIG. 10.3 : Impact du nombre de niveaux et de la probabilité des contraintes de précédence entre niveaux adjacents sur le nombre total de contraintes de précédence dans la génération par niveau

que si elles étaient générées aléatoirement. Dans ces conditions, il est plus difficile de minimiser le nombre de tâches et l'impact des contraintes de précédence se fait ressentir davantage. Pour EDF et DM, nous avons fixé le taux d'utilisation à 0.7 et la borne supérieure de génération des échéances à 0.8. Nous avons déterminé ces paramètres de manière empirique. Pour chaque probabilité des contraintes de précédence, nous évaluons le nombre de tâches sur 100 échantillons initialement ordonnançables de 200 tâches. Nous observons que le regroupement de tâches est de moins en moins efficace à mesure que la probabilité, et donc le nombre total de contraintes de précédence, augmente.

L'objectif de ces premières expérimentations était de fixer le nombre de niveaux maximal afin d'évaluer uniquement l'impact de la probabilité des contraintes de précédence sur le regroupement de tâches. Il est insuffisant de ne mesurer que l'influence de la probabilité des contraintes de précédence. De plus, le nombre maximal de niveaux choisi correspond à une topologie de graphe singulière. Dans la suite, nous faisons varier conjointement la probabilité des contraintes de précédence avec le nombre maximal de niveaux, dans des paramètres plus classiques.

Variation du nombre de niveaux et de la probabilité des contraintes de précédence

Les graphes de la littérature que nous avons observés, correspondent à un nombre maximal de niveaux environ égal à 50% du nombre total de tâches, pour une probabilité de contraintes de précédences d'environ 0.25. Dans les tableaux 10.1 et 10.2, nous indiquons les résultats de regroupement obtenus en faisant varier la profondeur maximale du graphe et la probabilité des contraintes de précédence, sous DM et EDF. Chacune des colonnes correspond à une probabilité de contrainte de précédence et chaque ligne, au nombre de niveaux en pourcentage du nombre de tâches. Remarquons qu'il est possible d'obtenir des résultats uniques en temps acceptable pour un nombre de niveaux maximal à moins de 40%. Néanmoins, sa complexité rend difficile la génération d'un nombre suffisant d'échantillons pour en tirer des conclusions statistiques. C'est la raison pour laquelle nous avons borné inférieurement le nombre maximal de niveaux à 40% et supérieurement, la probabilité de contraintes de précédence à 0.75 (0.5 pour EDF). Les paramètres de génération des tâches sont les mêmes que dans la section précédente, pour 100

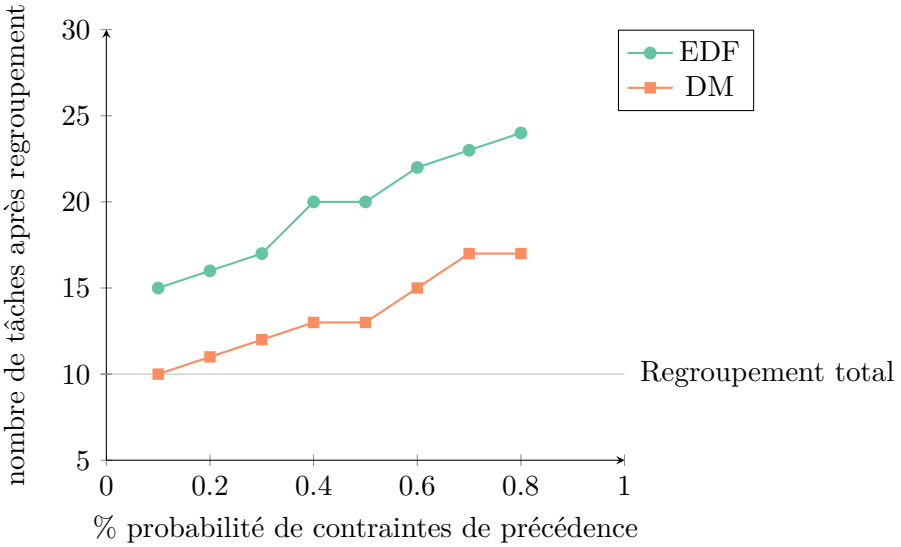


Fig. 10.4 : Impact de la probabilité des contraintes de précédence sur la diminution du nombre de tâches (n = 200, u = 0.7, $d_{min} = 0$, $d_{max} = 0.8$) à partir de la génération par niveau à 90%

	0.25	0.50	0.75
0.4	12	15	18
0.5	11	15	18
0.6	11	15	18
0.7	11	14	18

Tab. 10.1 : Variation du nombre de niveaux et de la probabilité des contraintes de précédence en DM à partir de la génération par niveau

	0.25	0.50
0.4	23	23
0.5	24	24
0.6	24	26
0.7	24	25

Tab. 10.2 : Variation du nombre de niveaux et de la probabilité des contraintes de précédence en EDF à partir de la génération par niveau

exécutions par point.

Nous observons dans le tableau 10.1 relatif à DM que dans les bornes observées, le nombre de tâches après regroupement augmente très légèrement du point de vue du nombre de niveaux, pour une même probabilité. De manière générale, c'est la probabilité des contraintes de précédence qui a le plus d'impact sur le regroupement. Pour un même niveau, la difficulté du regroupement va de pair avec la probabilité de contraintes de précédence générées. Dans les paramètres observés, la baisse du nombre de niveaux n'a que très peu d'effet sur le nombre total de contraintes de précédence générées (cf. 10.3) en comparaison avec avec la hausse de la probabilité. Par conséquent, il apparaît ici que dans des paramètres classiques, la probabilité des contraintes de précédence à plus d'incidence que le nombre de niveaux sur la qualité du regroupement.

En EDF, la tendance est moins claire, le nombre de tâches augmentant à peine avec le hausse de la probabilité des contraintes de précédence, pour les niveaux observés. Nos observations dans le détail montrent que les résultats sous EDF sont très irréguliers en comparaison avec ceux obtenus pour DM. Comme à l'accoutumée, le nombre de tâches obtenues après regroupement est plus élevé sous EDF que DM. Cet écart est renforcé par la non-utilisation de l'optimisation basée sur le calcul temps de réponse, afin de pratiquer les expérimentations sur un plus grand nombre d'échantillons.

Remarques sur l'efficacité de l'heuristique en temps d'exécution

L'heuristique adapté au cas des tâches dépendantes et moins efficace en temps que pour des ensembles de tâches indépendantes. Le regroupement peut alors prendre jusqu'à plusieurs dizaines de minutes de calcul. Ce ralentissement est principalement dû à la présence des contraintes de précédence qui diminue le nombre de regroupements à coût nul applicables. Il est alors nécessaire d'exécuter plus fréquemment les tests d'ordonnançabilité qui peuvent se révéler coûteux.

Le nombre de niveaux par période impacte la forme que prendra le graphe. Les expérimentations ont montré que le nombre de niveaux n'avaient pas d'effet majeur sur les performances du regroupement en nombre de tâches. Néanmoins, il influence les performances de l'heuristique en temps d'exécution. En effet, si le nombre de niveaux est bas, le graphe est très aplati et de ce fait, à chaque étape, l'algorithme a un nombre important de candidats au regroupement possibles. Si au contraire le nombre de niveaux est élevé, les graphes prennent la forme de chaînes. L'algorithme peut alors avancer plus rapidement dans l'espace des solutions, car il a peu de successeurs directs. Avec des bornes d'échéances peu favorables, nos expérimentations ont confirmé ces dires puisque l'algorithme peut mettre 2 à 3 fois plus de temps d'exécution, lorsque le nombre de niveaux est bas.

Conclusion de la troisième partie

Dans cette troisième partie, il a été question de considérer des contraintes de précédence entre les tâches. La prise en compte des dépendances se fait à deux niveaux. Tout d'abord, elle implique d'étudier en quoi consiste désormais un regroupement dans un graphe et son impact sur l'ordonnançabilité du système. À ce sujet, il est notamment apparu qu'il n'était plus possible de pratiquer des regroupements à coût nul comme en deuxième partie. Ensuite, elle suppose d'utiliser des méthodes qui permettent de vérifier l'ordonnançabilité d'un ensemble de tâches dépendantes. Nous avons sélectionné dans ce travail la méthode de Chetto et al. qui transforme un ensemble de tâches dépendantes en un ensemble de tâches indépendantes, par ajustement des contraintes de temps. Ainsi, il est possible d'utiliser les algorithmes d'ordonnancement monoprocesseur. À travers nos expérimentations, nous avons montré que le regroupement de tâches était sensible non plus uniquement aux contraintes de temps de l'ensemble de départ, mais aussi à la topologie des graphes. En particulier, la probabilité de génération des contraintes de précédences ainsi que la profondeur maximale des graphes influent sur l'efficacité du regroupement en nombre de tâches, mais aussi en termes de temps d'exécution. Dans la prochaine et dernière partie, nous nous intéressons à la problématique du regroupement de tâches sur les architectures multiprocesseurs.

Quatrième partie

Regroupement de tâches avec contraintes de précédence en contexte multiprocesseur

Chapitre 11

Ordonnancement Multiprocesseur

Dans ce chapitre, nous introduisons les architectures multiprocesseurs. Préalablement abordées sous l'angle du matériel, nous étudions ensuite les aspects liés à l'ordonnancement de tâches sur ce type d'architecture. Ce chapitre n'a pas vocation à présenter de manière exhaustive l'ordonnancement multiprocesseur mais vise à préciser les notions requises pour traiter le problème du regroupement de tâches. Pour une vue d'ensemble plus complète du sujet, le lecteur pourra se référer aux travaux [37, 30, 53].

11.1 Architecture multiprocesseur

Dans les parties précédentes, nous nous sommes penchés sur le regroupement de tâches sur des architectures monoprocesseurs. La complexité grandissante des applications ainsi que les limites de puissance des transistors CMOS [63] ont favorisé l'émergence d'une nouvelle classe d'architecture parallèle, dotée de plusieurs unités de calcul pour l'exécution simultanée d'un ensemble de tâches : le multiprocesseur.

Nous parlons de multicœurs lorsque plusieurs unités de calcul sont sur la même puce ou autrement de multiprocesseur. Dans la littérature relative à l'ordonnancement temps réel, c'est principalement le terme multiprocesseur qui est utilisé pour ces deux catégories, nous en ferons de même. L'important à notre niveau est de savoir qu'il est possible d'exécuter plusieurs tâches en parallèle sur ce type d'architecture.

Il existe différentes compositions de plateformes multiprocesseurs :

- Les processeurs sont **identiques**. Ils fonctionnent tous à la même vitesse et ont les mêmes caractéristiques.
- Les processeurs sont **homogènes** ou uniformes. Ils ont les mêmes propriétés, et à une vitesse identique, ils calculent un traitement similaire avec le même nombre d'instructions. Néanmoins, ils peuvent fonctionner simultanément à des vitesses différentes.
- Les processeurs sont **hétérogènes**. Ces processeurs ne sont pas comparables. Ils sont de natures différentes, peuvent fonctionner à des vitesses différentes et il est probable qu'à une vitesse identique, le nombre d'instructions pour effectuer un traitement similaire varie. Par exemple, sur des plateformes mobiles, nous distinguerons le processeur central (CPU pour Central Processing Unit en anglais) du processeur graphique (GPU pour graphical Processing Unit en anglais) spécialisé dans le traitement image ou vidéo ou encore du processeur de signal numérique (DSP pour Digital Signal Processing en anglais), dédié au traitement numérique des signaux.

Dans ce travail, nous nous restreignons aux processeurs identiques. Au-delà de l'aspect matériel des architectures multiprocesseurs, nous nous intéressons à la manière de regrouper des

ensembles de tâches sur ce type de plateforme. La problématique d'ordonnancement est plus complexe que dans le cas monoprocesseur. Il est certes possible d'ordonnancer des ensembles de tâches aux taux d'utilisation plus conséquents, mais il se pose alors un nouveau problème d'allocation spatiale. Pour ce faire, nous dégageons trois stratégies d'ordonnancement :

- La stratégie ou l'ordonnancement **global** consiste à allouer dynamiquement des tâches sur des processeurs lorsqu'elles sont prêtes à être exécutées.
- La stratégie ou l'ordonnancement **partitionné** implique une phase d'allocation statique des tâches sur les processeurs. Les tâches sont ensuite ordonnancées dans chacun d'eux au moyen des algorithmes d'ordonnancement monoprocesseur.
- La stratégie ou l'ordonnancement **semi-partitionné**. Cette stratégie est une approche hybride des stratégies globale et partitionnée. Les tâches sont allouées statiquement sur les processeurs mais certaines sont éventuellement déplacées sur d'autres pendant l'exécution.

Dans la section suivante, nous présentons tout d'abord la stratégie globale. Nous examinons dans un second temps la stratégie partitionnée puisque c'est celle que nous utiliserons pour traiter le regroupement de tâches dans le cas multiprocesseur. Pour cette dernière, nous passerons également en revue l'ordonnancement partitionné de tâches avec contraintes de précédence.

11.2 Ordonnancement multiprocesseur

Cette section a pour objectif d'expliquer les principes des stratégies globale et partitionnée de l'ordonnancement multiprocesseur. Nous ne détaillerons pas d'algorithme d'ordonnancement multiprocesseur de la littérature. Pour un panorama de tels méthodes, le lecteur pourra se rapporter à l'étude de Davis et Burns [37]. Leur travail se restreint à la classe des processeurs identiques, ce qui correspond au cas traité dans cette partie.

11.2.1 Ordonnancement global

À la manière des algorithmes de l'ordonnancement monoprocesseur, l'ordonnanceur suivant la stratégie globale dispose d'un ensemble de tâches prêtes à l'exécution et choisit d'exécuter la plus prioritaire de celles-ci, sur le processeur de son choix. Ainsi, il est possible qu'une instance de tâche soit exécutée sur un processeur et que l'instance suivante de la même tâche soit exécutée sur un autre. Il s'agit alors d'une *migration* de tâche. La migration n'est pas restreinte, c'est-à-dire qu'un travail préempté sur un processeur peut également terminer son exécution sur un autre. Généraliser les algorithmes d'ordonnancement monoprocesseur optimaux tels que EDF ou RM au cas multiprocesseur ne permet pas d'obtenir d'algorithmes optimaux [72]. Les algorithmes d'ordonnancement global « équitables » offrent une meilleure utilisation des ressources processeur. Nous évoquons ici RM et non pas DM, car dans le cas multiprocesseur, le modèle des échéances implicites ($D_i = T_i$) est souvent utilisé pour simplifier le problème d'ordonnancement.

11.2.2 Ordonnancement partitionné

La stratégie partitionnée consiste dans un premier temps à allouer l'ensemble des tâches sur m différents processeurs. Cette allocation est définitive et les tâches sont ensuite ordonnancées localement sur chacun des processeurs en utilisant des algorithmes d'ordonnancement monoprocesseur. La partie la plus délicate est donc de partitionner les tâches de manière à ce que le problème soit réduit à m ensembles de tâches ordonnançables.

Le problème de partitionnement se décline en une version décisionnelle et en une version d'optimisation. Dans la première, le nombre de processeurs est fixé et le problème revient à décider si l'ensemble de tâches est ordonnançable. Dans la seconde, on cherche à trouver le

nombre minimal de processeurs nécessaires pour l'ordonnancer. Dans les deux cas, le problème de partitionnement correspond au problème de bin-packing. Le problème consiste alors à ranger un nombre fini des tâches (les objets) de taille correspondant au taux d'utilisation de la tâche dans des processeurs (boîtes) de capacité limitée. Cette capacité correspond aux conditions d'ordonnançabilité monoprocesseur. Pour des tâches à échéances implicites, ces conditions sont basées sur le taux d'utilisation processeur. La limite est de 1 pour EDF (condition nécessaire et suffisante) et elle tend vers ln 2 pour RM lorsque le nombre tâches augmente (condition suffisante).

Le problème étant NP-difficile, il est dur d'obtenir des solutions optimales en temps raisonnable, en particulier pour des instances de grandes tailles. Bien que des solutions exactes existent pour des nombres limités d'objets, le problème est le plus souvent traité grâce à des heuristiques.

La première étape de l'heuristique consiste à effectuer un tri des tâches. Généralement, elles sont triées par taux d'utilisation décroissant. La seconde correspond à la manière d'allouer les tâches. Les tâches triées sont traitées successivement en fonction d'une « liste » de processeurs. Nous pouvons citer les méthodes d'allocation classiques suivantes :

- First-Fit : la tâche est allouée sur le premier processeur de la liste capable de la recevoir tout en restant ordonnançable, en partant du premier processeur ;
- Next-Fit : similaire au First-Fit sauf que le processeur de départ correspond au processeur suivant le dernier utilisé pour allocation ;
- Best-Fit : la tâche est allouée sur le processeur disposant de la plus petite capacité disponible. La technique vise le meilleur choix d'allocation dans un objectif de minimisation du nombre de processeurs ;
- Worst-Fit : la tâche est allouée au processeur disposant de la plus grande capacité disponible.

Par défaut, la liste des processeurs n'est pas triée. Néanmoins, en pratique, la liste des processeurs est maintenue triée par capacité pour les politiques Best-Fit et Worst-Fit afin de réaliser un parcours plus efficace.

La méthode de partitionnement consiste donc en un tri, suivi d'une allocation puis d'une politique d'ordonnancement monoprocesseur.

11.2.3 Regroupement dans le cadre de l'ordonnancement global ou partitionné ?

Les stratégies globales et partitionnées ne sont pas comparables. Dans la classe des ordonnancements à priorité fixe par tâche, Leung et Whitehead [71] ont établi que certains ensembles ordonnançables par la stratégie globale, ne pouvaient pas l'être par la stratégie partitionnée et réciproquement. En 2007, Baruah [12] a prouvé l'équivalent pour les tâches à priorité fixe par travail.

Intuitivement, appliquer le regroupement de tâches pour un ordonnancement global amène plusieurs difficultés. En monoprocesseur, les tests exacts RTA et PDA sont viables vis-à-vis des échéances. Nous avons utilisé le principe de viabilité de l'ordonnancement pour limiter l'espace de recherche puisqu'un ensemble jugé non ordonnançable ne peut pas le devenir, quel que soit le regroupement supplémentaire effectué. Dans le cas multiprocesseur, des anomalies peuvent apparaître, c'est-à-dire que le comportement de certains algorithmes d'ordonnancement n'est pas intuitif lorsque certains paramètres sont modifiés « positivement ». Par exemple, au contraire de l'algorithme Global EDF (priorité fixe par travail), l'algorithme Global FP (priorité fixe par tâche) est viable lorsqu'il est sujet à des accroissements des échéances [8]. Cela signifie qu'il est possible qu'un ensemble jugé non-ordonnançable devienne ordonnançable lorsque l'on diminue des échéances. Au regard de notre heuristique de regroupement monoprocesseur, un ensemble

non ordonnançable pourrait alors produire des ensembles ordonnançables, ce qui compliquerait la recherche de solutions.

Dans notre travail, nous nous intéressons au regroupement de tâches dépendantes en multiprocesseur selon la stratégie partitionnée. La raison de ce choix réside dans la possibilité d'appliquer les techniques de regroupement de tâches en monoprocesseur, une fois le partitionnement réalisé.

11.3 Ordonnancement partitionné pour tâches avec contraintes de précédence

L'ordonnancement de tâches en ordonnancement partitionné requiert deux étapes. La première consiste à distribuer les tâches sur les différents processeurs. La seconde a pour but d'ordonnancer ces tâches sur chacun des processeurs.

11.3.1 Partitionnement avec contraintes de précédence

Les références évoquées plus haut ciblent des systèmes sans contrainte de précédence. Sans évoquer encore l'ordonnançabilité d'un système avec de telles contraintes, il est nécessaire d'opter pour une méthode de partitionnement qui prenne en compte les contraintes de précédence. En effet, il peut exister des contraintes de précédence entre des tâches placées sur des processeurs différents qui contraignent le système de manière importante. Il est alors peu probable qu'une méthode de partitionnement se basant uniquement sur les taux d'utilisation permettent d'obtenir, in fine, un ensemble de tâches ordonnançable.

Dans [85], Peng et Shin présentent deux algorithmes de type *séparation et évaluation* pour partitionner des tâches dépendantes sur un système distribué. Les méthodes sont exactes et ne passent pas à l'échelle pour les ensembles de taille comparable à ceux que nous visons. Afin de diminuer la taille de l'espace de recherche, Abdelzaher et Shin [1] proposent de regrouper des tâches ensembles puis d'assigner récursivement ces groupes de tâches sur des processeurs hétérogènes. Toujours dans le contexte des systèmes distribués, Ramamritham [86] proposait plus tôt une heuristique qui vise à allouer et à ordonnancer statiquement des tâches sur différents processeurs.

Nous présentons plus en détail les heuristiques de partitionnement de tâches sous contraintes de précédence de Buttazo et al. dans [26] que nous avons adaptées dans ce travail. À l'origine, la technique vise à allouer des tâches sous contraintes de précédence sur différents processeurs virtuels d'une plateforme multicœur. Une réservation de ressource processeur (*ressource reservation*) est associée à chacun des processeurs virtuels. Celle-ci est caractérisée par un couple (α, Δ) [75] où la bande passante α correspond à un taux d'utilisation disponible et Δ au pire temps d'attente pour accéder à cette ressource. Les auteurs distribuent les tâches sur les processeurs virtuels en cherchant à optimiser la bande passante utilisée tout en garantissant l'ordonnançabilité. Plus généralement, ce travail décrit une méthode de partitionnement des tâches sous contraintes de précédence vers un ensemble d'unités appelées *flux*, assimilables à des processeurs présentant chacun un taux d'utilisation inférieur ou égal à un. Ce partitionnement est guidé par un objectif de minimisation des ressources, soit du point de vue de la bande passante, soit du nombre de processeurs utilisés. Nous nous basons sur une partie de leur travail dans le but de minimiser le nombre de processeurs en assurant l'ordonnançabilité. Nous décrivons l'adaptation de la technique dans l'algorithme 3. Elle repose sur la notion de *chemin critique*.

Définition 21 (Chemin critique)
Le chemin critique correspond à la chaîne de tâches dont la somme des pires temps d'exécution est la plus grande. Elle constitue par la même la durée minimale nécessaire pour exécuter le graphe de tâches, quel que soit le nombre d'unités de calcul disponibles.

Les tâches sont partitionnées dans des flux et chaque flux est destiné à être exécuté sur un cœur ou processeur différent. Le principe général des heuristiques est le suivant : Si m_{inf} est le nombre de cœurs minimal requis par l'application, l'algorithme contiendra m_{inf} flux. L'algorithme sélectionne le chemin critique du graphe de tâches avec contraintes de précédence et assigne toutes les tâches de celui-ci dans le premier flux. Il choisit à nouveau le chemin critique du graphe restant (où toutes les tâches assignées ont été retirées) et l'ajoute au flux existant si le flux résultant reste faisable. Le flux est considéré faisable si la condition de faisabilité (un test d'ordonnançabilité comme la PDA par exemple) est vérifiée pour ce flux. Si le flux résultant n'est pas faisable, l'algorithme construit un nouveau flux dans lequel les tâches du chemin critique sont assignées. Dans l'algorithme 3, la méthode `vaSinonNouveau` est responsable de l'ajout du chemin critique (et plus tard d'une tâche) à un flux existant, s'il est faisable, ou de la création d'un nouveau flux dans le cas contraire. La procédure est répétée jusqu'à ce que m_{inf} flux soient construits. Ensuite, toutes les tâches non encore assignées sont triées par pires temps d'exécution décroissants et ajoutées aux flux existants en utilisant une politique de « meilleur ajustement en premier » (*Best-Fit* en anglais). Le critère utilisé pour choisir le flux adéquat n'est pas précisé, mais il peut par exemple correspondre au taux d'utilisation processeur du flux. Si une tâche ne peut être assignée à aucun des flux existant, alors un nouveau flux est créé. Les heuristiques H1 et H2 proposées diffèrent par la valeur de m_{inf}. Pour H2, elle est fixée à 1 tandis que pour H1, les auteurs proposent de la calculer en fonction du taux d'utilisation processeur total et du nombre de tâches présentant un taux d'utilisation élevé.

Toutefois, le modèle de base de l'article diffère légèrement du nôtre puisqu'il considère un graphe de tâches monopériodique doté d'une seule échéance relative. Les dates de réveil ainsi que les échéances individuelles des tâches sont des paramètres libres qui sont fixés pour les besoins de l'ordonnancement. La définition 21 du chemin critique est valable pour les ensembles monopériodiques. Pour prendre en compte la fréquence d'activation des chemins, nous considérons le chemin critique C_c défini comme il suit :

$$C_c \stackrel{\text{def}}{=} \max \sum_{i \in \text{ chaîne } j} \left(\frac{C_i}{T_j} \right) \tag{11.1}$$

Le chemin critique représente la chaîne de charge maximale du graphe de tâches. Il nous a semblé raisonnable de la pondérer par la période d'activation de la chaîne.

Notons qu'une tâche isolée constitue une chaîne à un seul élément et que la période d'une chaîne est égale à celle de la ou les tâches qui la composent puisque les contraintes de précédence n'existent qu'entre tâches de même période.

11.3.2 Ordonnançabilité de tâches avec contraintes de précédence en partitionné

Dans ce travail, nous nous orientons vers un ordonnancement multiprocesseur partitionné. Le problème consiste ici à partitionner et à ordonnancer un ensemble de tâches sur différentes unités de calcul, de manière à ce que toutes les contraintes de temps et de précédence soient garanties. L'idée, nous l'avons vu, consiste donc à distribuer les tâches sur différentes unités de calcul puis à appliquer les principes de l'ordonnancement monoprocesseur sur chacun d'eux.

Dans le chapitre 5, nous expliquons comment nous traitons les relations de précédence qui peuvent exister entre les tâches. Dans le cas des tâches synchrones, nous assurons le respect de celles-ci par un ajustement des échéances. L'ajustement des échéances des tâches permet d'assurer, par le jeu des priorités, qu'une tâche ne pourra commencer à s'exécuter avant la date de

Algorithme 3 Algorithme de partitionnement adapté de Buttazo et al. [26]

```
Fonction partitionnement(G, M_inf)
  G' ← G
  F ← ∅
  /*Partitionne les M_inf chemins critiques*/
  Tant Que |F| ≤ M_inf Faire
    C_c ← cheminCritique(G')
    vaSinonNouveau(F, C_c)
    G' ← G' \ C_c
  Fin Tant Que

  /*Partitionne les tâches restantes*/
  Pour Tout τ_i ∈ G' Faire
    vaSinonNouveau(F, {τ_i})
  Fin Pour
Retourne F

  /*Cherche un flux compatible, en crée un nouveau sinon*/
Fonction vaSinonNouveau(F, C)
  Pour Tout Flux ∈ F Faire
    F' ← Flux ∪ C
    Si ordonnançable(F' ) Alors
      F ← F \ Flux ∪ F'
      Retourne  /*Chaîne ajoutée à un flux existant, sortie de la fonction*/
    Fin Si
  Fin Pour
F ← F ∪ Flux /*Chaîne ajoutée comme nouveau flux*/
```

fin d'exécution de ses prédécesseurs. L'ordonnancement des tâches est propre aux priorités assignées au sein de du processeur. En ordonnancement multiprocesseur partitionné, nous faisons désormais face à une nouvelle problématique. Comment assurer le respect d'une contrainte de précédence entre deux tâches qui se trouvent sur des unités de calculs différentes ?

Dans la suite de ce chapitre, nous examinons plusieurs méthodes pour y parvenir. Ces techniques se basent sur l'ajustement des dates de réveil et des échéances, de manière à assurer comme dans le cas des tâches asynchrones, qu'une tâche ne puisse se réveiller avant son successeur.

11.3.3 Ajustement des dates de réveil et des échéances

Le problème d'ajustement optimal des échéances et des dates de réveil en ordonnancement partitionné est un problème encore ouvert [9]. Buttazo et al. [26] modifient l'algorithme d'ajustement des échéances et des dates de réveil de Chetto et al. [32] (présenté en chapitre 5) pour prendre en compte la possibilité que des tâches s'exécutent en parallèle sur différentes unités de calcul. Suivant l'objectif de minimisation des communications, Wu et al. [107] présentent une formulation de type OLNE pour assigner les échéances et les dates de réveil aux tâches. Leur méthode considère les échéances intermédiaires des tâches comme des paramètres libres et les méthodes OLNE supportent mal le passage à l'échelle. C'est pourquoi nous utilisons et détaillons dans la suite la méthode de Buttazo et al. [26] qui adapte la méthode Chetto et al. [32] au cas multiprocesseur partitionné.

Adaptation de la méthode de Chetto et al. au cas partitionné

D'une part, l'ajustement des échéances assure qu'un prédécesseur ne puisse pas s'exécuter avant son successeur direct :

$$D_i^* = \min(D_i, \min_{\tau_j \in succs(\tau_i)}(D_j^* - C_j))$$

D'autre part, l'ajustement des dates de réveil garantit qu'un successeur ne puisse commencer son exécution avant celle son prédécesseur :

$$O_i^* = \max(O_i, \max_{\tau_j \in preds(\tau_i)}(O_j^* + C_j))$$

Ces équations d'encodage concernent le cas monoprocesseur. Il est donc implicite que τ_i et τ_j appartiennent au même processeur m_k. Cependant, dans le cas multiprocesseur où les tâches peuvent s'exécuter en parallèle, l'ajustement des échéances n'est pas suffisant. Il est également nécessaire qu'une tâche ne puisse s'activer avant la fin de l'exécution de son prédécesseur même si celui-ci est sur un processeur différent.

Soit $\tau_i \in m_k$ et $\tau_j \notin m_k$ avec $\tau_j \rightarrow \tau_i$, il est nécessaire que τ_i ne puisse commencer à s'exécuter avant la fin d'exécution de τ_j. La date de fin d'exécution de τ_j n'est pas connue, c'est donc sur son échéance que se base l'encodage proposé :

$$O_i^* = \max(O_i, \max_{\tau_j \in preds(\tau_i), \tau_j \notin m_k}(D_j^*))$$

Il est important de souligner que cette méthode d'assignation des dates de réveil est suffisante mais pas nécessaire [9, 107] (donc pessimiste). En effet, la date de réveil d'une tâche τ_i d'un processeur peut être assignée en fonction de l'échéance d'un de ses prédécesseurs τ_j qui se trouve sur un autre processeur. En conséquence, si τ_j termine son exécution beaucoup plus tôt que l'échéance de celle-ci, τ_i devra attendre la date correspond à l'échéance de τ_j pour commencer à s'exécuter.

Une première idée pourrait être de considérer le pire temps de réponse de la tâche prédécesseur plutôt que son échéance. Néanmoins, cette méthode entraînerait des modifications en cascade des temps de réponse des autres tâches ; ce qui la rapprocherait de l'analyse holistique [101] utilisée dans les systèmes distribués. L'analyse holistique vise à prendre en compte l'ordonnancement des tâches et des messages dans un système distribué hétérogène. L'analyse holistique considère les dépendances induites par l'échange de messages entre les tâches émettrices et réceptrices, situées sur des unités de calculs possiblement différentes. En particulier, elle tient compte du fait qu'un message ne peut pas commencer à s'exécuter avant la fin d'exécution de sa tâche émettrice et que, la tâche réceptrice ne peut démarrer son exécution avant la fin de l'exécution du message. Pour ce faire, elle modélise et calcule les délais (gigues ou *jitter* en anglais) sur les activations des instances des tâches et des messages. Les analyses de temps de réponse sont répétées, et les contraintes de temps réajustées, jusqu'à atteindre un point fixe. Appliquer l'analyse holistique en définissant les pires temps d'exécution des messages à 0 revient à spécifier des contraintes de précédence entre les tâches. La complexité de cette technique est assez élevée, nous n'avons pas considéré cette piste dans ce travail.

11.3.4 Ordonnancement monoprocesseur de tâches asynchrones

L'ajustement des dates de réveil impose désormais de considérer des ensembles de tâches asynchrones, c'est-à-dire des tâches dont la date de réveil n'est pas nécessairement fixée à 0. La figure 11.1 représente un ensemble de tâches asynchrones décrit dans la tableau 11.1. Dans cette section, nous présentons des résultats relatifs à ce type de systèmes.

En préambule, nous rappelons une propriété intéressante [14] des systèmes de tâches asynchrones :

	C_i	D_i	T_i	O_i
τ_1	2	7	7	1
τ_2	3	7	4	2
τ_3	1	15	15	0

Tab. 11.1 : Contraintes de temps d'un ensemble de tâches asynchrones

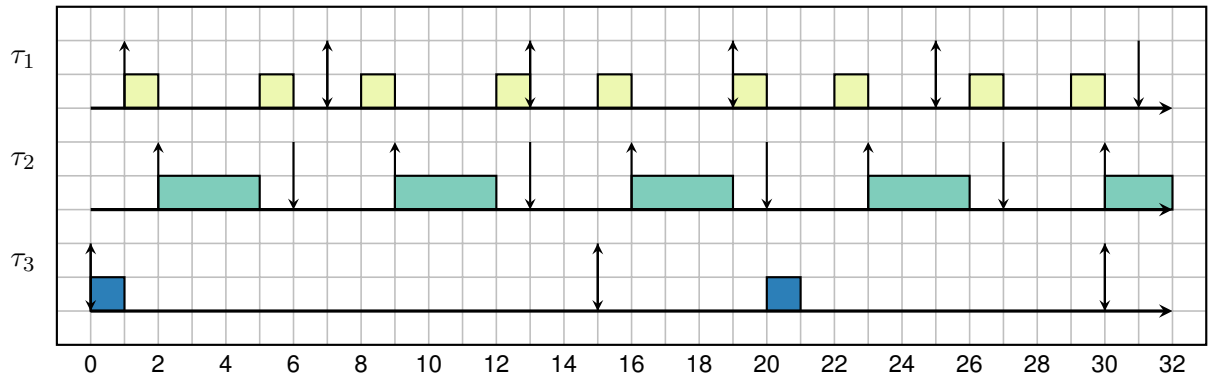


Fig. 11.1 : Diagramme d'exécution des tâches d'un ensemble asynchrone sous EDF

Théorème 9
Soit un système asynchrone $\mathcal{A}$ et un système $\mathcal{S}$, version du système $\mathcal{A}$ dans laquelle toutes les dates de réveil ont été fixées à 0 (donc synchrone). Si $\mathcal{S}$ est jugé faisable alors $\mathcal{A}$ est faisable. Dans le cas contraire, rien ne peut être affirmé.

Les tests exacts pour tâches synchrones deviennent donc des tests suffisants pour leurs équivalents asynchrones. Le problème de détermination d'un test nécessaire et suffisant pour les systèmes de tâches a été prouvé co-NP-complet au sens fort [70, 14].

Ordonnancement à priorité fixe par tâche

Deadline Monotonic n'est plus optimal lorsque l'on considère des tâches asynchrones. Une méthode d'assignation de priorités optimale a été mise au point par Audsley [5]. Bernat [17] a proposé une analyse de temps de réponse pour chaque instance de tâche asynchrone en ordonnancement à priorité fixe par tâche.

Ordonnancement à priorité fixe par travail

Dans l'ordonnancement de tâches asynchrones à priorité fixe par travail, EDF reste optimal. Pellizoni et Lipari jugent l'utilisation de tests exacts pour ensemble de tâches synchrones à l'endroit des ensembles de tâches asynchrones très pessimiste [83]. En conséquence, ils présentent un test pseudo-polynomial suffisant basé sur l'analyse des motifs d'exécutions critiques possibles. Une analyse de temps de réponse exacte pour une instance particulière d'une tâche pour EDF [41] existe, mais sa complexité est exponentielle.

Chapitre 12

Regroupement de tâches multiprocesseur

Dans ce chapitre, nous étudions la problématique du regroupement de tâches en contexte multiprocesseur. En particulier, nous examinons dans quelle mesure le problème peut être ramené à plusieurs problèmes monoprocesseurs.

12.1 Adaptation du problème monoprocesseur

Après partitionnement des tâches avec contraintes de précédence sur les différents processeurs, nous distinguons deux situations :

- Les processeurs sont indépendants. Il n'y a aucune contrainte de précédence existant entre des tâches situées sur des processeurs différents.
- Les processeurs sont interdépendants. Il existe au moins une contrainte de précédence entre deux tâches placées sur des processeurs différents. Notons que le regroupement dans la première situation est un cas particulier du regroupement dans la seconde.

Lorsque les processeurs sont indépendants, le regroupement de tâches multiprocesseur revient à appliquer le regroupement de tâches monoprocesseur sur chacun des processeurs. Les tâches sont strictement isolées et le regroupement de tâches sur un processeur n'a aucun effet sur les autres.

12.1.1 Processeurs interdépendants

Dans le cas des processeurs interdépendants, il peut exister des contraintes de précédence entre des tâches de différents processeurs. La technique d'ajustement des échéances et des dates de réveil présentée en section 11.3.2 prend en compte cette éventualité et permet d'appliquer les principes de l'ordonnancement monoprocesseur sur chacun des processeurs.

Cependant, contrairement au cas des processeurs indépendants et bien que nous ne regroupions que des tâches placées sur les mêmes processeurs, il est nécessaire de considérer l'ensemble du graphe de tâches lors d'une tentative de regroupement. Par exemple, nous remarquons qu'il est impossible de regrouper les tâches τ_j et τ_k dans la figure 12.1. Ces tâches sont placées sur le même processeur m_1 et en cas de regroupement, les contraintes de précédence qui les lient avec τ_i ne pourraient pas être respectées.

De surcroît, si les processeurs sont interdépendants, les regroupements peuvent également avoir un impact sur l'assignation contraintes de temps de l'ensemble des tâches. En effet, le modèle de regroupement que nous avons introduit dans le cas monoprocesseur amène à modifier

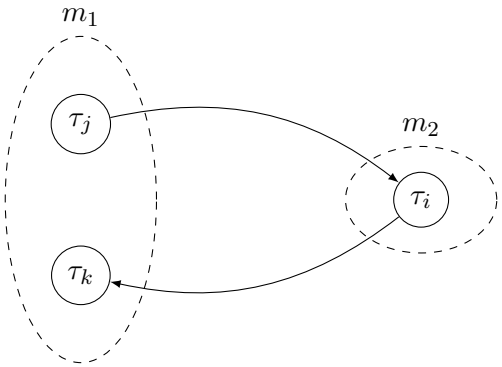
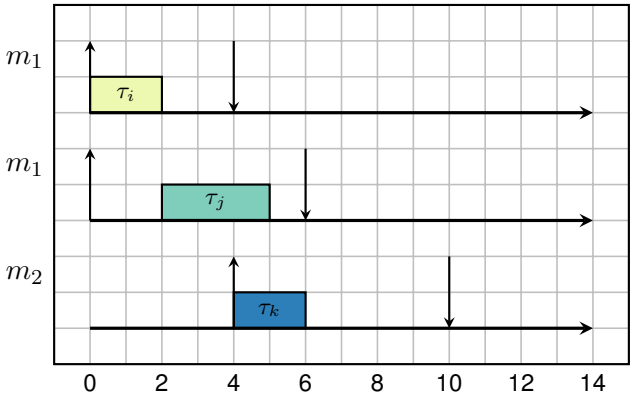
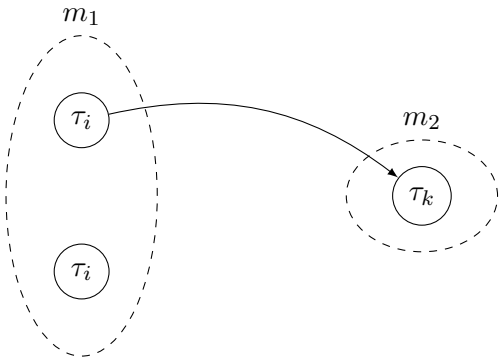


Fig. 12.1 : Regroupement infaisable des tâches τ_j et τ_k du processeur m_1



(a) Diagramme de tâches de l'ensemble $\mathcal{R}$ exécuté sur deux processeurs



(b) Graphe de tâches de l'ensemble $\mathcal{R}$

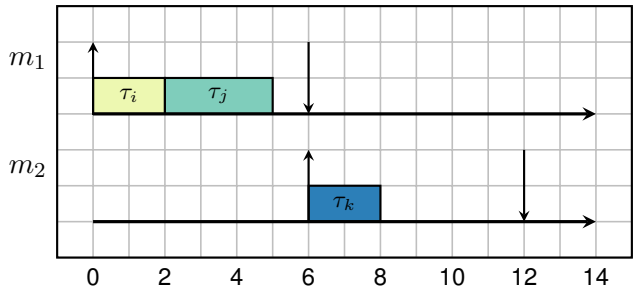
Fig. 12.2 : Ensemble de tâches $\mathcal{R}$ exécuté sur deux processeurs

les échéances des tâches. La modification de l'échéance d'une tâche sur un processeur peut influer sur l'ajustement des échéances et des dates de réveil d'autres tâches éventuellement placées sur des processeurs différents et in fine, modifier l'ordonnançabilité d'autres processeurs. Pour illustrer cette situation, observons la figure 12.2. Elle représente trois tâches, τ_i et τ_j placées sur le processeur m_1 et τ_k située sur le processeur m_2. Nous regroupons τ_i et τ_j sur l'échéance de τ_j. Il apparaît dans la figure 12.3 qu'après ajustement des échéances des dates de réveil que celle de τ_k est repoussée à la date 6, ce qui retarde le début de son exécution. Nous pouvons également observer le caractère pessimiste de la méthode puisque dans ces deux figures, la date de réveil de τ_k aurait pu être fixée à une date antérieure sans entraver l'ordonnançabilité de l'ensemble. Lorsqu'il existe des tâches assignées à des processeurs différents qui entretiennent des relations de précédence, il est nécessaire de considérer le graphe dans son ensemble.

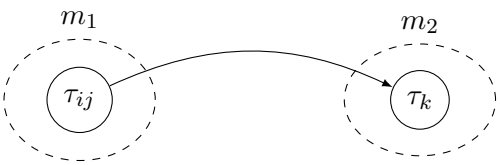
12.2 Minimisation du nombre de tâches

La procédure de minimisation du nombre de tâches regroupe 3 principes dans le contexte multiprocesseur partitionné.

- Du partitionnement : le graphe de tâches est partitionné sur différentes unités de calcul, comme décrit pour l'exemple dans la figure 12.4 ;
- De l'encodage : l'ajustement des contraintes de précédence et des dates de réveil transforme le graphe des tâches sous contraintes de précédence en un ensemble de tâches indé-



(a) Diagramme de tâches de $\mathcal{R}$ après regroupement des tâches τ_i et τ_j exécuté sur deux processeurs



(b) Graphe de tâches de l'ensemble $\mathcal{R}$ après regroupement des tâches τ_i et τ_j en τ_{ij}

FIG. 12.3 : Ensemble de tâches $\mathcal{R}$ après regroupement des tâches τ_i et τ_j en τ_{ij}

pendantes ;

- Du regroupement monoprocesseur : les techniques de regroupement de tâches monoprocesseur peuvent être appliquées sur chacune des unités de calcul.

Remarque 12.2.1 *La complexité du problème reste au moins aussi élevée que pour n'importe quel problème de la classe NP-difficile. En effet, le problème d'ordonnancement de tâches en multiprocesseur est NP-difficile [71] et nous avons montré en section 6.1.2 que le regroupement de tâches était au moins aussi difficile qu'un problème co-NP-complet au sens fort. Or, le regroupement de tâches en multiprocesseur est une combinaison de ces deux problèmes.*

12.2.1 Principe et algorithme

L'algorithme 4 détaille la procédure de regroupement en multiprocesseur. La première étape consiste à partitionner le graphe de tâches dépendantes en flux. Pour ce faire, nous avons légèrement adapté le principe des heuristiques de partitionnement de Buttazo et al. [26]. N'importe quelle autre méthode de partitionnement qui permet de distribuer un ensemble de tâches sur plusieurs unités de calcul pourrait être utilisé à la place.

Une fois les tâches réparties dans les différents flux, nous différencions les flux interdépendants des flux indépendants (méthodes `indépendant` et `dépendant`). Pour chacun des flux indépendants, nous pouvons appliquer directement la méthode de regroupement de tâches avec contraintes de précédence en monoprocesseur `regroupementDep`.

L'étape suivante consiste à regrouper des tâches à l'intérieur de chacun des flux interdépendants. Un regroupement à l'intérieur d'un flux est faisable si après celui-ci et encodage des échéances et des dates de réveil de l'ensemble du graphe initial de tâches, chacun des flux est encore ordonnançable. Les tests d'ordonnançabilité sont appliqués localement mais les dépendances entre les flux imposent de réaliser l'encodage sur l'ensemble du graphe.

Chacun des flux est parcouru successivement, dans chacun d'eux un regroupement qui minimise la fonction de coût locale est réalisé avant de passer au flux suivant. La procédure est répétée jusqu'à qu'il ne soit plus possible de réaliser un regroupement dans aucun des flux. Nous utilisons ici la méthode `regroupementDepNonRec`, une version légèrement modifiée de la méthode de regroupement de tâches avec contraintes de précédence. Celle-ci n'est plus récursive et vérifie la préservation de l'ordonnançabilité de l'ensemble des flux après ajustement des contraintes de temps du graphe de tâches initial. Nous n'appliquons pas récursivement la méthode dans chacun des flux de manière à équilibrer le nombre de tâches par flux après regroupement. La notation $\mathcal{F}_1^{dep}$ décrit le premier élément de l'ensemble ordonné des flux dépendants.

Nous notons qu'une fonction de coût globale à l'ensemble des flux pourrait être combinée à la fonction de coût locale. Son choix n'est pas trivial puisqu'elle peut être contradictoire

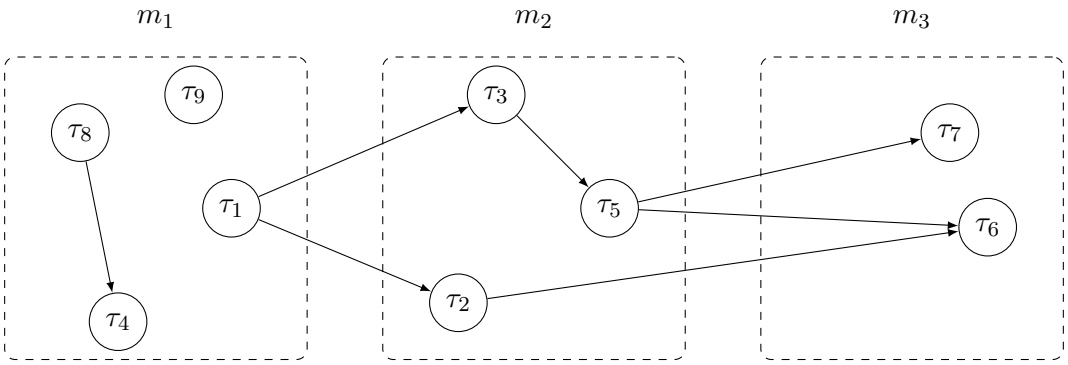


Fig. 12.4 : Partitionnement du graphe de tâches de la figure 8.1 sur trois unités de calcul

avec les objectifs de fonction locale. Par exemple, certains regroupements peuvent augmenter les échéances de tâches ce qui est favorable à l'ordonnançabilité d'un flux $\mathcal{F}_1$. Néanmoins, si cette tâche précède une tâche placée sur un autre flux $\mathcal{F}_2$, cette dernière verra sa date de réveil repoussée après ajustement, ce qui pourrait contraindre l'ordonnançabilité du flux $\mathcal{F}_2$. Pour le moment, cette possibilité n'a pas été évaluée de manière approfondie.

En sortie, l'algorithme produit donc un ensemble de flux sur lesquels le nombre de tâches a été minimisé.

Remarque 12.2.2 *Le regroupement de tâches sur un processeur peut modifier l'ordonnançabilité d'un autre processeur par modification des dates de réveil. Notons que si le test d'ordonnançabilité utilisé est synchrone, le regroupement de tâches peut être effectué de manière indépendante sur chacun des processeurs puisque celui-ci ne sera pas sensible aux modifications des dates de réveil.*

Algorithme 4 Algorithme de l'heuristique de regroupement de tâches en multiprocesseur
Fonction regroupement($\mathcal{G}$)

Entrée : $\mathcal{G} = (\mathbb{V}, \mathbb{E})$: graphe de tâches initial de tâches

$\mathcal{F} \leftarrow$ partitionnement($\mathcal{G}$)
$invalide \leftarrow faux$
Pour Tout $Flux \in \mathcal{F}$ **Faire**
  /*Avec encodage des dates de réveil et des échéances en multiprocesseur*/
  Si ! ordonnançable($Flux$) **Alors**
    $invalide \leftarrow vrai$
  Fin Si
Fin Pour
Si ! $invalide$ **Alors**
  $\mathcal{F}^{ind} \leftarrow$ indépendant($\mathcal{F}$)
  $\mathcal{F}^{dep} \leftarrow$ dépendant($\mathcal{F}$)

  /*Traitement des flux indépendants*/
  Pour Tout $Flux \in \mathcal{F}^{ind}$ **Faire**
    $Flux \leftarrow$regroupementDep($Flux$)
  Fin Pour

  /*Traitement des flux interdépendants*/
  $FluxCourant \leftarrow \mathcal{F}_1^{dep}$
  $idxFlux \leftarrow 1$
  $nbRegroupementsInfaisables \leftarrow 0$
  Tant Que $nbRegroupementsInfaisables \leq |\mathcal{F}^{dep}|$ **Faire**
    $FluxCourant' \leftarrow$regroupementDepNonRec($FluxCourant$)
    Si $|FluxCourant'| = |FluxCourant|$ **Alors** /*Si aucun regroupement n'est faisable*/
      $nbRegroupementsInfaisables \leftarrow nbRegroupementsInfaisables + 1$
    Sinon
      $nbRegroupementsInfaisables = 0$
      $\mathcal{F}^{dep}_{idxFlux} \leftarrow FluxCourant'$
    Fin Si
    $idxFlux \leftarrow idxFlux + 1$
    $FluxCourant \leftarrow \mathcal{F}_{(idxFlux) \pmod{|\mathcal{F}^{dep}|}}$
  Fin Tant Que
Fin Si
Retourne $\mathcal{F}^{ind} \cup \mathcal{F}^{dep}$

12.3 Expérimentations

Dans ce chapitre, nous présentons tout d'abord deux méthodes de génération de taux d'utilisation pour le contexte multiprocesseur. Ensuite, nous exposons les premières expérimentations menées en multiprocesseur à partir de l'heuristique détaillée dans le chapitre précédent. Nous mesurons l'impact du regroupement sur le nombre total de tâches.

12.3.1 Génération de taux d'utilisation

Algorithme UUnifast-Discard

La méthode UUnifast présentée dans le cas monoprocesseur n'est pas utilisée en contexte multiprocesseur, lorsque le taux d'utilisation du processeur u peut dépasser 1. En effet, lorsque que le taux d'utilisation total dépasse 1, UUnifast présente le risque de générer des taux d'utilisation par tâche supérieurs à 1. Tâche qu'il n'est alors possible d'ordonnancer sur aucun processeur. Pour y remédier, Davis et Burns [36] ont proposé une extension appelée *UUnifast-Discard*. Elle consiste simplement à employer UUnifast avec u supérieur à 1 et à rejeter les ensembles pour lesquelles au moins un taux d'utilisation par tâche est supérieur à 1. Son implantation est simple mais cette méthode a l'inconvénient d'être particulièrement inefficace lorsque u approche $\frac{n}{2}$ [45]. Cette limite n'est pas problématique dans notre travail puisque nous traitons des ensembles de plusieurs centaines de tâches. Il y a donc peu de chance que nous approchions ce ratio.

Algorithme Randfixedsum

L'algorithme *Randfixedsum* a été initialement développé par Stafford [100] pour générer des matrices aléatoires. Emerson et al. [45] ont relevé qu'il pouvait être appliqué directement à la génération uniforme de taux d'utilisation par tâche lorsque $u > 1$. L'implantation originelle de Stafford de l'algorithme Randfixedsum est disponible en Matlab et une traduction Python est a été écrite par Paul Emerson[1]. Dans notre prototype, l'algorithme est implanté en Scala à l'aide de la librairie de traitement numérique *Breeze* [57]. Une version moins efficace mais ne nécessitant pas Breeze est également disponible. L'implantation est plus complexe que celle de UUnifast-Discard, notamment car elle requiert nombre d'opérations matricielles. Toutefois, en pratique, l'implantation s'exécute quasi-instantanément pour plusieurs milliers de taux d'utilisation à générer.

Les deux méthodes ci-dessous sont compatibles avec le regroupement de tâches. Arbitrairement, nous utiliserons l'algorithme Randfixedsum.

12.3.2 Variation du nombre total de tâches

L'intérêt majeur du multiprocesseur réside dans sa capacité à traiter des ensembles de tâches avec des taux d'utilisation supérieurs à 1. Nous avons expérimenté l'effet du taux d'utilisation sur la qualité du regroupement selon la méthode de génération de graphe par niveau. Ces résultats sont reproduits dans la figure 12.5. Nous avons choisi les mêmes paramètres de génération que dans la section 10.2.3, c'est-à-dire un nombre maximal de niveaux approchant les 50% du nombre total de tâches par période et une probabilité de contrainte de précédence à 0.25. Nous traçons des moyennes sur 100 échantillons de 200 tâches par point et les échéances sont tirées aléatoirement entre des bornes d'échéances fixées à 0 et 1. Les expérimentations ont été menées sous EDF en utilisant le test QPA réservé aux tâches synchrones. Nous avons fait ce choix pour son efficacité en temps d'exécution, du fait du grand nombre d'échantillons à générer. En pratique, nous conseillons d'employer le test « Offset Analysis » de Pellizzoni et Lipari [83] destiné aux tâches asynchrones. Comme relevé plus tôt, ce dernier est moins pessimiste que l'utilisation d'un test exact pour tâches synchrones. Nous observons que plus le taux d'utilisation est élevé en abscisse,

[1] `http://retis.sssup.it/waters2010/data/taskgen-0.1.tar.gz`

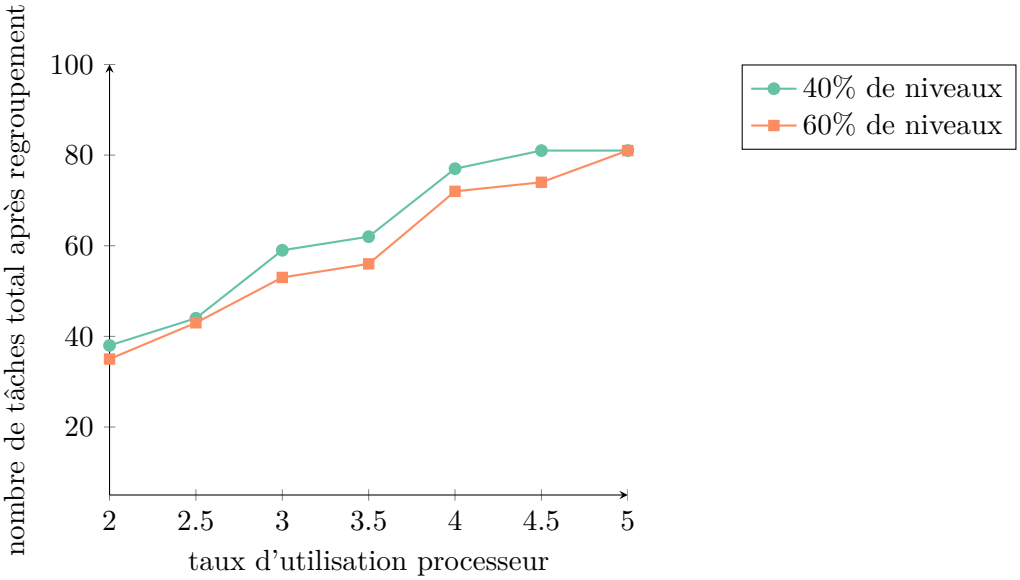


FIG. 12.5 : Impact du taux d'utilisation sur le nombre total de tâches après regroupement (n = 200, p = 0.25, $d_{min} = 0$ et $d_{max} = 1$)

plus le nombre total de tâches augmente en ordonnée. Le nombre total de tâches est égal à la somme du nombre de tâches par flux. Ces résultats sont cohérents puisque la hausse du taux d'utilisation accroît le nombre de flux, ce qui a tendance à grandir le nombre de tâches total. De plus, comme nous ne regroupons que des tâches de même période, il est possible que le nombre minimum de tâches par flux atteigne le nombre de périodes différentes. Une observation plus fine des résultats montre que certains flux sont anormalement hauts. Il apparaît que la méthode de partitionnement crée parfois des flux avec des taux d'utilisation et des densité importants, de telle sorte que le regroupement devient alors très difficile. Il apparaît que la baisse du nombre de niveaux provoque une très légère augmentation du nombre total de tâches après regroupement. Cette effet confirme les observations faites en monoprocesseur sur le faible impact du nombre de niveaux.

Remarques sur l'efficacité de l'algorithme en temps d'exécution

Généralement, l'heuristique est beaucoup plus efficace en temps lorsque le taux d'utilisation processeur est élevé. En effet, plus le taux d'utilisation augmente, plus le nombre de flux est haut et plus le nombre moyen de tâches par flux est bas. En conséquence, l'heuristique doit traiter des ensembles de tâches plus réduit sur chacun des flux et affiche de meilleures performances en temps, de l'ordre de plusieurs dizaines de minutes.

Conclusion de la quatrième partie

Cette dernière partie était consacrée au regroupement de tâches multiprocesseur. Après un bref état de l'art de ce type d'architecture, nous avons choisi de nous orienter vers la stratégie partitionnée qui revient à répartir un ensemble de tâches sur différents processeurs puis à appliquer les méthodes monoprocesseur en leur sein. Bien qu'a priori le problème multiprocesseur se réduise alors à plusieurs problèmes monoprocesseur, nous avons observé que les contraintes de précédence qui peuvent exister entre différents processeurs obligent à considérer l'ensemble du graphe lors des tentatives de regroupement. En effet, il s'avère qu'un regroupement effectué sur un processeur peut avoir des incidences sur l'encodage des contraintes de temps des processeurs liés et donc sur leur ordonnançabilité. Pour traiter le cas multiprocesseur, nous avons adapté la méthode de partitionnement de Buttazo et al. et proposé une heuristique de regroupement qui

considère l'intégralité du graphe de tâches. Nos expérimentations illustrent que le nombre total de tâches après regroupement augmente en même que le taux d'utilisation.

Conclusion et perspectives

Résumé

Dans ce manuscrit, nous avons mis en exergue le surcoût lié à l'implantation d'un grand nombre de tâches sur un système temps réel critique. Ces surcoûts sont notamment liés à la consommation mémoire de l'allocation de chaque tâche vers un thread, ainsi que du nombre de changements de contexte important générés par la présence d'un grand nombre de threads dans le système.

Ce constat nous a orienté vers l'étude des techniques de la littérature visant à réduire ce surcoût. Celles-ci se placent généralement dans le contexte du passage de la conception d'un système temps réel à son implantation dans lequel les threads sont formés par allocation de traitements à grain plus fin. Dans notre travail, nous avons cherché à minimiser le nombre de tâches par regroupement afin de réduire les coûts supplémentaires évoqués. Cet objectif nous a, entre autres, amené à examiner l'impact du regroupement de tâches sur l'ordonnançabilité.

L'étude de la complexité du problème nous a orienté vers des méthodes approchées qui permettent de réduire efficacement le nombre de tâches d'un système temps réel, tout en préservant son ordonnançabilité. La diminution du nombre de tâches obtenue permet d'outrepasser les restrictions des systèmes d'exploitations temps réel qui bien souvent limitent le nombre maximal de tâches et de niveaux de priorité disponibles. Nous avons également montré que le regroupement de tâches permettait de réduire le nombre de changements de contexte globaux.

En particulier, nous avons proposé dans un premier temps des techniques de regroupement de tâches indépendantes sur un unique processeur. Cette étape a permis de saisir la difficulté et les problématiques du regroupement de tâches dans un contexte plus simple. Dans un deuxième temps et dans l'optique de se rapprocher de systèmes concrets, nous avons considéré les dépendances entre les tâches par l'ajout à notre modèle des contraintes de précédences. Finalement, nous avons adapté notre travail au contexte multiprocesseur selon la stratégie partitionnée.

Un prototype développé en Scala rend possible l'utilisation des méthodes de regroupement de tâches ainsi que la reproduction des expérimentations proposées dans ce travail, dans diverses configurations (algorithmes d'ordonnancement, test d'ordonnançabilité, fonction de coût, paramètres des tâches, etc.).

Limitations et perspectives

Minimisation du nombre de tâches

Nous nous sommes restreints à regrouper des tâches de périodes identiques. Cette limitation peut être dépassée en établissant un ordonnancement supplémentaire à l'intérieur des tâches via l'ordonnancement hiérarchique. Lorsque le nombre de périodes différentes est élevé, le regrou-

pement de tâches de périodes différentes permettrait de réduire de manière plus importante le nombre de tâches.

Les expérimentations réalisées sur l'impact des densités démontrent que lorsque les marges sont importantes, le regroupement est dit total. Le nombre de tâches obtenu est alors égal au nombre de périodes différentes présentes. D'un point de vue théorique, une autre perspective consisterait à établir des bornes ou à dériver des analyses suivant lesquelles toutes les tâches de mêmes périodes pourraient être regroupées ensemble d'emblée, sans effectuer d'analyse d'ordonnançabilité supplémentaire.

Nous nous sommes penchés sur la complexité théorique du problème de minimisation du nombre de tâches par regroupement. Il s'avère que le problème est au moins aussi difficile que de résoudre le problème de faisabilité d'un ensemble de tâches, lequel est co-NP-complet au sens fort. Nous avons pour perspective de définir la complexité exacte du problème.

Pire temps d'exécution d'un regroupement

D'un point de vue pratique, l'hypothèse est faite que le pire temps d'exécution d'un regroupement de tâches est égal à la somme des pires temps d'exécution des tâches qui la composent. Cette hypothèse est pessimiste et il serait intéressant d'investiguer les effets concrets du regroupement de tâches dans le cadre de travaux sur la mesure du pire temps d'exécution. Il est en effet possible que le pire temps d'exécution du regroupement soit inférieur, notamment du fait de partage de ressources. Cette information permettrait par exemple de favoriser les regroupements qui minimisent le pire temps d'exécution. La minimisation du pire temps d'exécution serait alors considérée comme un critère objectif, au même titre que la minimisation du nombre de tâches.

Communications et ressources partagées

La présence de communications inter-tâches sont retranscrites dans ce travail à travers les contraintes de précédence. Ces dépendances entre les tâches sont prises en compte aux niveaux de l'ordonnancement et du regroupement. Néanmoins, nous n'avons pas pris en compte le coût des communications qui pourrait constituer, avec le nombre de tâches, un second critère objectif à optimiser. En effet, il est probable que le regroupement de tâches qui communiquent entre elles de manière importante puissent réduire le coût global des communications. Les coûts de communication pourraient être représentés dans notre modèle par des poids assignés aux arcs du graphe.

Les protocoles d'accès aux ressources garantissent un accès exclusif des tâches aux ressources partagées et la cohérence des données manipulées. La protection des ressources partagées induit des temps de blocage qui doivent être considérés par les analyses d'ordonnançabilité. Nous n'avons pas considéré les problématiques d'accès aux ressources partagées dans ce travail.

Multiprocesseur

En contexte multiprocesseur, nous nous sommes orientés vers des méthodes de l'ordonnancement partitionné où les tâches sont définitivement assignées sur des unités de calcul et ordonnancées via des algorithmes monoprocesseur sur chacune d'elles. Nous avons proposé un travail préliminaire ; dans le futur, nous avons pour objectif de comparer plusieurs méthodes de partitionnement et d'investiguer d'autres stratégies de regroupement. Le partitionnement est une manière de restreindre l'espace des tâches qui peuvent être regroupées ensemble. Le rapport entre le regroupement et le partitionnement pourrait être investigué, des regroupements pouvant éventuellement être réalisés en amont du partitionnement. Néanmoins, il convient de remarquer qu'effectuer des regroupements avant la phase de partitionnement autorise moins de flexibilité dans la répartition des tâches sur les processeurs.

Nous avons évoqué les anomalies qui ne permettent pas directement d'appliquer les méthodes de regroupement du contexte monoprocesseur à l'ordonnancement global. Considérer le regrou-

pement de tâches en multiprocesseur selon un ordonnancement global est également une piste à approfondir.

Bibliographie

[1] T. Abdelzaher et K. Shin. « Period-based load partitioning and assignment for large real-time applications ». *IEEE Transactions on Computers* 49.1 (jan. 2000), p. 81–87.

[2] A. Ahmadinia, C. Bobda et J. Teich. « Temporal task clustering for online placement on reconfigurable hardware ». *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*. 2003, p. 359–362.

[3] M. Alras et al. « Model-Based Design of Embedded Control Systems by Means of a Synchronous Intermediate Model ». *International Conference on Embedded Software and Systems, 2009. ICESS '09*. Mai 2009, p. 3–10.

[4] N. C. Audsley et al. *Deadline monotonic scheduling*. 1990.

[5] N. C. Audsley. *Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start Times*. 1991.

[6] N. Audsley et al. « Applying new scheduling theory to static priority pre-emptive scheduling ». *Software Engineering Journal* 8.5 (1993), p. 284.

[7] AUTOSAR. *RTE Standard Specifications*. URL : `http://www.autosar.org/download/R4.0/AUTOSAR_SWS_RTE.pdf`.

[8] T. Baker et S. Baruah. « Sustainable Multiprocessor Scheduling of Sporadic Task Systems ». *21st Euromicro Conference on Real-Time Systems, 2009. ECRTS '09*. Juil. 2009, p. 141–150.

[9] M. Bambagini, G. Buttazzo et S. Hendseth. « Exploiting Uni-Processor Schedulabilty Analysis for Partitioned Task Allocation on Multi-Processors with Precedence Constraints ». *RTSOPS 2012* (2012), p. 17.

[10] S. Baruah. « The limited-preemption uniprocessor scheduling of sporadic task systems ». *17th Euromicro Conference on Real-Time Systems, 2005. (ECRTS 2005). Proceedings*. Juil. 2005, p. 137–144.

[11] S. Baruah et A. Burns. « Sustainable Scheduling Analysis ». *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*. Déc. 2006, p. 159–168.

[12] S. Baruah. « Techniques for Multiprocessor Global Schedulability Analysis ». *Proceedings of the 28th IEEE International Real-Time Systems Symposium*. RTSS '07. Washington, DC, USA : IEEE Computer Society, 2007, p. 119–128.

[13] S. K. Baruah, A. K. Mok et L. E. Rosier. « Preemptively scheduling hard-real-time sporadic tasks on one processor ». *Real-Time Systems Symposium, 1990. Proceedings., 11th*. IEEE, 1990, p. 182–190.

[14] S. K. Baruah, L. E. Rosier et R. R. Howell. « Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor ». *Real-Time Systems* 2.4 (nov. 1990), p. 301–324.

[15] A. Benveniste et G. Berry. « The synchronous approach to reactive and real-time systems ». *Proceedings of the IEEE* 79.9 (1991), p. 1270–1282.

[16] M. Berkelaar, K. Eikland, P. Notebaert et al. « lpsolve : Open source (mixed-integer) linear programming system ». *Eindhoven U. of Technology* (2004).

[17] G. Bernat. « Response Time Analysis of Asynchronous ». *Real-Time Systems* 25.2 (sept. 2003), p. 131–156.

[18] A. Bertout, J. Forget et R. Olejnik. « A Heuristic to Minimize the Cardinality of a Real-time Task Set by Automated Task Clustering ». *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC '14. Gyeongju, Korea : ACM, 2014, p. 1431–1436.

[19] A. Bertout, J. Forget et R. Olejnik. « Minimizing a Real-time Task Set Through Task Clustering ». *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*. RTNS '14. Versailles, France : ACM, 2014, p. 23–31.

[20] A. Bertout, J. Forget et R. Olejnik. « Minimizing the cardinality of a real-time task set by automated task clustering ». Proceedings of the 7th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2013). 16 oct. 2013, p. 9–12.

[21] E. Bini et S. Baruah. « Partitioned Scheduling of Sporadic Task Systems : an ILP-based approach ». EURASIP, 2008.

[22] E. Bini et G. C. Buttazzo. « Measuring the Performance of Schedulability Tests ». *Real-Time Systems* 30.1 (mai 2005), p. 129–154.

[23] A. Burns, K. Tindell et A. Wellings. « Effective analysis for engineering real-time fixed priority schedulers ». *IEEE Transactions on Software Engineering* 21.5 (1995), p. 475–480.

[24] A. Burns. « Preemptive priority-based scheduling : an appropriate engineering approach ». *Advances in real-time systems*. Prentice-Hall, Inc. 1995, p. 225–248.

[25] A. Burns et A. Wellings. *Real-Time Systems and Programming Languages : Ada, Real-Time Java and C/Real-Time POSIX*. 4th. USA : Addison-Wesley Educational Publishers Inc, 2009. isbn : 9780321417459.

[26] G. Buttazzo, E. Bini et Y. Wu. « Partitioning Real-Time Applications Over Multicore Reservations ». *IEEE Transactions on Industrial Informatics* 7.2 (mai 2011), p. 302–315.

[27] G. Buttazzo, M. Bertogna et G. Yao. « Limited Preemptive Scheduling for Real-Time Systems. A Survey ». *IEEE Transactions on Industrial Informatics* 9.1 (fév. 2013), p. 3–15.

[28] G. C. Buttazzo. « Hard Real-Time Computing Systems : Predictable Scheduling Algorithms and Applications ». Springer Science & Business Media, 15 sept. 2011. Chap. 10. isbn : 9781461406761.

[29] G. C. Buttazzo. « Rate Monotonic vs. EDF : Judgment Day ». *Embedded Software*. Sous la dir. de R. Alur et I. Lee. Lecture Notes in Computer Science 2855. Springer Berlin Heidelberg, 2003, p. 67–83.

[30] M. Chéramy. « Etude et évaluation de politiques d'ordonnancement temps réel multiprocesseur ». Thèse de doct. Toulouse, INSA, 2014.

[31] M. Chéramy, P.-E. Hladik et A.-M. Déplanche. « SimSo : A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms ». *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. Madrid, Spain, juil. 2014.

[32] H. Chetto, M. Silly et T. Bouchentouf. « Dynamic scheduling of real-time tasks under precedence constraints ». *Real-Time Systems* 2.3 (sept. 1990), p. 181–194.

[33] D. Cordeiro et al. « Random Graph Generation for Scheduling Simulations ». *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. SIMUTools '10. ICST, Brussels, Belgium, Belgium : ICST (Institute for Computer Sciences, Social-Informatics et Telecommunications Engineering), 2010.

[34] CPLEX. « IBM ILOG. V12. 1 : User's Manual for CPLEX ». *International Business Machines Corporation* 46.53 (2009), p. 157.

[35] A. Curic. « Implementing Lustre Programs on Distributed Platforms with Real-time Constrains ». 2005.

[36] R. Davis et A. Burns. « Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems ». *30th IEEE Real-Time Systems Symposium, 2009, RTSS 2009*. Déc. 2009, p. 398–409.

[37] R. I. Davis et A. Burns. « A Survey of Hard Real-time Scheduling for Multiprocessor Systems ». *ACM Comput. Surv.* 43.4 (oct. 2011), p. 35–44.

[38] R. Davis, N. Merriam et N. Tracey. « How embedded applications using an rtos can stay within on-chip memory limits ». *In Proceedings of the Work in Progress and Industrial Experience Session, Euromicro Conference on RealTime Systems*. 2000, p. 43–50.

[39] M. Dertouzos. « Control Robotics : The Procedural Control of Physical Processes ». Proceedings of IFIP Congress (IFIP'74). 1974, p. 807–813.

[40] U. Devi. « An improved schedulability test for uniprocessor periodic task systems ». *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings*. Juil. 2003, p. 23–30.

[41] R. Devillers et J. Goossens. « General Response Time Computation for the Deadline Driven Scheduling of Periodic Tasks ». *Fundam. Inf.* 40.2 (août 1999), p. 199–219.

[42] R. P. Dick, D. L. Rhodes et W. Wolf. « TGFF : Task Graphs for Free ». *Proceedings of the 6th International Workshop on Hardware/Software Codesign*. CODES/CASHE '98. Washington, DC, USA : IEEE Computer Society, 1998, p. 97–101.

[43] J. Echague, I. Ripoll et A. Crespo. « Hard real-time preemptively scheduling with high context switch cost ». *Seventh Euromicro Workshop on Real-Time Systems, 1995. Proceedings*. 1995, p. 184–190.

[44] P. Ekberg et W. Yi. « Uniprocessor Feasibility of Sporadic Tasks with Constrained Deadlines Is Strongly coNP-Complete ». *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*. Juil. 2015, p. 281–286.

[45] P. Emberson, R. Stafford et R. I. Davis. « Techniques for the synthesis of multiprocessor tasksets ». *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, p. 6–11.

[46] P. Erdős et A. Rényi. « On random graphs I. » *Publ. Math. Debrecen* 6 (1959), p. 290–297.

[47] H. Faragardi, B. Lisper et T. Nolte. « Towards a communication-efficient mapping of AUTOSAR runnables on multi-cores ». *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*. Sept. 2013, p. 1–5.

[48] J. Forget et al. « Scheduling dependent periodic tasks without synchronization mechanisms ». *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*. IEEE. 2010, p. 301–310.

[49] *FreeRTOS*. url : `http://www.freertos.org/`.

[50] P. B. Galvin, G. Gagne et A. Silberschatz. *Operating System Concepts*. 9th. New York, NY, USA : John Wiley &amp ; Sons, Inc., 2013. isbn : 9781118093757.

[51] M. R. Garey et D. S. Johnson. « Computers and intractability : a guide to the theory of NP-completeness. 1979 ». *San Francisco, LA : Freeman* (1979).

[52] J. Goossens et C. Macq. « Limitation of the Hyper-Period in Real-Time Periodic Task Set Generation ». *In Proceedings of the RTS Embedded System (RTS'01)*. 2001, p. 133–147.

[53] J. Goossens et P. Richard. « Ordonnancement temps réel multiprocesseur ». Août 2013.

[54] N. Guan et W. Yi. « General and Efficient Response Time Analysis for EDF Scheduling ». *Proceedings of the Conference on Design, Automation & Test in Europe*. DATE '14. 3001 Leuven, Belgium, Belgium : European Design et Automation Association, 2014.

[55] L. Guodong et al. « Task Clustering and Scheduling to Multiprocessors with Duplication ». *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IPDPS '03. Washington, DC, USA : IEEE Computer Society, 2003.

[56] N. Halbwachs et al. « The synchronous data flow programming language LUSTRE ». *Proceedings of the IEEE* 79.9 (sept. 1991), p. 1305–1320.

[57] D. Hall. *Breeze library*. `https://github.com/scalanlp/breeze`. 2009.

[58] D. S. Johnson. « Fast allocation algorithms ». *Switching and Automata Theory, 1972., IEEE Conference Record of 13th Annual Symposium on*. IEEE. 1972, p. 144–154.

[59] M. Joseph et P. Pandya. « Finding Response Times in a Real-Time System ». *The Computer Journal* 29.5 (1986), p. 390–395.

[60] A. B. Kahn. « Topological Sorting of Large Networks ». *Commun. ACM* 5.11 (nov. 1962), p. 558–562.

[61] S. Kim, S. Cho et S. Hong. « Schedulability-aware mapping of real-time object-oriented models to multi-threaded implementations ». *Seventh International Conference on Real-Time Computing Systems and Applications, 2000. Proceedings*. 2000, p. 7–14.

[62] V. King. « Fully Dynamic Transitive Closure ». *Encyclopedia of Algorithms*. Sous la dir. de M.-Y. K. P. o. C. Science. Springer US, 2008, p. 1–99.

[63] L. B. Kish. « End of Moore's law : thermal (noise) death of integration in micro and nano electronics ». *Physics Letters A* 305.3 (2 déc. 2002), p. 144–149.

[64] S. Kodase, S. Wang et K. G. Shin. « Transforming Structural Model to Runtime Model of Embedded Software with Real-Time Constraints ». *Proceedings of the Conference on Design, Automation and Test in Europe : Designers' Forum - Volume 2*. DATE '03. Washington, DC, USA : IEEE Computer Society, 2003.

[65] E. Koutsofios, S. North et al. *Drawing graphs with dot*. Rapp. tech.

[66] J. J. Labrosse. *uC/OS-III, The Real-Time Kernel, or a High Performance, Scalable, ROMable, Preemptive, Multitasking Kernel for Microprocessors, Microcontrollers & DSPs*. USA : Micrium Press, 2009. isbn : 9780982337530.

[67] W. Lamie. « Preemption threshold ». *White paper, Express Logic. Available online* (1997).

[68] E. L. Lawler et D. E. Wood. « Branch-and-bound methods : A survey ». *Operations research* 14.4 (1966), p. 699–719.

[69] C.-G. Lee et al. « Analysis of cache-related preemption delay in fixed-priority preemptive scheduling ». *Computers, IEEE Transactions on* 47.6 (1998), p. 700–713.

[70] J. Y.-T. Leung et M. Merrill. « A note on preemptive scheduling of periodic, real-time tasks ». *Information Processing Letters* 11.3 (18 nov. 1980), p. 115–118.

[71] J. Y.-T. Leung et J. Whitehead. « On the complexity of fixed-priority scheduling of periodic, real-time tasks ». *Performance Evaluation* 2.4 (déc. 1982), p. 237–250.

[72] G. Levin et al. « DP-FAIR : A Simple Model for Understanding Optimal Multiprocessor Scheduling ». *2010 22nd Euromicro Conference on Real-Time Systems (ECRTS)*. Juil. 2010, p. 3–13.

[73] G. Lipari. *rtsched – Draw Real-Time scheduling (GANT) charts*. Version 2.0. 2015. url : `https://github.com/glipari/rtsched`.

[74] C. L. Liu et J. W. Layland. « Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment ». *Journal of the ACM* 20.1 (jan. 1973), p. 46–61.

[75] A. Mok, X. Feng et D. Chen. « Resource partition for real-time systems ». *Seventh IEEE Real-Time Technology and Applications Symposium, 2001. Proceedings*. 2001, p. 75–84.

[76] A. Monot et al. « Multisource Software on Multicore Automotive ECUs - Combining Runnable Sequencing With Task Scheduling ». *IEEE Transactions on Industrial Electronics* 59.10 (oct. 2012), p. 3934–3942.

[77] R. Mzid et al. « DPMP : A Software Pattern for Real-time Tasks Merge ». *Proceedings of the 9th European Conference on Modelling Foundations and Applications*. ECMFA'13. Berlin, Heidelberg : Springer-Verlag, 2013, p. 101–117.

[78] F. Ndoye. « Ordonnancement temps réel préemptif multiprocesseur avec prise en compte du coût du système d'exploitation ». Thèse de doct. Paris 11, 2014.

[79] M. Odersky, L. Spoon et B. Venners. *Programming in Scala, 2/e*. Artima Series. Artima Press, 2010. isbn : 9780981531649.

[80] C. Pagetti et al. « Multi-task Implementation of Multi-periodic Synchronous Programs ». *Discrete Event Dynamic Systems* 21.3 (2011), p. 307–338.

[81] J. Palencia et M. Harbour. « Offset-based response time analysis of distributed systems scheduled under EDF ». *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings*. Juil. 2003, p. 3–12.

[82] M. Palis, J.-C. Liou et D. Wei. « Task clustering and scheduling for distributed memory parallel architectures ». *Parallel and Distributed Systems, IEEE Transactions on* 7.1 (jan. 1996), p. 46–55.

[83] R. Pellizzoni et G. Lipari. « Feasibility Analysis of Real-Time Periodic Tasks with Offsets ». *Real-Time Syst.* 30.1 (mai 2005), p. 105–128.

[84] R. Pellizzoni et G. Lipari. « Holistic Analysis of Asynchronous Real-time Transactions with Earliest Deadline Scheduling ». *J. Comput. Syst. Sci.* 73.2 (mar. 2007), p. 186–206.

[85] D.-T. Peng et K. Shin. « Static allocation of periodic tasks with precedence constraints in distributed real-time systems ». *9th International Conference on Distributed Computing Systems, 1989*. Juin 1989, p. 190–198.

[86] K. Ramamritham. « Allocation and Scheduling of Precedence-Related Periodic Tasks ». *IEEE Trans. Parallel Distrib. Syst.* 6.4 (1995), p. 412–420.

[87] M. Richard et al. « Contraintes de précédences et ordonnancement mono-processeur ». *Real-time and embedded systems (RTS'02)* (2002).

[88] G.-C. Rota. « The number of partitions of a set ». *The American Mathematical Monthly* 71.5 (1964), p. 498–504.

[89] *RTEMS C User Guide Edition 4.10.99.0*. 24 mar. 2013. url : `https://docs.rtems.org`.

[90] J. Rumbaugh, I. Jacobson et G. Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[91] M. Saksena, P. Karvelas et Y. Wang. « Automatic synthesis of multi-tasking implementations from real-time object-oriented models ». *Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2000. (ISORC 2000) Proceedings*. 2000, p. 360–367.

[92] M. Saksena et Y. Wang. « Scalable real-time system design using preemption thresholds ». *The 21st IEEE Real-Time Systems Symposium, 2000. Proceedings*. 2000, p. 25–34.

[93] L. Santinelli et al. « Scheduling with functional and non-functional requirements : the sub-functional approach ». *Work-in-Progress Session of ECRTS* 2 (2013), p. 9.

[94] O. Scheickl et M. Rudorfer. « Automotive real time development using a timing-augmented AUTOSAR specification ». *Proceedings of ERTS2008* 4 (2008).

[95] S. Schliecker et al. « System Level Performance Analysis for Real-Time Automotive Multicore and Network Architectures ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.7 (2009), p. 979–992.

[96] B. Selic. « Using UML for modeling complex real-time systems ». *Languages, Compilers, and Tools for Embedded Systems*. Springer Berlin Heidelberg, 1998, p. 250–260.

[97] F. Singhoff et al. « Cheddar : A Flexible Real Time Scheduling Framework ». *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada : The Engineering of Correct and Reliable Software for Real-time & Distributed Systems Using Ada and Related Technologies*. Atlanta, Georgia, USA : ACM, 2004, p. 1–8.

[98] N. J. A. Sloane. *The On-Line Encyclopedia of Integer Sequences*. URL : `http://oeis.org/A000292A000292`.

[99] M. Spuri. *Analysis of Deadline Scheduled Real-Time Systems*. Research report RR-2772. REFLECS Project. INRIA, 1996.

[100] R. Stafford. *Random vectors with fixed sum*. 2006. URL : `http://www.mathworks.com/matlabcentral/fileexchange/9700`.

[101] K. Tindell et J. Clark. « Holistic Schedulability Analysis for Distributed Hard Real-time Systems ». *Microprocess. Microprogram.* 40.2 (avr. 1994), p. 117–134.

[102] *UML Profile for MARTE : Modeling and Analysis of Real-time Embedded Systems*. URL : `http://www.omg.org/spec/MARTE/index.htm`.

[103] Y. Wang et M. Saksena. « Scheduling fixed-priority tasks with preemption threshold ». *Sixth International Conference on Real-Time Computing Systems and Applications, 1999. RTCSA '99*. 1999, p. 328–335.

[104] S. Warshall. « A Theorem on Boolean Matrices ». *J. ACM* 9.1 (jan. 1962), p. 11–12.

[105] *Wind River VxWorks Programmer's guide 6.7*. 17 nov. 2008. URL : `http://www.windriver.com/products/vxworks/`.

[106] E. Wozniak et al. « An optimization approach for the synthesis of AUTOSAR architectures ». *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*. Sept. 2013, p. 1–10.

[107] Y. Wu, Z. Gao et G. Dai. « Deadline and activation time assignment for partitioned real-time application on multiprocessor reservations ». *Journal of Systems Architecture* 60.3 (2014), p. 247–257.

[108] H. Zeng et M. Di Natale. « An Efficient Formulation of the Real-Time Feasibility Region for Design Optimization ». *IEEE Transactions on Computers* 62.4 (avr. 2013), p. 644–661.

[109] H. Zeng et M. Di Natale. « Efficient implementation of AUTOSAR components with minimal memory usage ». *2012 7th IEEE International Symposium on Industrial Embedded Systems (SIES)*. Juin 2012, p. 130–137.

[110] F. Zhang et A. Burns. « Improvement to Quick Processor-Demand Analysis for EDF-Scheduled Real-Time Systems ». *21st Euromicro Conference on Real-Time Systems, 2009. ECRTS '09*. Juil. 2009, p. 76–86.

[111] M. Zhang et Z. Gu. « Optimization issues in mapping AUTOSAR components to distributed multithreaded implementations ». *2011 22nd IEEE International Symposium on Rapid System Prototyping (RSP)*. Mai 2011, p. 23–29.